Software

Design Engineering and Architecture

(Concepts & Applications)

Leigha Whiteside

Amado Dexter

First Edition, 2012

ISBN 978-81-323-0986-4

Published by:
Academic Studio
4735/22 Prakashdeep Bldg,
Ansari Road, Darya Ganj,
Delhi - 110002
Email: info@wtbooks.com

Table of Contents

Chapter 1 - Design Pattern (Computer Science)

Chapter 2 - Software Architecture

Chapter 3 - Data Modeling

Chapter 4 - Unified Modeling Language

Chapter 5 - Design Rationale

Chapter 6 - Domain–Driven Design

Chapter 7 - Object–Oriented Design

Chapter 8 - Structured Analysis

Chapter 9 - Shlaer–Mellor Method

Chapter 10 - Object–Oriented Analysis and Design

Chapter 11 - Architecture Description Language

Chapter 12 - Enterprise Architecture Frameworks

Chapter 13 - Event–Driven Architecture and Space–Based Architecture

Chapter 14 - Software Architect and Software Architectural Model

Chapter 15 - Functional Software Architecture and COLA (Software Architecture)

Chapter 16 - IEEE 1471 and Representational State Transfer

Chapter 1

Design Pattern (Computer Science)

In software engineering, a **design pattern** is a general reusable solution to a commonly occurring problem in software design. A design pattern is not a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations. Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved. Many patterns imply object-orientation or more generally mutable state, and so may not be as applicable in functional programming languages, in which data is immutable or treated as such.

Design patterns reside in the domain of modules and interconnections. At a higher level there are architectural patterns that are larger in scope, usually describing an overall pattern followed by an entire system.

There are many types of design patterns, like

- **Algorithm strategy patterns** addressing concerns related to high-level strategies describing how to exploit application characteristic on a computing platform.
- **Computational design patterns** addressing concerns related to key computation identification.
- **Execution patterns** that address concerns related to supporting application execution, including strategies in executing streams of tasks and building blocks to support task synchronization.
- **Implementation strategy patterns** addressing concerns related to implementing source code to support

1. program organization, and
2. the common data structures specific to parallel programming.

- **Structural design patterns** addressing concerns related to high-level structures of applications being developed.

Practice

Design patterns can speed up the development process by providing tested, proven development paradigms. Effective software design requires considering issues that may not become visible until later in the implementation. Reusing design patterns helps to prevent subtle issues that can cause major problems, and it also improves code readability for coders and architects who are familiar with the patterns.

In order to achieve flexibility, design patterns usually introduce additional levels of indirection, which in some cases may complicate the resulting designs and hurt application performance.

By definition, a pattern must be programmed anew into each application that uses it. Since some authors see this as a step backward from software reuse as provided by components, researchers have worked to turn patterns into components. Meyer and Arnout were able to provide full or partial componentization of two-thirds of the patterns they attempted.

Often, people only understand how to apply certain software design techniques to certain problems. These techniques are difficult to apply to a broader range of problems. Design patterns provide general solutions, documented in a format that does not require specifics tied to a particular problem.

Structure

Design patterns are composed of several sections. Of particular interest are the Structure, Participants, and Collaboration sections. These sections describe a *design motif*: a prototypical *micro-architecture* that developers copy and adapt to their particular designs to solve the recurrent problem described by the design pattern. A micro-architecture is a set of program constituents (e.g., classes, methods...) and their relationships. Developers use the design pattern by introducing in their designs this prototypical micro-architecture, which means that micro-architectures in their designs will have structure and organization similar to the chosen design motif.

In addition to this, patterns allow developers to communicate using well-known, well understood names for software interactions. Common design patterns can be improved over time, making them more robust than *ad-hoc* designs.

Domain-specific patterns

Efforts have also been made to codify design patterns in particular domains, including use of existing design patterns as well as domain specific design patterns. Examples include user interface design patterns, information visualization, secure design, "secure usability", Web design and business model design.

The annual Pattern Languages of Programming Conference proceedings include many examples of domain specific patterns.

Classification and list

Design patterns were originally grouped into the categories: creational patterns, structural patterns, and behavioral patterns, and described using the concepts of delegation, aggregation, and consultation. For further background on object-oriented design, see coupling and cohesion, inheritance, interface, and polymorphism. Another classification has also introduced the notion of architectural design pattern that may be applied at the architecture level of the software such as the Model–View–Controller pattern.

Name	Description	In *Design Patterns*	In *Code Complete*	Other
Creational patterns				
Abstract factory	Provide an interface for creating families of related or dependent objects without specifying their concrete classes.	Yes	Yes	N/A
Builder	Separate the construction of a complex object from its representation allowing the same construction process to create various representations.	Yes	No	N/A
Factory method	Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.	Yes	Yes	N/A
Lazy initialization	Tactic of delaying the creation of an object, the calculation of a value, or some other expensive process until the first time it is needed.	No	No	*PoEAA*
Multiton	Ensure a class has only named instances, and provide global point of access to them.	No	No	N/A
Object pool	Avoid expensive acquisition and release of resources by recycling objects that are no longer in use. Can be considered a generalisation of connection pool and thread pool patterns.	No	No	N/A
Prototype	Specify the kinds of objects to create using a prototypical	Yes	No	N/A

	instance, and create new objects by copying this prototype.			
Resource acquisition is initialization	Ensure that resources are properly released by tying them to the lifespan of suitable objects.	No	No	N/A
Singleton	Ensure a class has only one instance, and provide a global point of access to it.	Yes	Yes	N/A
Structural patterns				
Adapter or Wrapper	Convert the interface of a class into another interface clients expect. An adapter lets classes work together that could not otherwise because of incompatible interfaces.	Yes	Yes	N/A
Bridge	Decouple an abstraction from its implementation allowing the two to vary independently.	Yes	Yes	N/A
Composite	Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.	Yes	Yes	N/A
Decorator	Attach additional responsibilities to an object dynamically keeping the same interface. Decorators provide a flexible alternative to subclassing for extending functionality.	Yes	Yes	N/A
Facade	Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.	Yes	Yes	N/A
Front Controller	Provide a unified interface to a set of interfaces in a subsystem. Front Controller defines a higher-level interface that makes the subsystem easier to use.	No	Yes	N/A
Flyweight	Use sharing to support large	Yes	No	N/A

	numbers of fine-grained objects efficiently.			
Proxy	Provide a surrogate or placeholder for another object to control access to it.	Yes	No	N/A
Behavioral patterns				
Blackboard	Generalized observer, which allows multiple readers and writers. Communicates information system-wide.	No	No	N/A
Chain of responsibility	Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.	Yes	No	N/A
Command	Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.	Yes	No	N/A
Interpreter	Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.	Yes	No	N/A
Iterator	Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.	Yes	Yes	N/A
Mediator	Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.	Yes	No	N/A
Memento	Without violating encapsulation, capture and externalize an object's internal state allowing	Yes	No	N/A

	the object to be restored to this state later.			
Null object	Avoid null references by providing a default object.	No	No	N/A
Observer or Publish/subscribe	Define a one-to-many dependency between objects where a state change in one object results with all its dependents being notified and updated automatically.	Yes	Yes	N/A
Servant	Define common functionality for a group of classes	No	No	N/A
Specification	Recombinable business logic in a Boolean fashion	No	No	N/A
State	Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.	Yes	No	N/A
Strategy	Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.	Yes	Yes	N/A
Template method	Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.	Yes	Yes	N/A
Visitor	Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.	Yes	No	N/A

Name	Description	In *POSA2*	Other
Concurrency patterns			
Active Object	Decouples method execution from method invocation that reside in their own thread of	Yes	N/A

	control. The goal is to introduce concurrency, by using asynchronous method invocation and a scheduler for handling requests.		
Balking	Only execute an action on an object when the object is in a particular state.	No	N/A
Binding properties	Combining multiple observers to force properties in different objects to be synchronized or coordinated in some way.	No	N/A
Messaging design pattern (MDP)	Allows the interchange of information (i.e. messages) between components and applications.	No	N/A
Double-checked locking	Reduce the overhead of acquiring a lock by first testing the locking criterion (the 'lock hint') in an unsafe manner; only if that succeeds does the actual lock proceed. Can be unsafe when implemented in some language/hardware combinations. It can therefore sometimes be considered an anti-pattern.	Yes	N/A
Event-based asynchronous	Addresses problems with the asynchronous pattern that occur in multithreaded programs.	No	N/A
Guarded suspension	Manages operations that require both a lock to be acquired and a precondition to be satisfied before the operation can be executed.	No	N/A
Lock	One thread puts a "lock" on a resource, preventing other threads from accessing or modifying it.	No	*PoEAA*
Monitor object	An object whose methods are subject to mutual exclusion, thus preventing multiple objects from erroneously trying to use it at the same time.	Yes	N/A
Reactor	A reactor object provides an asynchronous interface to resources that must be handled synchronously.	Yes	N/A
Read-write lock	Allows concurrent read access to an object, but requires exclusive access for write operations.	No	N/A
Scheduler	Explicitly control when threads may execute single-threaded code.	No	N/A
Thread pool	A number of threads are created to perform a number of tasks, which are usually organized in a queue. Typically, there are many more tasks than threads. Can be considered a special case of the object pool pattern.	No	N/A

Thread-specific storage	Static or "global" memory local to a thread.	Yes	N/A

Documentation

The documentation for a design pattern describes the context in which the pattern is used, the forces within the context that the pattern seeks to resolve, and the suggested solution. There is no single, standard format for documenting design patterns. Rather, a variety of different formats have been used by different pattern authors. However, according to Martin Fowler, certain pattern forms have become more well-known than others, and consequently become common starting points for new pattern-writing efforts. One example of a commonly used documentation format is the one used by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides (collectively known as the "Gang of Four", or GoF for short) in their book *Design Patterns*. It contains the following sections:

- **Pattern Name and Classification:** A descriptive and unique name that helps in identifying and referring to the pattern.
- **Intent:** A description of the goal behind the pattern and the reason for using it.
- **Also Known As:** Other names for the pattern.
- **Motivation (Forces):** A scenario consisting of a problem and a context in which this pattern can be used.
- **Applicability:** Situations in which this pattern is usable; the context for the pattern.
- **Structure:** A graphical representation of the pattern. Class diagrams and Interaction diagrams may be used for this purpose.
- **Participants:** A listing of the classes and objects used in the pattern and their roles in the design.
- **Collaboration:** A description of how classes and objects used in the pattern interact with each other.
- **Consequences:** A description of the results, side effects, and trade offs caused by using the pattern.
- **Implementation:** A description of an implementation of the pattern; the solution part of the pattern.
- **Sample Code:** An illustration of how the pattern can be used in a programming language.
- **Known Uses:** Examples of real usages of the pattern.
- **Related Patterns:** Other patterns that have some relationship with the pattern; discussion of the differences between the pattern and similar patterns.

Criticism

The concept of design patterns has been criticized in several ways.

The design patterns may be just sign a some missing features of a given programming language (Java or C++ for instance). Peter Norvig demonstrates that 16 out of the 23 patterns in the Design Patterns book (which is primarily focused on C++) are simplified or eliminated (via direct language support) in Lisp or Dylan.

The idea may not be as new as suggested by the authors: for instance the Model-View-Controller paradigm is an example of a "pattern" which predates the concept of "design patterns" by several years.

Chapter 2

Software Architecture

The **software architecture** of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both. The term also refers to documentation of a system's software architecture. Documenting software architecture facilitates communication between stakeholders, documents early decisions about high-level design, and allows reuse of design components and patterns between projects.

Overview

The field of computer science has come across problems associated with complexity since its formation. Earlier problems of complexity were solved by developers by choosing the right data structures, developing algorithms, and by applying the concept of separation of concerns. Although the term "software architecture" is relatively new to the industry, the fundamental principles of the field have been applied sporadically by software engineering pioneers since the mid 1980s. Early attempts to capture and explain software architecture of a system were imprecise and disorganized, often characterized by a set of box-and-line diagrams. During the 1990s there was a concentrated effort to define and codify fundamental aspects of the discipline. Initial sets of design patterns, styles, best practices, description languages, and formal logic were developed during that time.

The software architecture discipline is centered on the idea of reducing complexity through abstraction and separation of concerns. To date there is still no agreement on the precise definition of the term "software architecture".

As a maturing discipline with no clear rules on the right way to build a system, designing software architecture is still a mix of art and science. The "art" aspect of software architecture is because a commercial software system supports some aspect of a business or a mission. How a system supports key business drivers is described via scenarios as non-functional requirements of a system, also known as quality attributes, determine how a system will behave. Every system is unique due to the nature of the business drivers it supports, as such the degree of quality attributes exhibited by a system such as fault-tolerance, backward compatibility, extensibility, reliability, maintainability, availability, security, usability, and such other –ilities will vary with each implementation. To bring a software architecture user's perspective into the software architecture, it can be said that software architecture gives the direction to take steps and do the tasks involved in each such user's speciality area and interest e.g. the stakeholders of software systems, the software developer, the software system operational support group, the software maintenance specialists, the deployer, the tester and also the business end user. In this sense software architecture is really the amalgamation of the multiple perspectives a system always embodies. The fact that those several different perspectives can be put together into a software architecture stands as the vindication of the need and justification of creation of software architecture before the software development in a project attains maturity.

History

The origin of software architecture as a concept was first identified in the research work of Edsger Dijkstra in 1968 and David Parnas in the early 1970s. These scientists emphasized that the structure of a software system matters and getting the structure right is critical. The study of the field increased in popularity since the early 1990s with research work concentrating on architectural styles (patterns), architecture description languages, architecture documentation, and formal methods.

Research institutions have played a prominent role in furthering software architecture as a discipline. Mary Shaw and David Garlan of Carnegie Mellon wrote a book titled *Software Architecture: Perspectives on an Emerging Discipline* in 1996, which brought forward the concepts in Software Architecture, such as components, connectors, styles and so on. The University of California, Irvine's Institute for Software Research's efforts in software architecture research is directed primarily in architectural styles, architecture description languages, and dynamic architectures.

The IEEE 1471: ANSI/IEEE 1471-2000: Recommended Practice for Architecture Description of Software-Intensive Systems is the first formal standard in the area of software architecture, and was adopted in 2007 by ISO as *ISO/IEC 42010:2007.*

Software architecture topics

Architecture description languages

Architecture description languages (**ADLs**) are used to describe a Software Architecture. Several different ADLs have been developed by different organizations, including AADL (SAE standard), Wright (developed by Carnegie Mellon), Acme (developed by Carnegie Mellon), xADL (developed by UCI), Darwin (developed by Imperial College London), DAOP-ADL (developed by University of Málaga), and ByADL (University of L'Aquila, Italy). Common elements of an ADL are component, connector and configuration.

Views

Software architecture is commonly organized in views, which are analogous to the different types of blueprints made in building architecture. A view is a representation of a set of system components and relationships among them. Within the ontology established by ANSI/IEEE 1471-2000, *views* are responses to *viewpoints*, where a viewpoint is a specification that describes the architecture in question from the perspective of a given set of stakeholders and their concerns. The viewpoint specifies not only the concerns addressed but the presentation, model kinds used, conventions used and any consistency (correspondence) rules to keep a view consistent with other views.

Some possible views (actually, *viewpoints* in the 1471 ontology) are:

- Functional/logic view
- Code/module view
- Development/structural view
- Concurrency/process/runtime/thread view
- Physical/deployment/install view
- User action/feedback view
- Data view/data model

Several languages for describing software architectures ('architecture description language' in ISO/IEC 42010 / IEEE-1471 terminology) have been devised, but no consensus exists on which symbol-set or language should be used to describe each architecture view. The UML is a standard that can be used *"for analysis, design, and implementation of software-based systems as well as for modeling business and similar processes."* Thus, the UML is a visual language that can be used to create software architecture views.

Architecture frameworks

Frameworks related to the domain of software architecture are:

- 4+1
- RM-ODP (Reference Model of Open Distributed Processing)
- Service-Oriented Modeling Framework (SOMF)

Other architectures such as the Zachman Framework, DODAF, and TOGAF relate to the field of Enterprise architecture.

The distinction from functional design

The IEEE Std 610.12-1990 Standard Glossary of Software Engineering Terminology defines the following distinctions:

- Architectural Design: the process of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a computer system.
- Detailed Design: the process of refining and expanding the preliminary design of a system or component to the extent that the design is sufficiently complete to begin implementation.

- Functional Design: the process of defining the working relationships among the components of a system.
- Preliminary Design: the process of analyzing design alternatives and defining the architecture, components, interfaces, and timing/sizing estimates for a system or components.

Software architecture, also described as strategic design, is an activity concerned with global requirements governing *how* a solution is implemented such as programming paradigms, architectural styles, component-based software engineering standards, architectural patterns, security, scale, integration, and law-governed regularities. Functional design, also described as tactical design, is an activity concerned with local requirements governing *what* a solution does such as algorithms, design patterns, programming idioms, refactorings, and low-level implemenation.

According to the Intension/Locality Hypothesis, the distinction between architectural and detailed design is defined by the Locality Criterion, according to which a statement about software design is non-local (architectural) if and only if a program that satisfies it can be expanded into a program which does not. For example, the client–server style is architectural (strategic) because a program that is built on this principle can be expanded into a program which is not client–server; for example, by adding peer-to-peer nodes.

Architecture is design but not all design is architectural. In practice, the architect is the one who draws the line between software architecture (architectural design) and detailed design (non-architectural design). There aren't rules or guidelines that fit all cases. Examples of rules or heuristics that architects (or organizations) can establish when they want to distinguish between architecture and detailed design include:

- Architecture is driven by non-functional requirements, while functional design is driven by functional requirements.
- Pseudo-code belongs in the detailed design document.
- UML component, deployment, and package diagrams generally appear in software architecture documents; UML class, object, and behavior diagrams appear in detailed functional design documents.

Examples of architectural styles and patterns

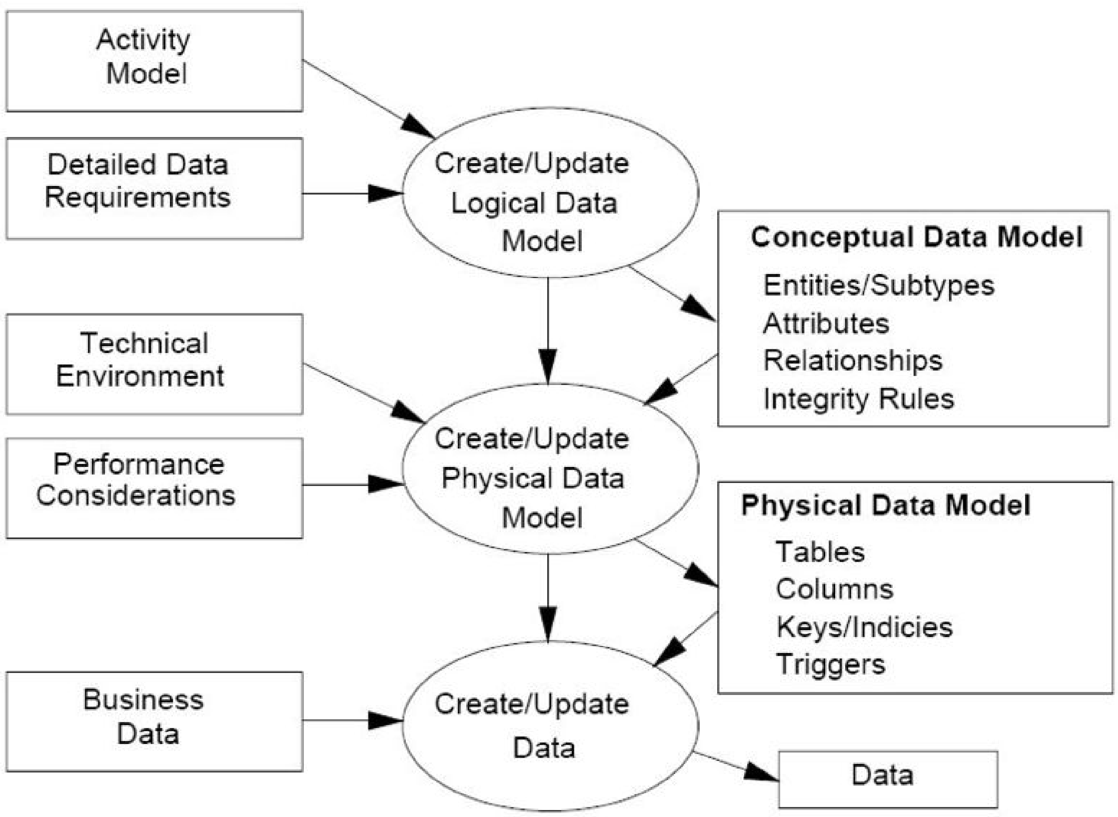
There are many common ways of designing computer software modules and their communications, among them:

- Blackboard
- Client–server model (2-tier, n-tier, peer-to-peer, cloud computing all use this model)
- Database-centric architecture (broad division can be made for programs which have database at its center and applications which don't have to rely on databases, E.g. desktop application programs, utility programs etc.)
- Distributed computing

- Event-driven architecture
- Front end and back end
- Implicit invocation
- Monolithic application
- Peer-to-peer
- Pipes and filters
- Plug-in (computing)
- Representational State Transfer
- Rule evaluation
- Search-oriented architecture (A pure SOA implements a service for every data access point.)
- Service-oriented architecture
- Shared nothing architecture
- Software componentry
- Space based architecture
- Structured (module-based but usually monolithic within modules)
- Three-tier model (An architecture with Presentation, Business Logic and Database tiers)

Chapter 3

Data Modeling



The data modeling process. The figure illustrates the way data models are developed and used today. A conceptual data model is developed based on the data requirements for the application that is being developed, perhaps in the context of an activity model. The data model will normally consist of entity types, attributes, relationships, integrity rules, and the definitions of those objects. This is then used as the start point for interface or database design.

Data modeling in software engineering is the process of creating a data model by applying formal data model descriptions using data modeling techniques.

Overview

Data modeling is a method used to define and analyze data requirements needed to support the business processes of an organization. The data requirements are recorded as a conceptual data model with associated data definitions. Actual implementation of the conceptual model is called a logical data model. To implement one conceptual data model may require multiple logical data models. Data modeling defines not just data elements, but their structures and relationships between them Data modeling techniques and methodologies are used to model data in a standard, consistent, predictable manner in order to manage it as a resource. The use of data modeling standards is strongly recommended for all projects requiring a standard means of defining and analyzing data within an organization, e.g., using data modeling:

- to manage data as a resource;
- for the integration of information systems;
- for designing databases/data warehouses (aka data repositories)

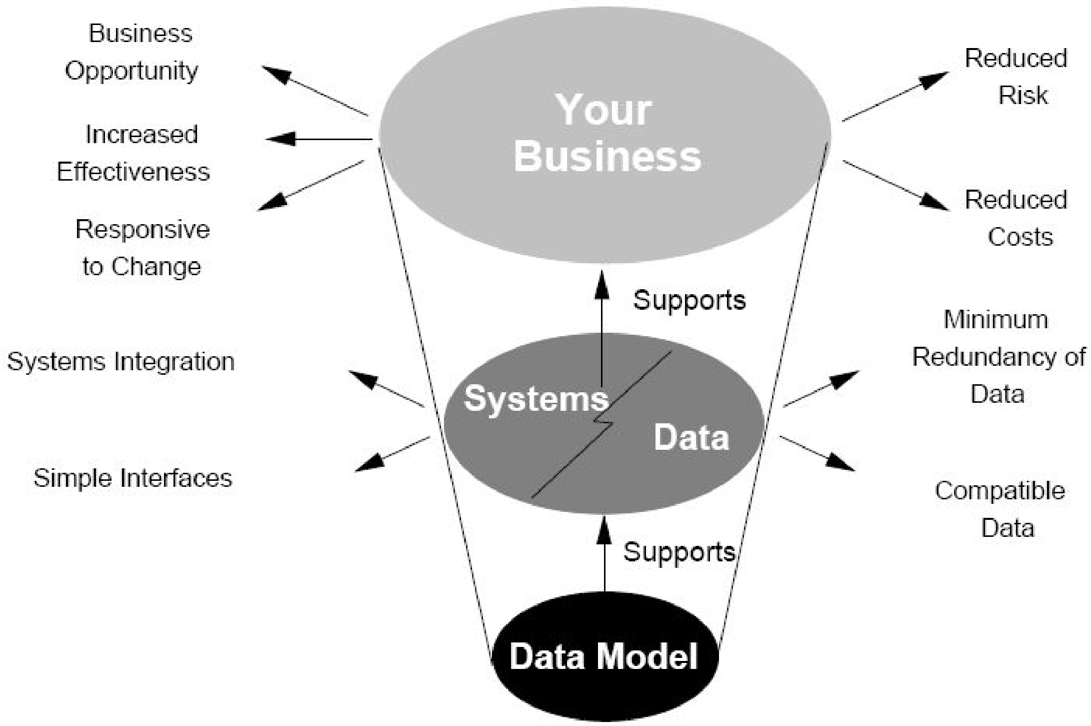
Data modeling may be performed during various types of projects and in multiple phases of projects. Data models are progressive; there is no such thing as the final data model for a business or application. Instead a data model should be considered a living document that will change in response to a changing business. The data models should ideally be stored in a repository so that they can be retrieved, expanded, and edited over time. Whitten (2004) determined two types of data modeling:

- Strategic data modeling: This is part of the creation of an information systems strategy, which defines an overall vision and architecture for information systems is defined. Information engineering is a methodology that embraces this approach.
- Data modeling during systems analysis: In systems analysis logical data models are created as part of the development of new databases.

Data modeling is also a technique for detailing business requirements for a database. It is sometimes called *database modeling* because a data model is eventually implemented in a database.

Data modeling topics

Data models



How data models deliver benefit.

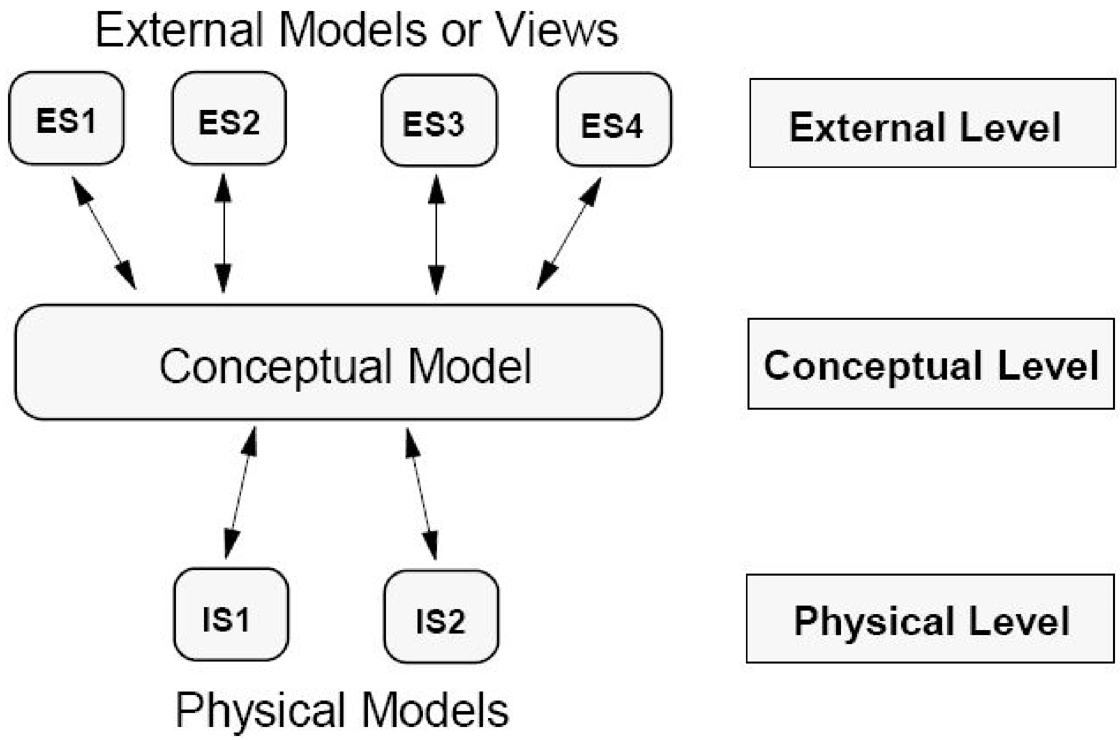
Data models support data and computer systems by providing the definition and format of data. If this is done consistently across systems then compatibility of data can be achieved. If the same data structures are used to store and access data then different applications can share data. The results of this are indicated above. However, systems and interfaces often cost more than they should, to build, operate, and maintain. They may also constrain the business rather than support it. A major cause is that the quality of the data models implemented in systems and interfaces is poor.

- Business rules, specific to how things are done in a particular place, are often fixed in the structure of a data model. This means that small changes in the way business is conducted lead to large changes in computer systems and interfaces.
- Entity types are often not identified, or incorrectly identified. This can lead to replication of data, data structure, and functionality, together with the attendant costs of that duplication in development and maintenance.
- Data models for different systems are arbitrarily different. The result of this is that complex interfaces are required between systems that share data. These interfaces can account for between 25-70% of the cost of current systems.
- Data cannot be shared electronically with customers and suppliers, because the structure and meaning of data has not been standardised. For example,

engineering design data and drawings for process plant are still sometimes exchanged on paper.

The reason for these problems is a lack of standards that will ensure that data models will both meet business needs and be consistent.

Conceptual, logical and physical schemes



The ANSI/SPARC three level architecture. This shows that a data model can be an external model (or view), a conceptual model, or a physical model. This is not the only way to look at data models, but it is a useful way, particularly when comparing models.

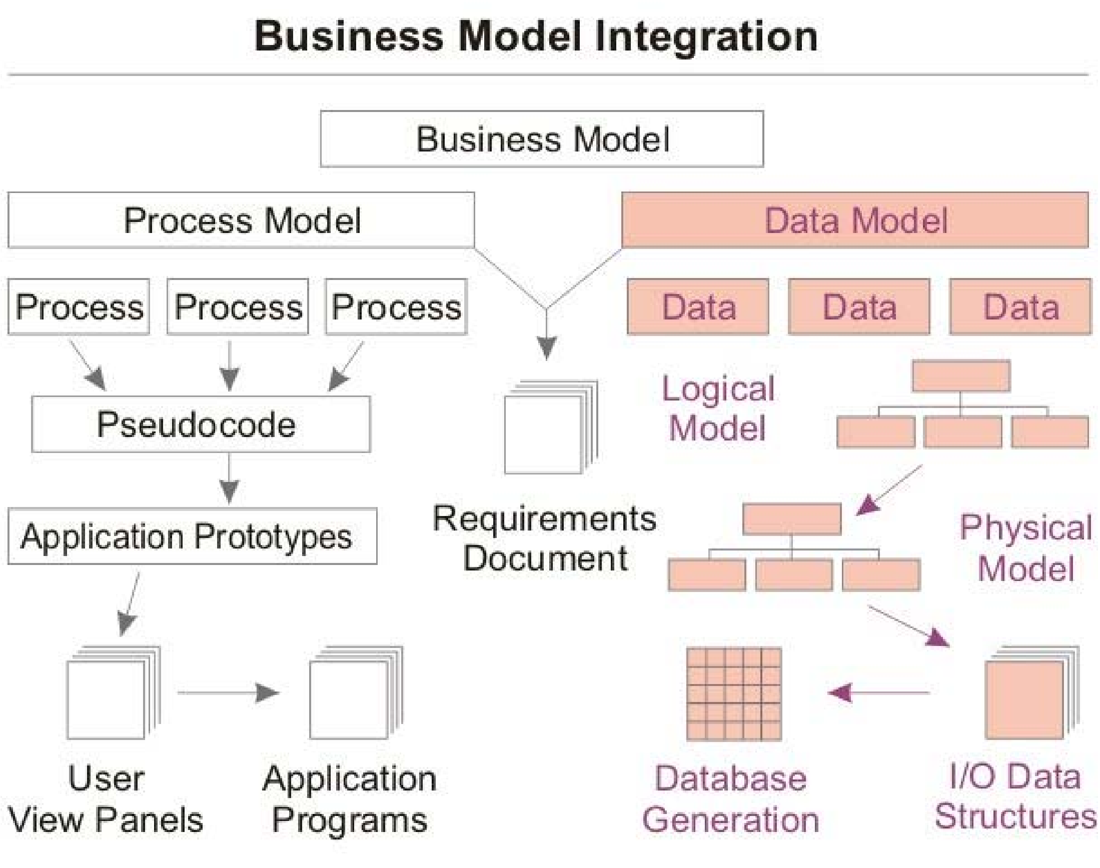
A data model *instance* may be one of three kinds according to ANSI in 1975:

- Conceptual schema: describes the semantics of a domain, being the scope of the model. For example, it may be a model of the interest area of an organization or industry. This consists of entity classes, representing kinds of things of significance in the domain, and relationships assertions about associations between pairs of entity classes. A conceptual schema specifies the kinds of facts or propositions that can be expressed using the model. In that sense, it defines the allowed expressions in an artificial 'language' with a scope that is limited by the scope of the model.
- Logical schema: describes the structure of some domain of information. This consists of descriptions of tables and columns, object oriented classes, and XML tags, among other things.

- Physical schema: describes the physical means by which data are stored. This is concerned with partitions, CPUs, tablespaces, and the like.

The significance of this approach, according to ANSI, is that it allows the three perspectives to be relatively independent of each other. Storage technology can change without affecting either the logical or the conceptual model. The table/column structure can change without (necessarily) affecting the conceptual model. In each case, of course, the structures must remain consistent with the other model. The table/column structure may be different from a direct translation of the entity classes and attributes, but it must ultimately carry out the objectives of the conceptual entity class structure. Early phases of many software development projects emphasize the design of a conceptual data model. Such a design can be detailed into a logical data model. In later stages, this model may be translated into physical data model. However, it is also possible to implement a conceptual model directly.

Data modeling process



Data modeling in the context of Business Process Integration.

In the context of Business Process Integration, see figure, data modeling will result in database generation. It complements business process modeling, which results in application programs to support the business processes.

The actual database design is the process of producing a detailed data model of a database. This logical data model contains all the needed logical and physical design choices and physical storage parameters needed to generate a design in a Data Definition Language, which can then be used to create a database. A fully attributed data model contains detailed attributes for each entity. The term database design can be used to describe many different parts of the design of an overall database system. Principally, and most correctly, it can be thought of as the logical design of the base data structures used to store the data. In the relational model these are the tables and views. In an Object database the entities and relationships map directly to object classes and named relationships. However, the term database design could also be used to apply to the overall process of designing, not just the base data structures, but also the forms and queries used as part of the overall database application within the Database Management System or DBMS.

In the process, system interfaces account for 25% to 70% of the development and support costs of current systems. The primary reason for this cost is that these systems do not share a common data model. If data models are developed on a system by system basis, then not only is the same analysis repeated in overlapping areas, but further analysis must be performed to create the interfaces between them. Most systems contain the same basic components, redeveloped for a specific purpose. For instance the following can use the same basic classification model as a component:

- Materials Catalogue,
- Product and Brand Specifications,
- Equipment specifications.

The same components are redeveloped because we have no way of telling they are the same thing.

Modeling methodologies

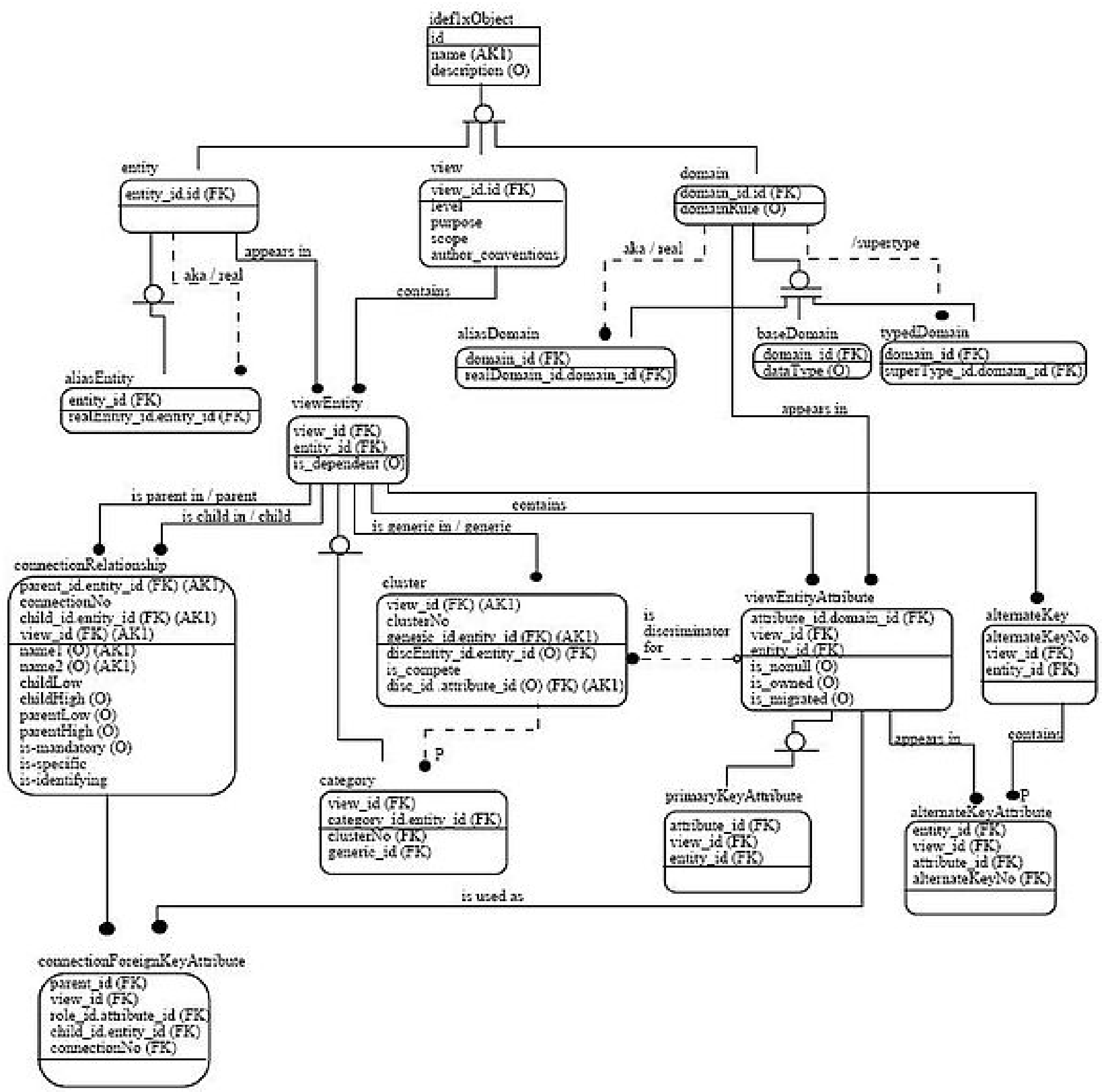
Data models represent information areas of interest. While there are many ways to create data models, according to Len Silverston (1997) only two modeling methodologies stand out, top-down and bottom-up:

- Bottom-up models are often the result of a reengineering effort. They usually start with existing data structures forms, fields on application screens, or reports. These models are usually physical, application-specific, and incomplete from an enterprise perspective. They may not promote data sharing, especially if they are built without reference to other parts of the organization.
- Top-down logical data models, on the other hand, are created in an abstract way by getting information from people who know the subject area. A system may not

implement all the entities in a logical model, but the model serves as a reference point or template.

Sometimes models are created in a mixture of the two methods: by considering the data needs and structure of an application and by consistently referencing a subject-area model. Unfortunately, in many environments the distinction between a logical data model and a physical data model is blurred. In addition, some CASE tools don't make a distinction between logical and physical data models.

Entity relationship diagrams



Example of a IDEF1X Entity relationship diagrams used to model IDEF1X itself. The name of the view is mm. The domain hierarchy and constraints are also given. The constraints are expressed as sentences in the formal theory of the meta model.

There are several notations for data modeling. The actual model is frequently called "Entity relationship model", because it depicts data in terms of the entities and relationships described in the data. An entity-relationship model (ERM) is an abstract conceptual representation of structured data. Entity-relationship modeling is a relational schema database modeling method, used in software engineering to produce a type of conceptual data model (or semantic data model) of a system, often a relational database, and its requirements in a top-down fashion.

These models are being used in the first stage of information system design during the requirements analysis to describe information needs or the type of information that is to be stored in a database. The data modeling technique can be used to describe any ontology (i.e. an overview and classifications of used terms and their relationships) for a certain universe of discourse i.e. area of interest.

Several techniques have been developed for the design of data models. While these methodologies guide data modelers in their work, two different people using the same methodology will often come up with very different results. Most notable are:

- Bachman diagrams
- Barker's Notation
- Chen's Notation
- Data Vault Modeling
- Extended Backus–Naur form
- IDEF1X
- Object-relational mapping
- Object Role Modeling
- Relational Model

Generic data modeling

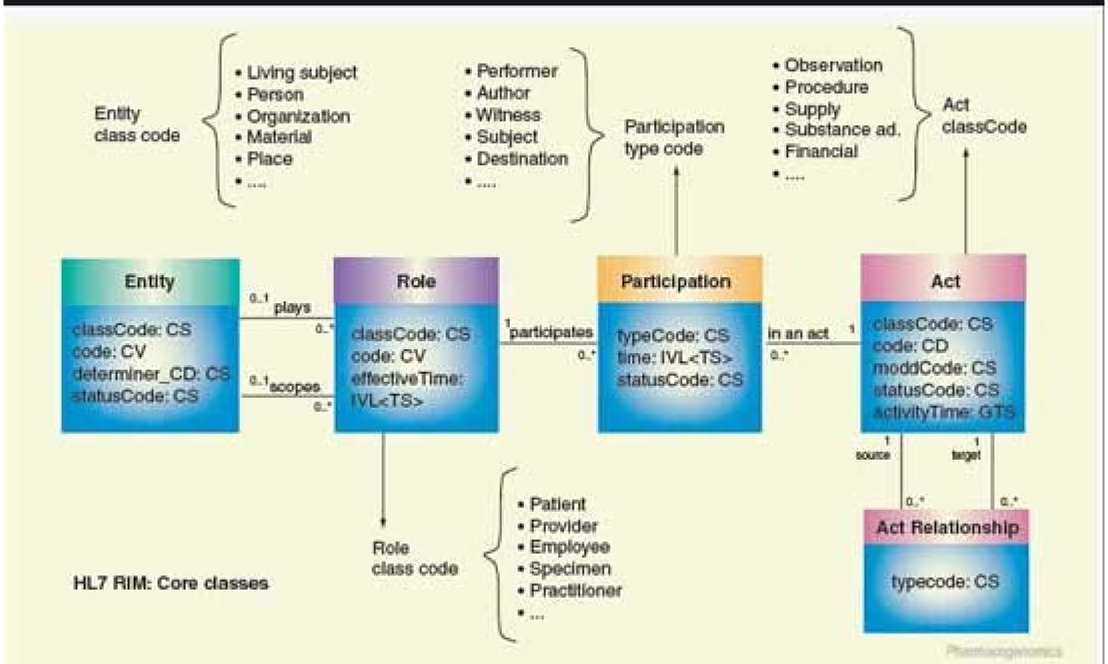


Figure 1. The backbone of the HL7 Reference Information Model.

An entity plays a role that participates in an action. The RIM backbone is a generic set of classes from which health specific classes are derived, for example, observation, procedure and substance administration are all subclasses of act.
CD: Concept descriptor; CS: Coded simple value; CV: Coded value; GTS: General Timing Specification; HL7: Health Level Seven; IVL: Interval; RIM: Reference Information Model; TS: Point in time.

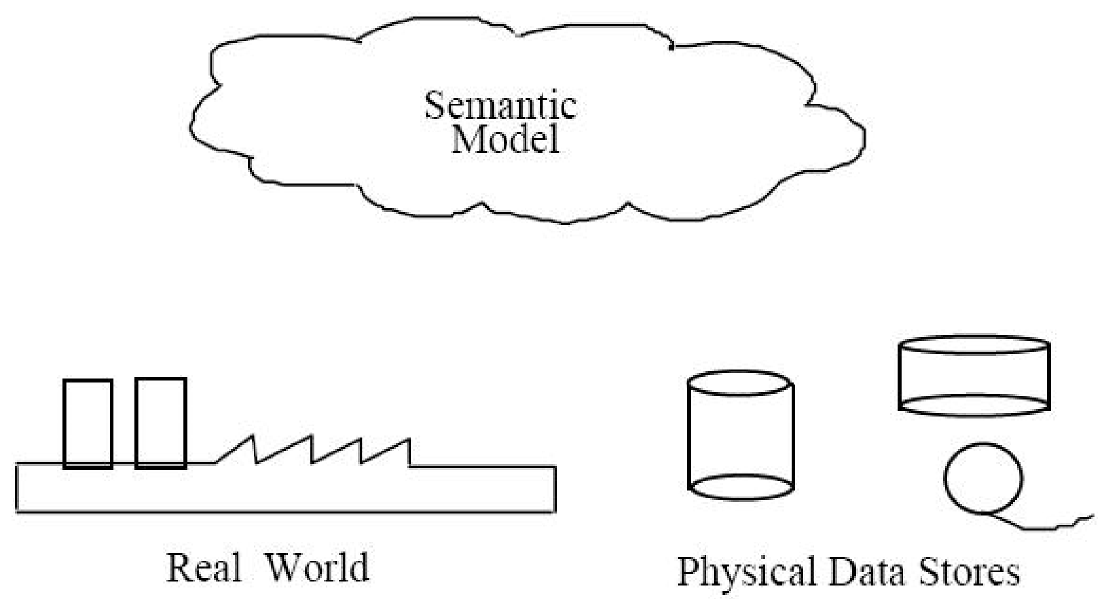
Example of a Generic data model.

Generic data models are generalizations of conventional data models. They define standardized general relation types, together with the kinds of things that may be related by such a relation type. The definition of generic data model is similar to the definition of a natural language. For example, a generic data model may define relation types such as a 'classification relation', being a binary relation between an individual thing and a kind of thing (a class) and a 'part-whole relation', being a binary relation between two things, one with the role of part, the other with the role of whole, regardless the kind of things that are related.

Given an extensible list of classes, this allows the classification of any individual thing and to specify part-whole relations for any individual object. By standardization of an extensible list of relation types, a generic data model enables the expression of an unlimited number of kinds of facts and will approach the capabilities of natural languages. Conventional data models, on the other hand, have a fixed and limited domain scope, because the instantiation (usage) of such a model only allows expressions of kinds of facts that are predefined in the model.

Semantic data modeling

The logical data structure of a DBMS, whether hierarchical, network, or relational, cannot totally satisfy the requirements for a conceptual definition of data because it is limited in scope and biased toward the implementation strategy employed by the DBMS.



Semantic data models.

Therefore, the need to define data from a conceptual view has led to the development of semantic data modeling techniques. That is, techniques to define the meaning of data within the context of its interrelationships with other data. As illustrated in the figure the real world, in terms of resources, ideas, events, etc., are symbolically defined within physical data stores. A semantic data model is an abstraction which defines how the stored symbols relate to the real world. Thus, the model must be a true representation of the real world.

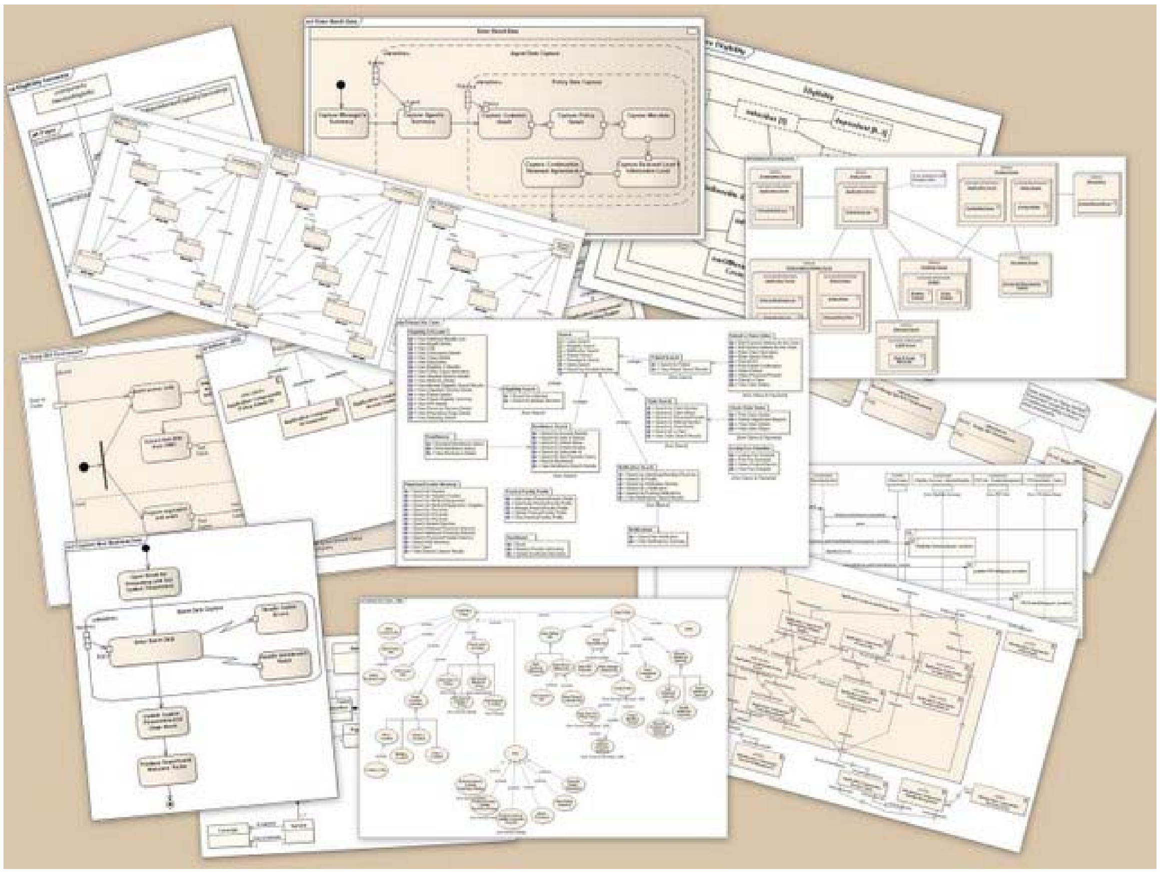
A semantic data model can be used to serve many purposes, such as:.

- planning of data resources
- building of shareable databases
- evaluation of vendor software
- integration of existing databases

The overall goal of semantic data models is to capture more meaning of data by integrating relational concepts with more powerful abstraction concepts known from the Artificial Intelligence field. The idea is to provide high level modeling primitives as integral part of a data model in order to facilitate the representation of real world situations.

Chapter 4

Unified Modeling Language



A collage of UML diagrams.

Unified Modeling Language (**UML**) is a standardized general-purpose modeling language in the field of object-oriented software engineering. The standard is managed, and was created by, the Object Management Group.

UML includes a set of graphic notation techniques to create visual models of object-oriented software-intensive systems.

Overview

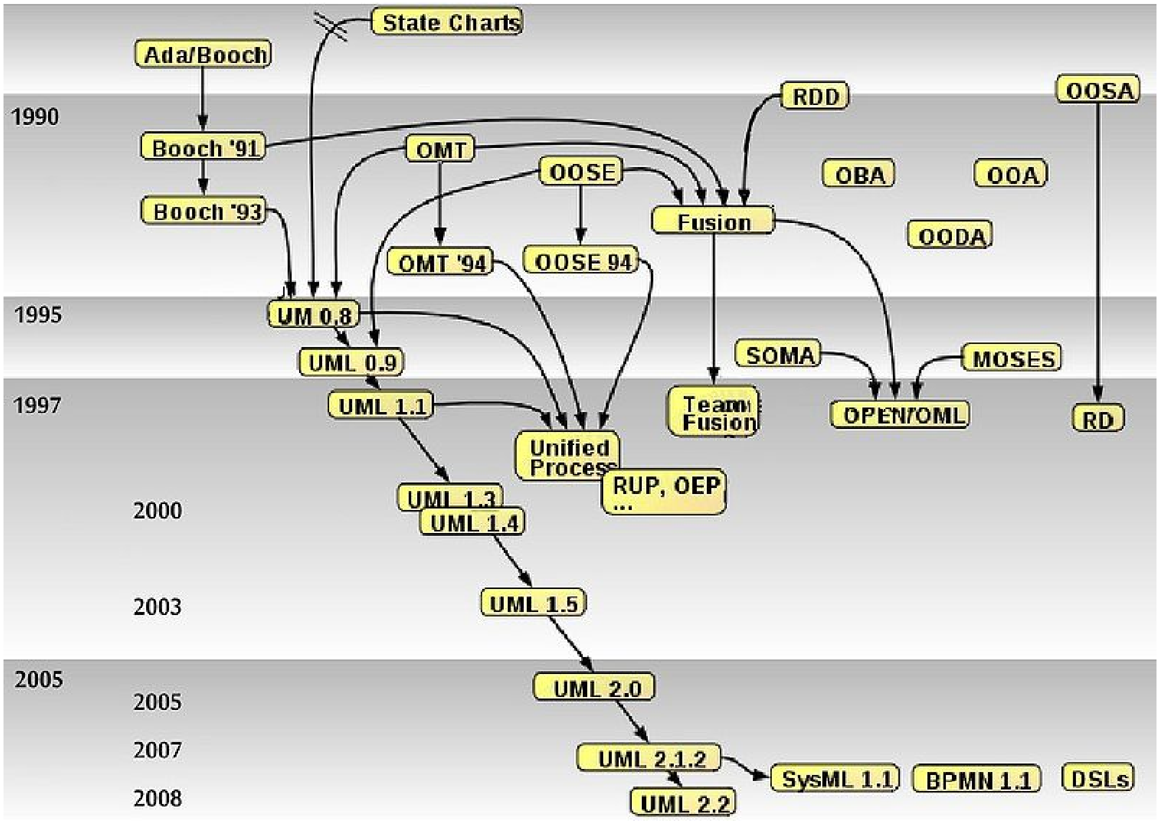
The Unified Modeling Language (UML) is used to specify, visualize, modify, construct and document the artifacts of an object-oriented software-intensive system under development. UML offers a standard way to visualize a system's architectural blueprints, including elements such as:

- activities
- actors
- business processes
- database schemas
- (logical) components
- programming language statements
- reusable software components.

UML combines techniques from data modeling (entity relationship diagrams), business modeling (work flows), object modeling, and component modeling. It can be used with all processes, throughout the software development life cycle, and across different implementation technologies. UML has synthesized the notations of the Booch method, the Object-modeling technique (OMT) and Object-oriented software engineering (OOSE) by fusing them into a single, common and widely usable modeling language. UML aims to be a standard modeling language which can model concurrent and distributed systems. UML is a de facto industry standard, and is evolving under the auspices of the Object Management Group (OMG).

UML models may be automatically transformed to other representations (e.g. Java) by means of QVT-like transformation languages. UML is extensible, with two mechanisms for customization: profiles and stereotypes.

History



History of object-oriented methods and notation.

Before UML 1.x

After Rational Software Corporation hired James Rumbaugh from General Electric in 1994, the company became the source for the two most popular object-oriented modeling approaches of the day: Rumbaugh's Object-modeling technique (OMT), which was better for object-oriented analysis (OOA), and Grady Booch's Booch method, which was better for object-oriented design (OOD). They were soon assisted in their efforts by Ivar Jacobson, the creator of the object-oriented software engineering (OOSE) method. Jacobson joined Rational in 1995, after his company, Objectory AB, was acquired by Rational. The three methodologists were collectively referred to as the *Three Amigos*.

In 1996 Rational concluded that the abundance of modeling languages was slowing the adoption of object technology, so repositioning the work on an unified method, they tasked the Three Amigos with the development of a non-proprietary Unified Modeling Language. Representatives of competing object technology companies were consulted during OOPSLA '96; they chose *boxes* for representing classes rather than the *cloud* symbols that were used in Booch's notation.

Under the technical leadership of the Three Amigos, an international consortium called the UML Partners was organized in 1996 to complete the *Unified Modeling Language*

(UML) specification, and propose it as a response to the OMG RFP. The UML Partners' UML 1.0 specification draft was proposed to the OMG in January 1997. During the same month the UML Partners formed a Semantics Task Force, chaired by Cris Kobryn and administered by Ed Eykholt, to finalize the semantics of the specification and integrate it with other standardization efforts. The result of this work, UML 1.1, was submitted to the OMG in August 1997 and adopted by the OMG in November 1997.

UML 1.x

As a modeling notation, the influence of the OMT notation dominates (e. g., using rectangles for classes and objects). Though the Booch "cloud" notation was dropped, the Booch capability to specify lower-level design detail was embraced. The use case notation from Objectory and the component notation from Booch were integrated with the rest of the notation, but the semantic integration was relatively weak in UML 1.1, and was not really fixed until the UML 2.0 major revision.

Concepts from many other OO methods were also loosely integrated with UML with the intent that UML would support all OO methods. Many others also contributed, with their approaches flavouring the many models of the day, including: Tony Wasserman and Peter Pircher with the "Object-Oriented Structured Design (OOSD)" notation (not a method), Ray Buhr's "Systems Design with Ada", Archie Bowen's use case and timing analysis, Paul Ward's data analysis and David Harel's "Statecharts"; as the group tried to ensure broad coverage in the real-time systems domain. As a result, UML is useful in a variety of engineering problems, from single process, single user applications to concurrent, distributed systems, making UML rich but also large.

The Unified Modeling Language is an international standard:

ISO/IEC 19501:2005 Information technology – Open Distributed Processing – Unified Modeling Language (UML) Version 1.4.2

UML 2.x

UML has matured significantly since UML 1.1. Several minor revisions (UML 1.3, 1.4, and 1.5) fixed shortcomings and bugs with the first version of UML, followed by the UML 2.0 major revision that was adopted by the OMG in 2005.

Although UML 2.1 was never released as a formal specification, versions 2.1.1 and 2.1.2 appeared in 2007, followed by UML 2.2 in February 2009. UML 2.3 was formally released in May 2010.

There are four parts to the UML 2.x specification:

1. The Superstructure that defines the notation and semantics for diagrams and their model elements

2. The Infrastructure that defines the core metamodel on which the Superstructure is based
3. The Object Constraint Language (OCL) for defining rules for model elements
4. The UML Diagram Interchange that defines how UML 2 diagram layouts are exchanged

The current versions of these standards follow: UML Superstructure version 2.3, UML Infrastructure version 2.3, OCL version 2.2, and UML Diagram Interchange version 1.0.

Although many UML tools support some of the new features of UML 2.x, the OMG provides no test suite to objectively test compliance with its specifications.

Topics

Software development methods

UML is not a development method by itself; however, it was designed to be compatible with the leading object-oriented software development methods of its time (for example OMT, Booch method, Objectory). Since UML has evolved, some of these methods have been recast to take advantage of the new notations (for example OMT), and new methods have been created based on UML, such as IBM Rational Unified Process (RUP). Others include Abstraction Method and Dynamic Systems Development Method.

Modeling

It is important to distinguish between the UML model and the set of diagrams of a system. A diagram is a partial graphic representation of a system's model. The model also contains documentation that drive the model elements and diagrams (such as written use cases).

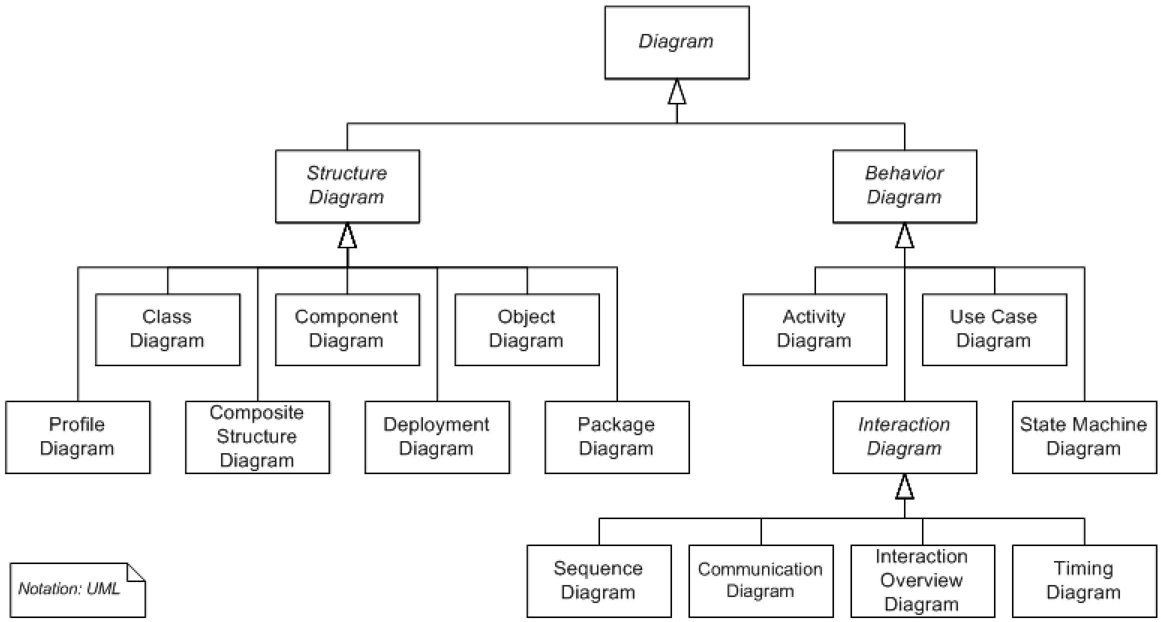
UML diagrams represent two different views of a system model:

- Static (or *structural*) view: emphasizes the static structure of the system using objects, attributes, operations and relationships. The structural view includes class diagrams and composite structure diagrams.
- Dynamic (or *behavioral*) view: emphasizes the dynamic behavior of the system by showing collaborations among objects and changes to the internal states of objects. This view includes sequence diagrams, activity diagrams and state machine diagrams.

UML models can be exchanged among UML tools by using the XMI interchange format.

Diagrams overview

UML 2.2 has 14 types of diagrams divided into two categories. Seven diagram types represent *structural* information, and the other seven represent general types of *behavior*, including four that represent different aspects of *interactions*. These diagrams can be categorized hierarchically as shown in the following class diagram:



UML does not restrict UML element types to a certain diagram type. In general, every UML element may appear on almost all types of diagrams; this flexibility has been partially restricted in UML 2.0. UML profiles may define additional diagram types or extend existing diagrams with additional notations.

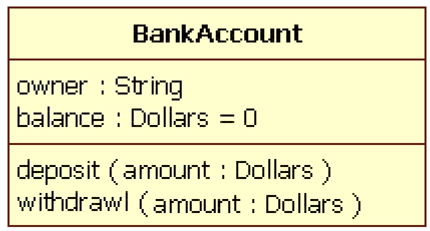
In keeping with the tradition of engineering drawings, a comment or note explaining usage, constraint, or intent is allowed in a UML diagram.

Structure diagrams

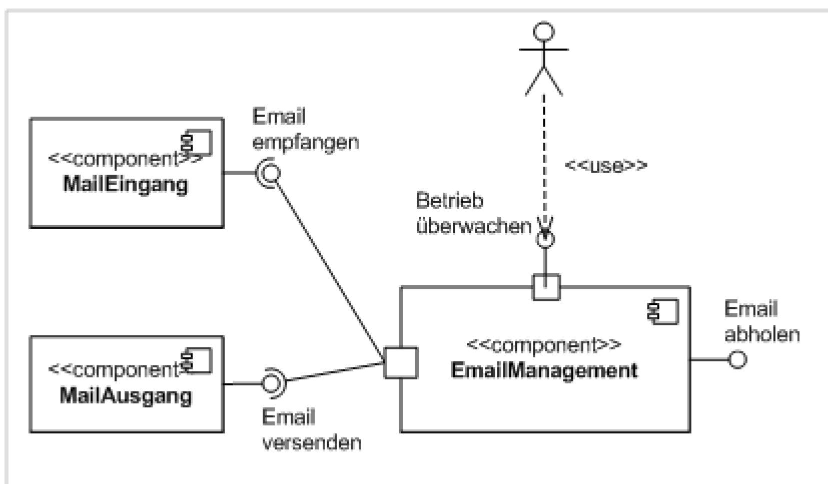
Structure diagrams emphasize the things that must be present in the system being modeled. Since structure diagrams represent the structure, they are used extensively in documenting the software architecture of software systems.

- Class diagram: describes the structure of a system by showing the system's classes, their attributes, and the relationships among the classes.
- Component diagram: describes how a software system is split up into components and shows the dependencies among these components.
- Composite structure diagram: describes the internal structure of a class and the collaborations that this structure makes possible.
- Deployment diagram: describes the hardware used in system implementations and the execution environments and artifacts deployed on the hardware.
- Object diagram: shows a complete or partial view of the structure of a modeled system at a specific time.
- Package diagram: describes how a system is split up into logical groupings by showing the dependencies among these groupings.

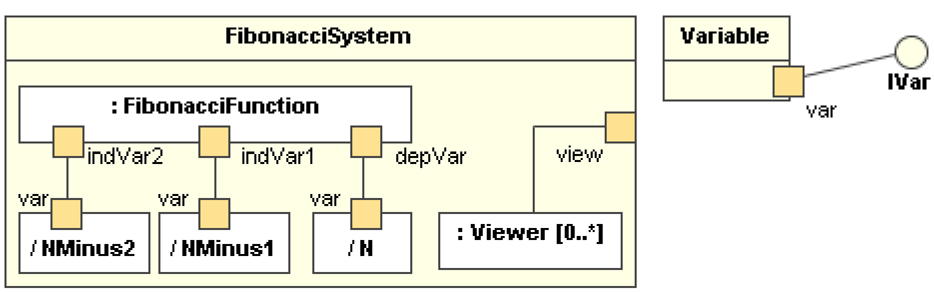
- Profile diagram: operates at the metamodel level to show stereotypes as classes with the <<stereotype>> stereotype, and profiles as packages with the <<profile>> stereotype. The extension relation (solid line with closed, filled arrowhead) indicates what metamodel element a given stereotype is extending.



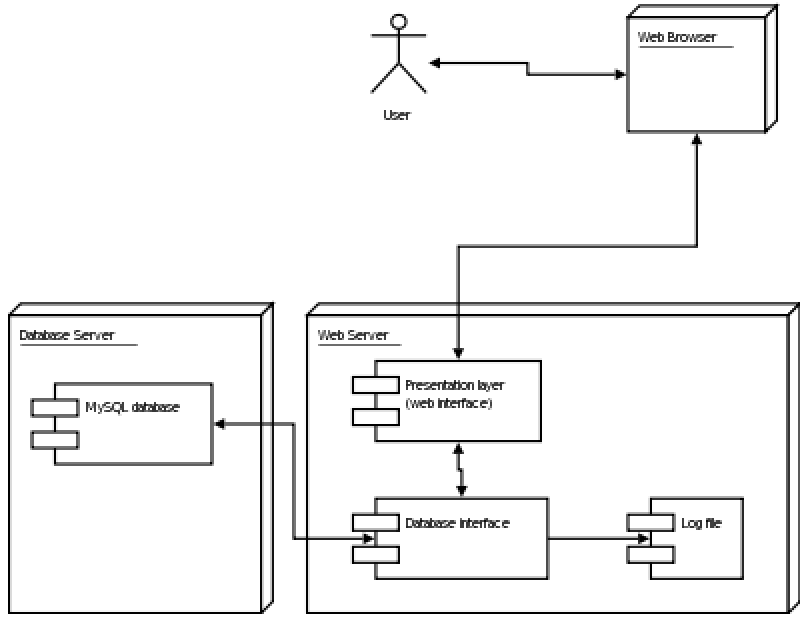
Class diagram



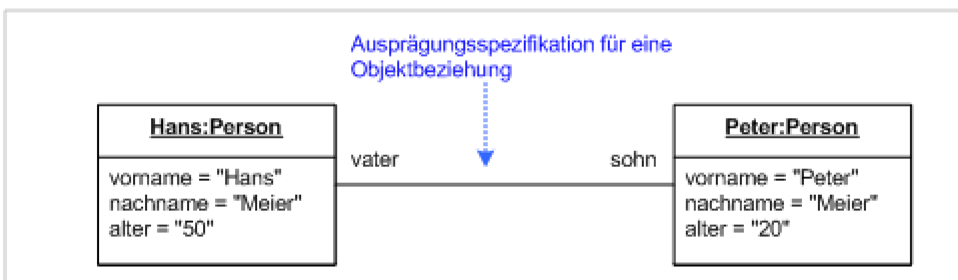
Component diagram



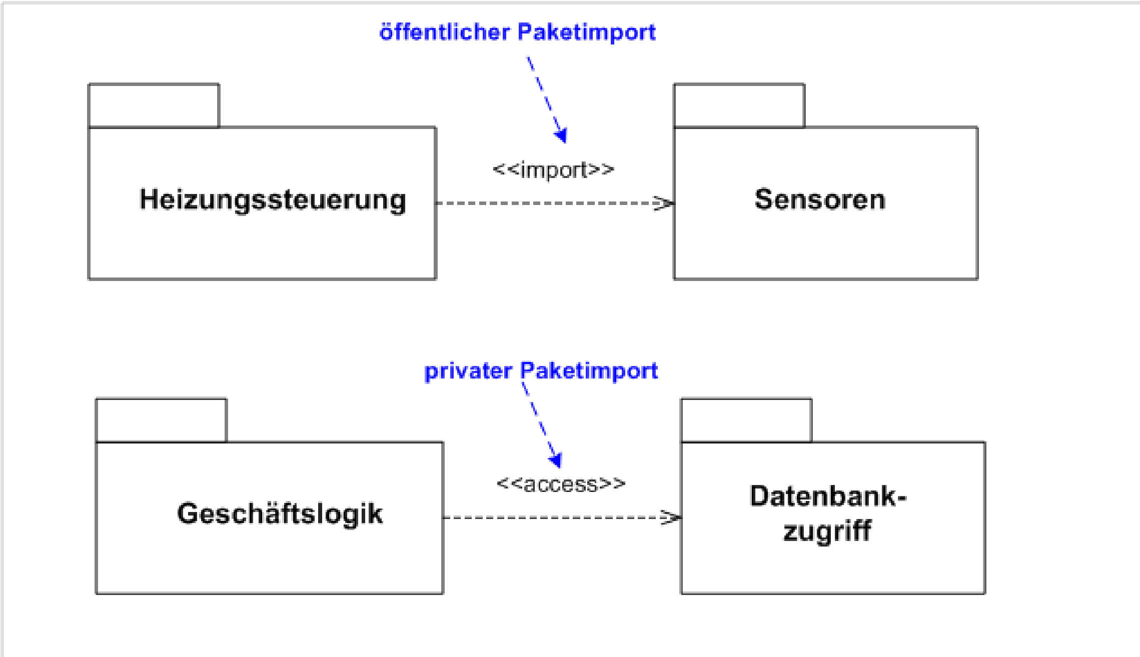
Composite structure diagrams



Deployment diagram



Object diagram

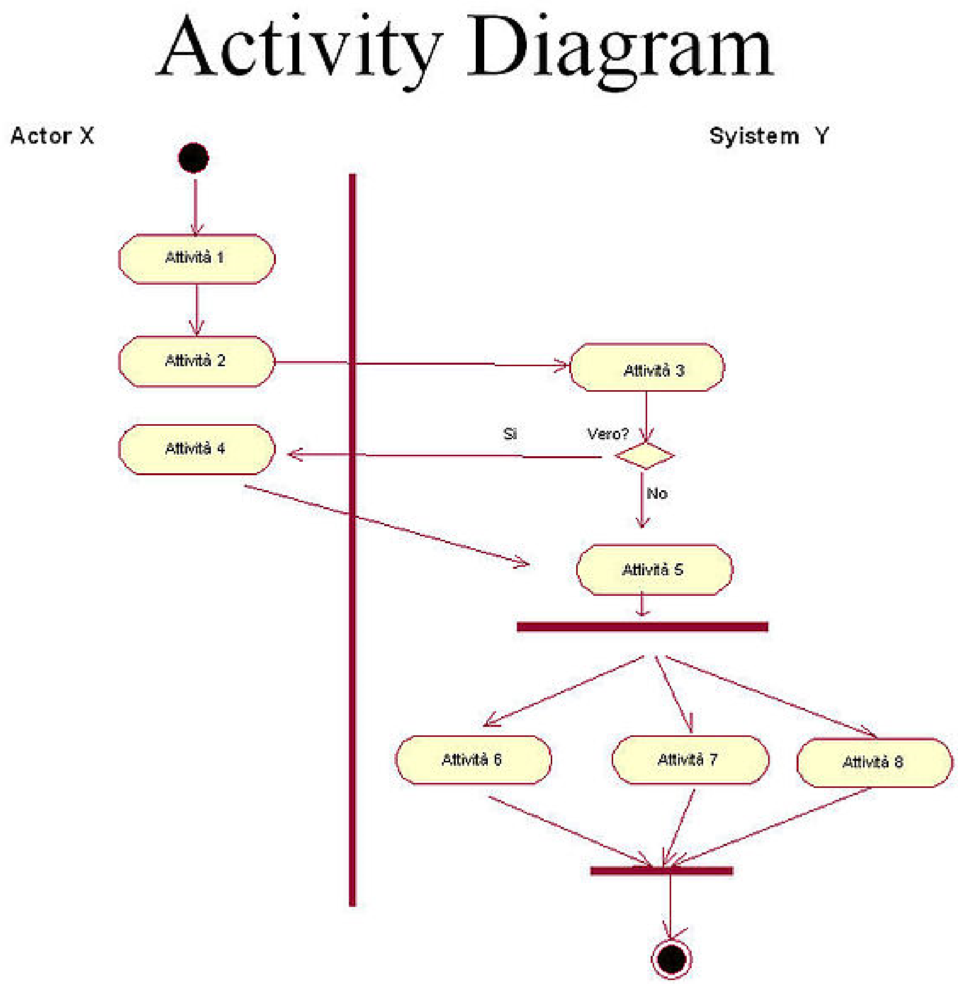


Package diagram

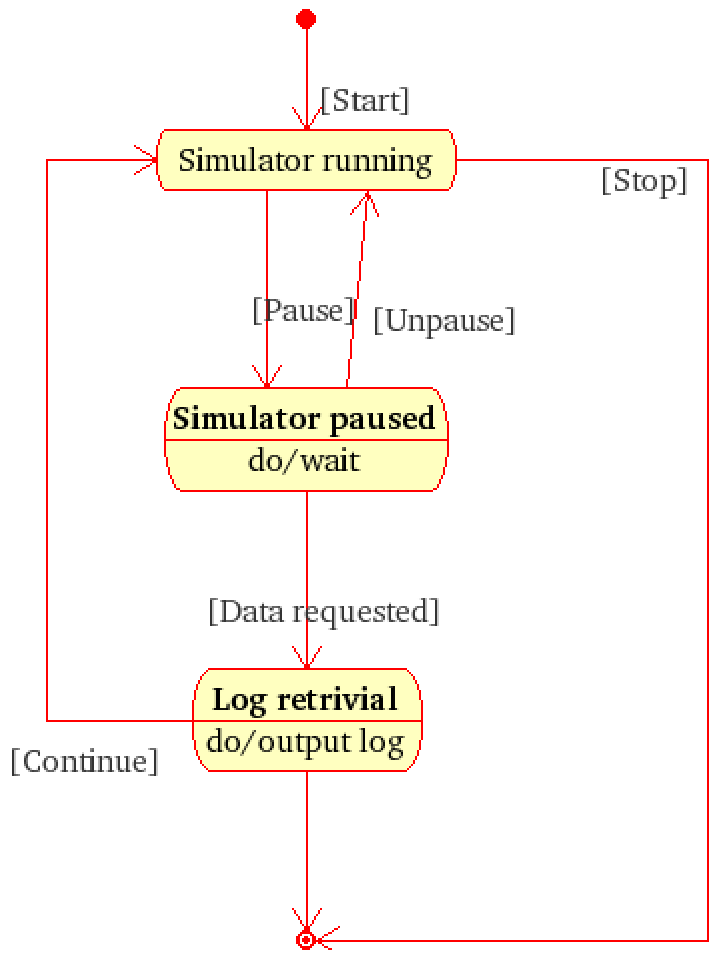
Behaviour diagrams

Behavior diagrams emphasize what must happen in the system being modeled. Since behavior diagrams illustrate the behavior of a system, they are used extensively to describe the functionality of software systems.

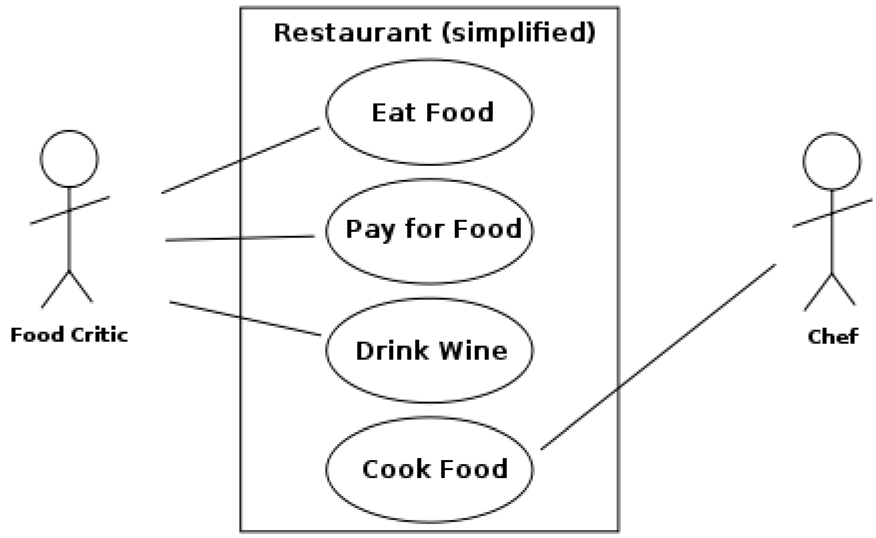
- Activity diagram: describes the business and operational step-by-step workflows of components in a system. An activity diagram shows the overall flow of control.
- UML state machine diagram: describes the states and state transitions of the system.
- Use case diagram: describes the functionality provided by a system in terms of actors, their goals represented as use cases, and any dependencies among those use cases.



UML Activity Diagram



State Machine diagram

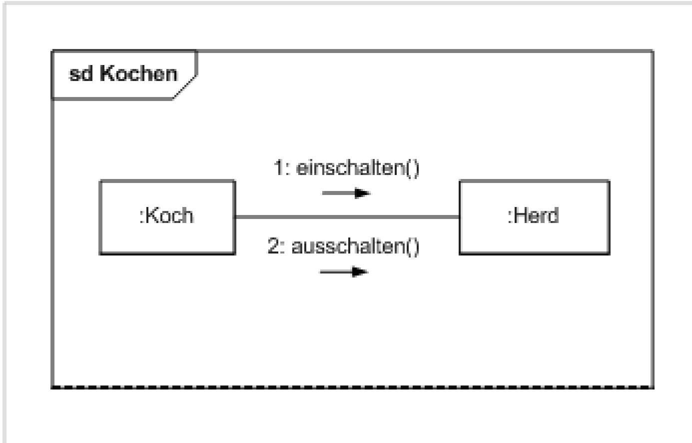


Use case diagram

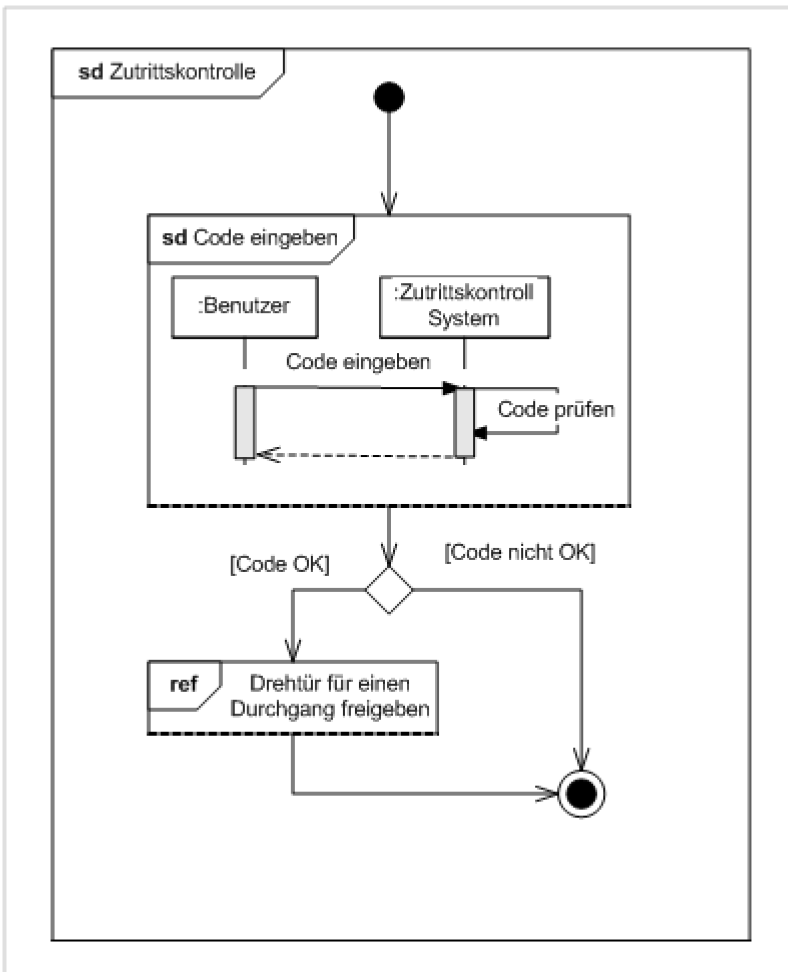
Interaction diagrams

Interaction diagrams, a subset of behaviour diagrams, emphasize the flow of control and data among the things in the system being modeled:

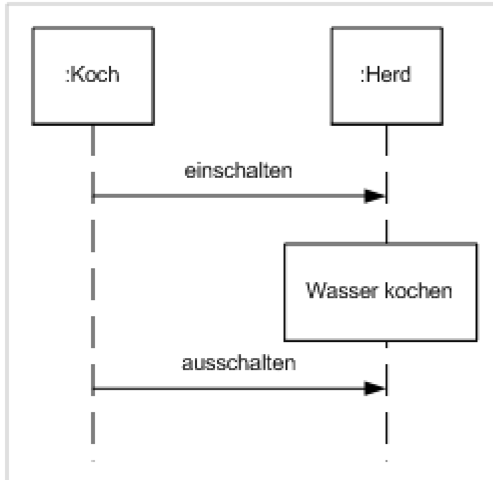
- Communication diagram: shows the interactions between objects or parts in terms of sequenced messages. They represent a combination of information taken from Class, Sequence, and Use Case Diagrams describing both the static structure and dynamic behavior of a system.
- Interaction overview diagram: provides an overview in which the nodes represent communication diagrams.
- Sequence diagram: shows how objects communicate with each other in terms of a sequence of messages. Also indicates the lifespans of objects relative to those messages.
- Timing diagrams: a specific type of interaction diagram where the focus is on timing constraints.



Communication diagram



Interaction overview diagram



Sequence diagram

The Protocol State Machine is a sub-variant of the State Machine. It may be used to model network communication protocols.

Meta modeling

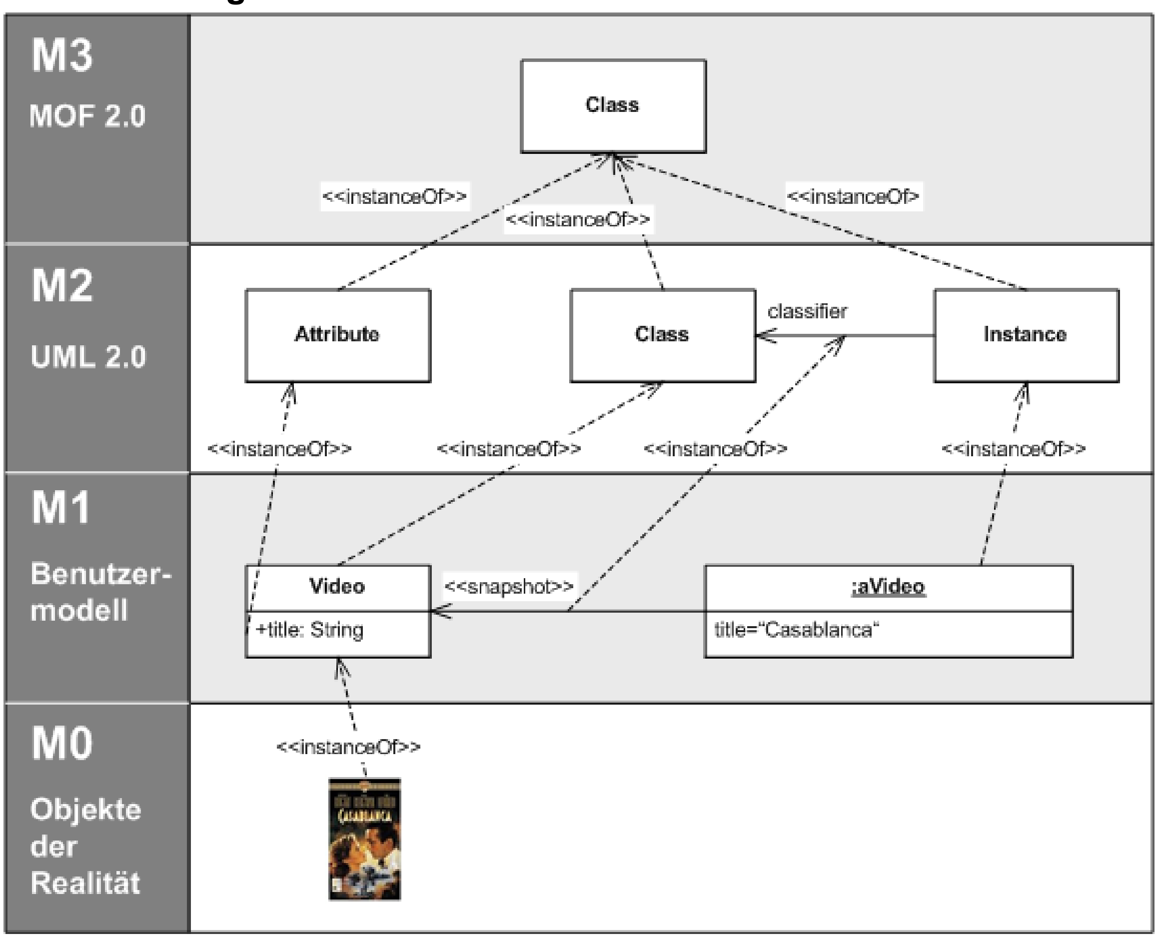


Illustration of the Meta-Object Facility.

The Object Management Group (OMG) has developed a metamodeling architecture to define the Unified Modeling Language (UML), called the Meta-Object Facility (MOF). The Meta-Object Facility is a standard for model-driven engineering, designed as a four-layered architecture, as shown in the image at right. It provides a meta-meta model at the top layer, called the M0 layer. This M0-model is the language used by Meta-Object Facility to build metamodels, called M1-models. The most prominent example of a Layer 1 Meta-Object Facility model is the UML metamodel, the model that describes the UML itself. These M1-models describe elements of the M2-layer, and thus M2-models. These would be, for example, models written in UML. The last layer is the M3-layer or data layer. It is used to describe runtime instance of the system.

Beyond the M0-model, the Meta-Object Facility describes the means to create and manipulate models and metamodels by defining CORBA interfaces that describe those operations. Because of the similarities between the Meta-Object Facility M0-model and UML structure models, Meta-Object Facility metamodels are usually modeled as UML class diagrams. A supporting standard of the Meta-Object Facility is XMI, which defines an XML-based exchange format for models on the M0-, M1-, or M2-Layer.

Criticisms

Although UML is a widely recognized and used modeling standard, it is frequently criticized for the following:

Standards bloat

> Bertrand Meyer, in a satirical essay framed as a student's request for a grade change, apparently criticized UML as of 1997 for being unrelated to object-oriented software development; a disclaimer was added later pointing out that his company nevertheless supports UML. Ivar Jacobson, a co-architect of UML, said that objections to UML 2.0's size were valid enough to consider the application of intelligent agents to the problem. It contains many diagrams and constructs that are redundant or infrequently used.

Problems in learning and adopting

> The problems cited in this section make learning and adopting UML problematic, especially when required of engineers lacking the prerequisite skills. In practice, people often draw diagrams with the symbols provided by their CASE tool, but without the meanings those symbols are intended to provide.

Linguistic incoherence

> The extremely poor writing of the UML standards themselves—assumed to be the consequence of having been written by a non-native English speaker—seriously reduces their normative value. In this respect the standards have been widely cited, and indeed pilloried, as prime examples of unintelligible geekspeak.

Capabilities of UML and implementation language mismatch

> As with any notational system, UML is able to represent some systems more concisely or efficiently than others. Thus a developer gravitates toward solutions that reside at the intersection of the capabilities of UML and the implementation language. This problem is particularly pronounced if the implementation language

does not adhere to orthodox object-oriented doctrine, as the intersection set between UML and implementation language may be that much smaller.

Dysfunctional interchange format

While the XMI (XML Metadata Interchange) standard is designed to facilitate the interchange of UML models, it has been largely ineffective in the practical interchange of UML 2.x models. This interoperability ineffectiveness is attributable to two reasons. Firstly, XMI 2.x is large and complex in its own right, since it purports to address a technical problem more ambitious than exchanging UML 2.x models. In particular, it attempts to provide a mechanism for facilitating the exchange of any arbitrary modeling language defined by the OMG's Meta-Object Facility (MOF). Secondly, the UML 2.x Diagram Interchange specification lacks sufficient detail to facilitate reliable interchange of UML 2.x notations between modeling tools. Since UML is a visual modeling language, this shortcoming is substantial for modelers who don't want to redraw their diagrams.

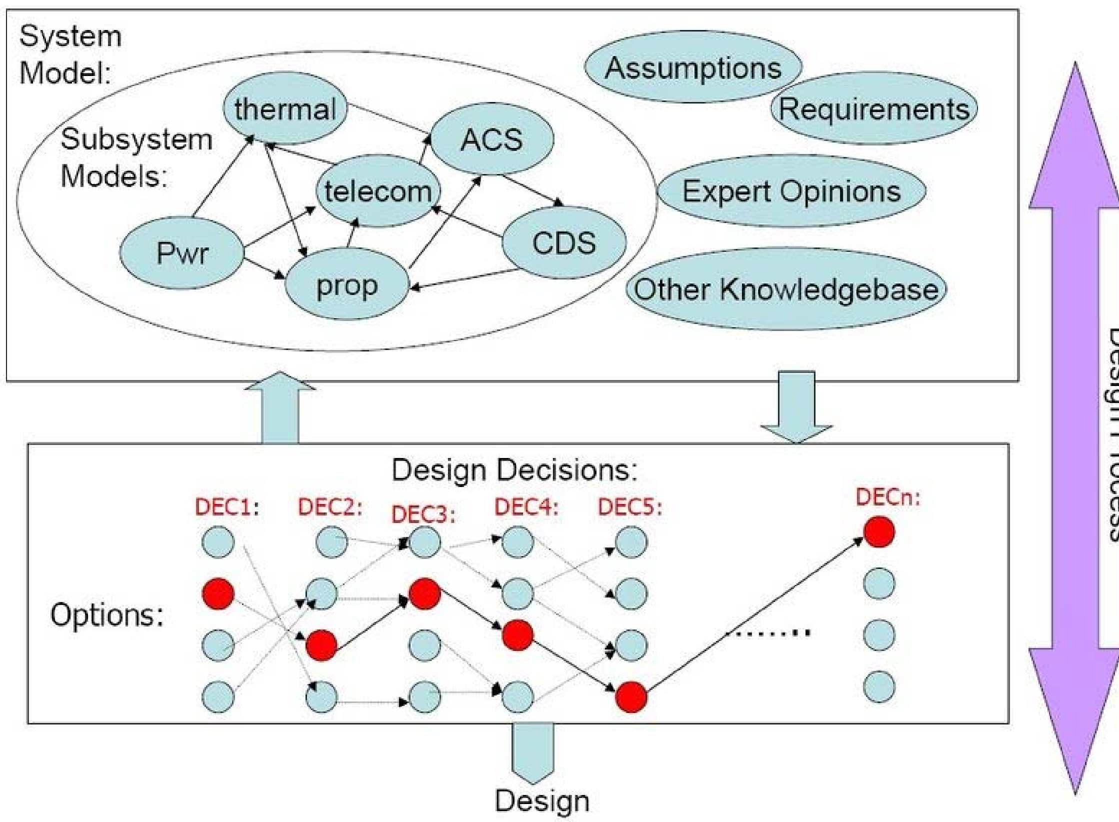
Modeling experts have written sharp criticisms of UML, including Bertrand Meyer's "UML: The Positive Spin", and Brian Henderson-Sellers and Cesar Gonzalez-Perez in "Uses and Abuses of the Stereotype Mechanism in UML 1.x and 2.0".

UML modelling tools

The most well-known UML modelling tool is IBM Rational Rose. Other tools include Rational Rhapsody, MagicDraw UML, StarUML, ArgoUML, Umbrello, BOUML, PowerDesigner, and Dia. Some of popular development environments also offer UML modelling tools, e.g.: Eclipse, NetBeans, and Visual Studio.

Chapter 5

Design Rationale



A Decision Based Design Structure, which spans the areas of Engineering Design, Design Rationale and Decision Analysis.

A **Design Rationale** is an explicit documentation of the reasons behind decisions made when designing a system or artifact. As initially developed by W.R. Kunz and Horst Rittel, design rationale seeks to provide argumentation-based structure to the political, collaborative process of addressing wicked problems.

Overview

A design rationale is the explicit listing of decisions made during a design process, and the reasons why those decisions were made. Its primary goal is to support designers by providing a means to record and communicate the argumentation and reasoning behind the design process. It should therefore include:

- the reasons behind a design decision,
- the justification for it,
- the other alternatives considered,
- the trade offs evaluated, and
- the argumentation that led to the decision.

Several science areas are involved in the study of design rationales, such as Computer Science Cognitive Science, Artificial Intelligence, and Knowledge Management To supporting design rationale, a lot of frameworks proposed, such as QOC, DRCS, IBIS, and DRL.

History

While argumentation formats can be traced back to Stephen Toulmin's work in the 1950s datums, claims, warrants, backings and rebuttals, the origin of design rationale can be traced back to W.R. Kunz and Horst Rittel's development of the Issue-Based Information System (IBIS) notation in 1970. Several variants on IBIS have since been proposed.

- The first was Procedural Hierarchy of Issues (PHI), first described in Ray McCall's PhD Dissertation although not named at the time.

- IBIS was also modified, in this case to support Software Engineering, by Potts & Bruns. The Potts & Bruns approach was then extended by the Decision Representation Language (DRL). which itself was extended by RATSpeak.

- Questions Options and Criteria (QOC), also known as Design Space Analysis is an alternative representation for argumentation-based rationale, as are Win-Win and the Decision Recommendation and Intent Model (DRIM).

The first Rationale Management System (RMS) was PROTOCOL, which supported PHI, which was followed by other PHI-based systems MIKROPOLIS and PHIDIAS. The first system providing IBIS support was Hans Dehlinger's STIEC. Rittel developed a small system in 1983 (also not published) and the better known gIBIS (graphical IBIS) was developed in 1987.

Not all successful DR approaches involve structured argumentation. Jack Carroll's Scenario-Claims Analysis approach captures rationale in scenarios that describe how the system is used and how well the system features support the user goals. Carroll and Rossen's approach to design rationale is intended to help designers of computer software and hardware identify underlying design tradeoffs and make inferences about the impact of potential design interventions.

Key Concepts in Design Rationale

There are a number of ways to characterize DR approaches. Some key distinguishing features are how it is captured, how it is represented, and how it can be used.

Rationale Capture

Rationale Capture is the process of acquiring rationale information to a rationale management system.

Capture Methods

- A method called “Reconstruction” captures rationales in a raw form such as video, and then reconstruct them into a more structured form. The advantage of Reconstruction method is that rationales can be carefully captured and capturing process won’t disrupt the designer. But this method might result in high cost and biases of the person producing the rationales
- The “Record-and-replay” method simply captures rationales as they unfold. Rationales are synchronously captured in a video conference or asynchronously captured via bulletin board or email-based discussion. If the system has informal and semi-formal representation, the method will be helpful.
- The “Methodological byproduct” method captures rationales during the process of design following a schema. But it’s hard to design such a schema. The advantage of this method is its low cost.
- With a rich knowledge base(KB) created in advance, the “Apprentice” method captures rationales by asking questions when confusing or disagreeing with the designer’s action. This method benefits not only the user but the system.
- In “Automatic Generation” method, design rationales are automatically generated from an execution history at low cost. It has the ability in maintaining consistent and up-to-date rationales. But the cost of compiling the execution history is high due to the complexity and difficulty of some machine-learning problems.
- The “Historian” method let a person or computer program watches all designer's actions but does not make suggestions. Rationales are captured during the design process.

Rationale Representation

The choice of Design Rationale representation is very important to make sure that the rationales we capture is what we desire and we can use efficiently. According to the degree of formality, the approaches that are used to represent design rationale can be divided into three main categories: informal, semiformal, or formal. In the informal representation, rationales can be recorded and captured by just using our traditionally accepted methods and media, such as word processors, audio and video records or even hand writings. However, these descriptions make it hard for automatic interpretation or other computer-based supports. In the formal representation, the rationale must be collected under a strict format so that the rationale can be interpreted and understood by computers. However, due to the strict format of rationale defined by formal representations, the contents can hardly be understood by human being and the process of capturing design rationale will require more efforts to finish, and therefore becomes more intrusive.

Semiformal representations try to combine the advantages of informal and formal representations. On one hand, the information captured should be able to be processed by computers so that more computer based support can be provided. On the other hand, the procedure and method used to capture information of design rationale should not be very intrusive. In the system with a semiformal representation, the information expected is suggested and the users can capture rationale by following the instructions to either fill out the attributes according to some templates or just type into natural language descriptions.

Argumentation-based models

The Toulmin model

One commonly accepted way for semiformal Design Rationale representation is structuring Design Rationale as argumentation. The earliest argumentation-based model used by many design rationale systems is the Toulmin model. The Toulmin model defines the rules of design rationale argumentation with six steps:

1. Claim is made;
2. Supporting data are provided;
3. Warrant provides evidence to the existing relations;
4. Warrant can be supported by a backing;
5. Model qualifiers (some, many, most, etc.) are provided;
6. Possible rebuttals are also considered.

One advantage of Toulmin model is that it uses words and concepts which can be easily understood by most people.

Issue-Based Information System (IBIS)

Another important approach to argumentation of Design Rationale is Rittel and Kunz's IBIS (Issue-Based Information System), which is actually not a software system but an argumentative notation. It is used and developed by gIBIS (graphical IBIS) and itIBIS (test-based IBIS). IBIS uses some rationale elements (denoted as nodes) such as issues, positions, arguments, resolutions and several relationships such as more general than, logical successor to, temporal successor to, replaces and similar to, to link the issue discussions.

Procedural Hierarchy of Issues(PHI)

PHI (Procedural Hierarchy of Issues) extended IBIS to noncontroversial issues and redefined the relationships. PHI adds the subissue relationship which means one issue's resolution depends on the resolution of another issue.

Questions, Options, and Criteria (QOC)

QOC (Questions, Options, and Criteria) is used for design space analysis. Similar to IBIS, QOC identifies the key design problems as questions and possible answers to questions as options. In addition, QOC uses criteria to explicitly describe the methods to evaluate the options, such as the requirements to be satisfied or the properties desired. The options are linked with criteria positively or negatively and these links are defined as assessments.

Decision Representation Language (DRL)

DRL (Decision Representation Language) extends the Potts and Bruns model of DR and defines the primary elements as decision problems, alternatives, goals, claims and groups. Lee (1991) has argued that DRL is more expressive than other languages. DRL focuses more on the representation of decision making and its rationale instead of on design rationale.

RATSpeak

Based on DRL, RATSpeak is developed and used as the representation language in SEURAT (Software Engineering Using RATionale). RATSpeak takes into account requirements (functional and non-functional) as part of the arguments for alternatives to the decision problems.

WinWin Spiral Model

And there is an Argument Ontology which is a hierarchy of argument types and includes the types of claims used in the system. The WinWin Spiral Model, which is used in the WinWin approach, adds the WinWin negotiation activities, including identifying key stakeholders of the systems, and identifying the win conditions of each stakeholder and negotiation, into the front of each cycle of the spiral software development model in order to achieve a mutually satisfactory (winwin) agreement for all stakeholders of the project.

In the WinWin Spiral Model, the goals of each stakeholder are defined as Win conditions. Once there is a conflict between win conditions, it is captured as an Issue. Then the stakeholders invent Options and explore trade-offs to resolve the issue. When the issue is solved, an Agreement which satisfies the win conditions of stakeholders and captures the agreed option is achieved. Design rationale behind the decisions is captured during the process of the WinWin model and will be used by stakeholders and the designers to improve their later decision making. The WinWin Spiral model reduces the overheads of the capture of Design Rationale by providing stakeholders a well-defined process to negotiate. In an ontology of decision rationale is defined and their model utilizes the ontology to address the problem of supporting decision maintenance in the WinWin collaboration framework.

Design Recommendation and Intent Model (DRIM)

DRIM (Design Recommendation and Intent Model) is used in SHARED-DRIM. The main structure of DRIM is a proposal which consists of the intents of each designer, the recommendations that satisfy the intents and the justifications of the recommendations. Negotiations are also needed when conflicts exist between the intents of different designers. The recommendation accepted becomes a design decision, and the rationale of the unaccepted but proposed recommendations are also recorded during this process, which can be useful during the iterative design and/or system maintenance.

Applications

Design rationale has the potential to be used in many different ways. One set of uses, defined by Burge and Brown (1998), are:

- Design verification — The design rationale can be used to verify if the design decisions and the product itself are the reflection of what the designers and the users actually wanted.
- Design evaluation — The design rationale is used to evaluate the various design alternatives discussed during the design process.
- Design maintenance — The design rationale helps to determine the changes that are necessary to modify the design.
- Design reuse — The design rationale is used to determine how the existing design could be reused for a new requirement with or without any changes in it. If there is a need to modify the design, then the DR also suggests what needs to be modified in the design.
- Design teaching — The design rationale could be used as a resource to teach people who are unfamiliar with the design and the system.
- Design communication — The design rationale facilitates better communication among people who are involved in the design process and thus helps to come up with a better design.
- Design assistance — The design rationale could be used to verify the design decisions made during the design process.
- Design documentation — The design rationale is used to document the entire design process which involves the meeting room deliberations, alternatives discussed, reasons behind the design decisions and the product overview.

DR is used by research communities in software engineering, mechanical design, artificial intelligence, civil engineering, and human-computer interaction research. In software engineering, it could be used to support the designers ideas during requirement analysis, capturing and documenting design meetings and predicting possible issues due to new design approach. In civil engineering, it helps to coordinate the variety of work that the designers do at the same time in different areas of a construction project. It also help the designers to understand and respect each other's ideas and resolve any possible issues.

The DR can also be used by the project managers to maintain their project plan and the project status up to date. Also, the project team members who missed a design meeting can refer back the DR to learn what was discussed on a particular topic. The unresolved issues captured in DR could be used to organize further meetings on those topics.

Design Rationale helps the designers to avoid the same mistakes made in the previous design. This can also be helpful to avoid duplication of work. In some cases DR could save time and money when a software system is upgraded from its previous versions.

There are several books and articles that provide excellent surveys of rationale approaches applied to HCI, Engineering Design and Software Engineering.

Chapter 6

Domain-Driven Design

Domain-driven design (DDD) is an approach to developing software for complex needs by deeply connecting the implementation to an evolving model of the core business concepts. The premise of domain-driven design is the following:

- Placing the project's primary focus on the core domain and domain logic
- Basing complex designs on a model
- Initiating a creative collaboration between technical and domain experts to iteratively cut ever closer to the conceptual heart of the problem.

Domain-driven design is not a technology or a methodology. DDD provides a structure of practices and terminology for making design decisions that focus and accelerate software projects dealing with complicated domains.

The term was coined by Eric Evans in his book of the same title.

Core definitions

- *Domain* A sphere of knowledge, influence, or activity. The subject area to which the user applies a program is the domain of the software.
- *Model* A system of abstractions that describes selected aspects of a domain and can be used to solve problems related to that domain.
- *Ubiquitous Language* A language structured around the domain model and used by all team members to connect all the activities of the team with the software.
- *Context* The setting in which a word or statement appears that determines its meaning.

Prerequisites for the successful application of DDD

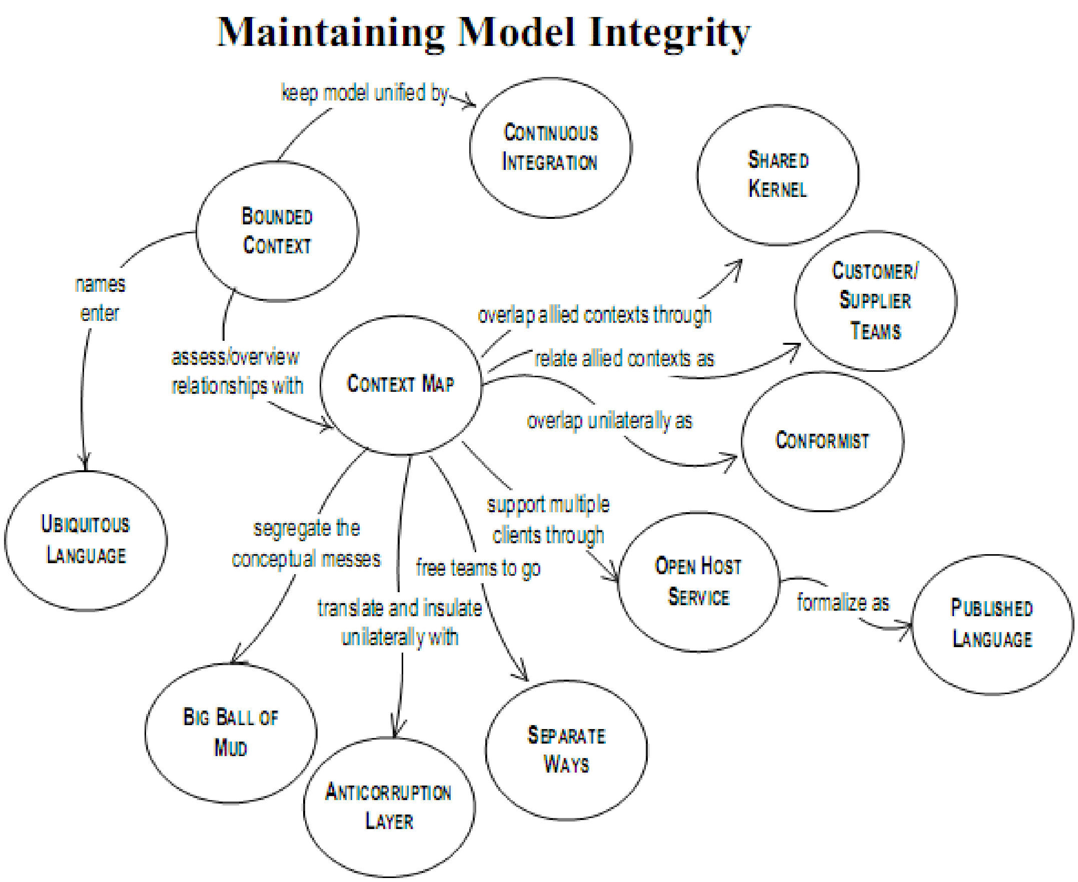
- Your domain is not trivial
- The project team has experience and interest in OOP/OOD
- You have access to domain experts
- You have an iterative process

Strategic domain-driven design

Ideally, we would prefer to have a single, unified model. While this is a noble goal, in reality it always fragments into multiple models. It is more useful to recognize this fact of life and work with it.

Strategic Design is a set of principles for maintaining model integrity, distillation of the Domain Model and working with multiple models.

The following image demonstrates the patterns in Strategic Domain-Driven Design and the relationships between them.



Bounded context

Multiple models are in play on any large project. Yet when code based on distinct models is combined, software becomes buggy, unreliable, and difficult to understand. Communication among team members becomes confused. It is often unclear in what context a model should not be applied.

Therefore: Explicitly define the context within which a model applies. Explicitly set boundaries in terms of team organization, usage within specific parts of the application, and physical manifestations such as code bases and database schemas. Keep the model strictly consistent within these bounds, but don't be distracted or confused by issues outside.

Continuous Integration

When a number of people are working in the same bounded context, there is a strong tendency for the model to fragment. The bigger the team, the bigger the problem, but as few as three or four people can encounter serious problems. Yet breaking down the system into ever-smaller contexts eventually loses a valuable level of integration and coherency.

Therefore: Institute a process of merging all code and other implementation artifacts frequently, with automated tests to flag fragmentation quickly. Relentlessly exercise the ubiquitous language to hammer out a shared view of the model as the concepts evolve in different people's heads.

Context map

An individual bounded context leaves some problems in the absence of a global view. The context of other models may still be vague and in flux.

People on other teams won't be very aware of the context bounds and will unknowingly make changes that blur the edges or complicate the interconnections. When connections must be made between different contexts, they tend to bleed into each other.

Therefore: Identify each model in play on the project and define its bounded context. This includes the implicit models of non- object-oriented subsystems. Name each bounded context, and make the names part of the ubiquitous language. Describe the points of contact between the models, outlining explicit translation for any communication and highlighting any sharing. Map the existing terrain

Building Blocks of DDD

In the book *Domain-Driven Design*, a number of high-level concepts and practices are articulated, such as *ubiquitous language* meaning that the domain model should form a *common language* given by domain experts for describing system requirements, that works equally well for the business users or sponsors and for the software developers. The book is very focused at describing the domain layer that is one of the common layers in an object-oriented system with a multilayered architecture. In DDD, there are artifacts to express, create, and retrieve domain models:

- **Entity**: An object that is not defined by its attributes, but rather by a thread of continuity and its identity.

Example: Most airlines distinguish each seat uniquely on every flight. Each seat is an entity in this context. However, Southwest Airlines (or EasyJet/RyanAir for Europeans) does not distinguish between every seat; all seats are the same. In this context, a seat is actually a value object.

- **Value Object**: An object that contains attributes but has no conceptual identity. They should be treated as immutable.

Example: When people exchange dollar bills, they generally do not distinguish between each unique bill; they only are concerned about the face value of the dollar bill. In this context, dollar bills are value objects. However, the Federal Reserve may be concerned about each unique bill; in this context each bill would be an entity.

- **Aggregate**: A collection of objects that are bound together by a root entity, otherwise known as an aggregate root. The aggregrate root guarantees the consistency of changes being made within the aggregate by forbidding external objects from holding references to its members.

Example: When you drive a car, you do not have to worry about moving the wheels forward, making the engine combust with spark and fuel, etc.; you are simply driving the car. In this context, the car is an aggregate of several other objects and serves as the aggregate root to all of the other systems.

- **Service**: When an operation does not conceptually belong to any object. Following the natural contours of the problem, you can implement these operations in services. The Service concept is called "Pure Fabrication" in GRASP.
- **Repository**: methods for retrieving domain objects should delegate to a specialized Repository object such that alternative storage implementations may be easily interchanged.
- **Factory**: methods for creating domain objects should delegate to a specialized Factory object such that alternative implementations may be easily interchanged.

Relationship to other ideas

Object-oriented analysis and design

> Although in theory, the general idea of DDD need not be restricted to object-oriented approaches, in practice DDD seeks to exploit the powerful advantages that object-oriented techniques make possible.

Model-driven engineering (MDE)

Model-driven architecture (MDA)

> While DDD is compatible with MDA, the intent of the two concepts is somewhat different. MDA is concerned more with the means of translating a model into code for different technology platforms than with the practice of defining better domain models.

POJOs and POCOs

POJOs and POCOs are technical implementation concepts, specific to the Java and .NET framework respectively. However, the emergence of the terms POJO and POCO, reflect a growing view that, within the context of either of those technical platforms, domain objects should be defined purely to implement the business behaviour of the corresponding domain concept, rather than be defined by the requirements of a more specific technology framework.

The naked objects pattern

This pattern is based on the premise that if you have a good enough domain model, the user interface can simply be a reflection of this domain model; and that if you require the user interface to be direct reflection of the domain model then this will force the design of a better domain model.

Domain-specific programming language (DSL)

DDD does not specifically require the use of a DSL, though it could be used to help define a DSL and support methods like domain-specific multimodeling.

Aspect-oriented programming (AOP)

AOP makes it easy to factor out technical concerns (such as security, transaction management, logging) from a domain model, and as such makes it easier to design and implement domain models that focus purely on the business logic.

- Command and Query Responsibility Segregation (CQRS): CQRS is simply the creation of two objects where there was previously only one. The separation occurs based upon whether the methods are a command or a query (the same definition that is used by Meyer in Command and Query Separation, a command is any method that mutates state and a query is any method that returns a value).

Software tools to support domain-driven design

Practicing DDD does not depend upon the use of any particular software tool or framework. Nonetheless, there is a growing number of open-source tools and frameworks that provide support to the specific patterns advocated in Evans' book or the general approach of DDD. Among these are:

- Actifsource is a plug-in for Eclipse which enables software development combining DDD with model-driven engineering and code generation.
- Apache Isis is the successor to the original Java implementation of the naked objects pattern. It directly supports the DDD concepts of Entity, Value, Repository, Factory, Domain Service. Architecturally it provides automatic dependency injection; with multiple viewers (web-based and RESTful), pluggable security and pluggable persistence stores.
- Castle Windsor/MicroKernel: an Inversion of Control/Dependency Injection container for the Microsoft.NET Framework to provide Services, Repositories and Factories to consumers.
- CodeFluent Entities: a model-driven software factory. It provides architects and developers a structured method and the corresponding tools to develop .NET applications, based on any type of architecture, from an ever changing business model and rules.

- DataObjects.Net: rapid database application development framework supporting object-relational mapping and DDD.
- Domdrides: A useful library for implementing DOMain-DRIven DESign in Java.
- ECO (Domain Driven Design): Framework with database, class, code and state machine generation from UML diagrams by CapableObjects.
- FLOW3: A PHP based application framework centered on DDD principles. Fosters clean Domain Models and supports the concept of Repository, Entity and Value Object. Also provides Dependency Injection and an AOP framework.
- Habanero.NET (Habanero) is an Open Source Enterprise Application framework for creating Enterprise applications using the principles of Domain-driven design and implemented in .NET.
- JavATE: a set of Java libraries that enables application development inspired by domain driven design.It gives you standard interfaces and implementations for the domain driven design building blocks so you can focus on your strategic design instead of reinventing the wheel of the building blocks each time.
- ManyDesigns Portofino is an open source, model-driven web-application framework for high productivity and maintainability. It provides CRUD forms, relationships, workflow management, dashboards, breadcrumbs, searches, single sign-on, permissions, and reporting.
- Metawidget: a User Interface widget that populates itself, at runtime, with UI components to match the properties of domain objects.
- Naked Objects MVC: implements the naked objects pattern; runs under the ASP.NET MVC Framework; supports dependency injection; and provides re-usable implementations of the DDD concepts of Repository, Factory and Service.
- NReco: lightweight open-source domain-specific MDD framework for .NET. Integrated with JQuery, Open NIC.NET, OGNL, Log4Net, Lucene.NET, SemWeb etc.
- OpenMDX: Open source, Java based, MDA Framework supporting Java SE, Java EE, and .NET. OpenMDX differs from typical MDA frameworks in that *"use models to directly drive the runtime behavior of operational systems"*.
- OpenXava: Generates an AJAX application from JPA entities. Only it's needed to write the domain classes to obtain a ready to use application.
- re-motion: DDD framework for .NET. Also contains components to hide implementation details of data access and object binding to ASP.NET forms.
- Roma Meta Framework: DDD centric framework. The innovative holistic approach lets the designer/developer to view anything as a POJO: GUI, I18N, Persistence, etc.
- Sculptor: Advanced Open source code-generation MDA tool for Java, that uses DDD as primary language and support also Event-driven architecture and CQRS. Sculptor is using Eclipse oAW as generation framework and generate code which fits to many standard/well-know frameworks and engines. It can generate also many types of GUIs, from GUI design definition.
- Sculpture - Model Your Life: Sculpture is one of the most powerful Open Source Model-driven development Generators. It can be used for everything from simple CRUD applications to complex enterprise applications - Sculpture comes with a

list of ready-made Molds for common architectures like NHibernate, Entity Framework, WCF, CSLA, Silverlight, WPF, and ASP.NET MVC.

- Strandz: A DDD framework that provides implementation independence from both the UI layer and domain layer of the application. The programmer constructs a wire model of the application using special classes.
- TrueView for .NET: An easy-to-use framework that supports DDD and the naked objects pattern. Useful for teams starting out with DDD.
- Tynamo: Tynamo is open source, model-driven, full-stack web framework based on Tapestry 5, implemented in Java in the spirit of naked objects pattern

Chapter 7

Object-Oriented Design

Object-oriented design is the process of planning a system of interacting objects for the purpose of solving a software problem. It is one approach to software design.

Overview

An object contains encapsulated data and procedures grouped together to represent an entity. The 'object interface', how the object can be interacted with, is also defined. An object-oriented program is described by the interaction of these objects. Object-oriented design is the discipline of defining the objects and their interactions to solve a problem that was identified and documented during object-oriented analysis.

What follows is a description of the class-based subset of object-oriented design, which does not include object prototype-based approaches where objects are not typically obtained by instancing classes but by cloning other (prototype) objects.

Object-oriented design topics

Input (sources) for object-oriented design

The input for object-oriented design is provided by the output of object-oriented analysis. Realize that an output artifact does not need to be completely developed to serve as input of object-oriented design; analysis and design may occur in parallel, and in practice the results of one activity can feed the other in a short feedback cycle through an iterative process. Both analysis and design can be performed incrementally, and the artifacts can be continuously grown instead of completely developed in one shot.

Some typical input artifacts for object-oriented design are:

- Conceptual model: Conceptual model is the result of object-oriented analysis, it captures concepts in the problem domain. The conceptual model is explicitly chosen to be independent of implementation details, such as concurrency or data storage.

- Use case: Use case is a description of sequences of events that, taken together, lead to a system doing something useful. Each use case provides one or more scenarios that convey how the system should interact with the users called actors to achieve a specific business goal or function. Use case actors may be end users or other systems. In many circumstances use cases are further elaborated into use case diagrams. Use case diagrams are used to identify the actor (users or other systems) and the processes they perform.

- System Sequence Diagram: System Sequence diagram (SSD) is a picture that shows, for a particular scenario of a use case, the events that external actors generate, their order, and possible inter-system events.

- User interface documentations (if applicable): Document that shows and describes the look and feel of the end product's user interface. It is not mandatory to have this, but it helps to visualize the end-product and therefore helps the designer.

- Relational data model (if applicable): A data model is an abstract model that describes how data is represented and used. If an object database is not used, the relational data model should usually be created before the design, since the strategy chosen for object-relational mapping is an output of the OO design process. However, it is possible to develop the relational data model and the object-oriented design artifacts in parallel, and the growth of an artifact can stimulate the refinement of other artifacts.

Object-oriented concepts

The five basic concepts of object-oriented design are the implementation level features that are built into the programming language. These features are often referred to by these common names:

- Object/Class: A tight coupling or association of data structures with the methods or functions that act on the data. This is called a *class*, or *object* (an object is created based on a class). Each object serves a separate function. It is defined by its properties, what it is and what it can do. An object can be part of a class, which is a set of objects that are similar.

- Information hiding: The ability to protect some components of the object from external entities. This is realized by language keywords to enable a variable to be declared as *private* or *protected* to the owning *class*.
- Inheritance: The ability for a *class* to extend or override functionality of another *class*. The so-called *subclass* has a whole section that is derived (inherited) from the *superclass* and then it has its own set of functions and data.
- Interface: The ability to defer the implementation of a *method*. The ability to define the *functions* or *methods* signatures without implementing them.
- Polymorphism: The ability to replace an *object* with its *subobjects*. The ability of an *object-variable* to contain, not only that *object*, but also all of its *subobjects*.

Designing concepts

- Defining objects, creating class diagram from conceptual diagram: Usually map entity to class.

- Identifying attributes.

- Use design patterns (if applicable): A design pattern is not a finished design, it is a description of a solution to a common problem, in a context. The main advantage of using a design pattern is that it can be reused in multiple applications. It can also be thought of as a template for how to solve a problem that can be used in many different situations and/or applications. Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved.

- Define application framework (if applicable): Application framework is a term usually used to refer to a set of libraries or classes that are used to implement the standard structure of an application for a specific operating system. By bundling a large amount of reusable code into a framework, much time is saved for the developer, since he/she is saved the task of rewriting large amounts of standard code for each new application that is developed.

- Identify persistent objects/data (if applicable): Identify objects that have to last longer than a single runtime of the application. If a relational database is used, design the object relation mapping.

- Identify and define remote objects (if applicable).

Output (deliverables) of object-oriented design

- Sequence Diagrams: Extend the System Sequence Diagram to add specific objects that handle the system events.

  A sequence diagram shows, as parallel vertical lines, different processes or objects that live simultaneously, and, as horizontal arrows, the messages exchanged between them, in the order in which they occur.

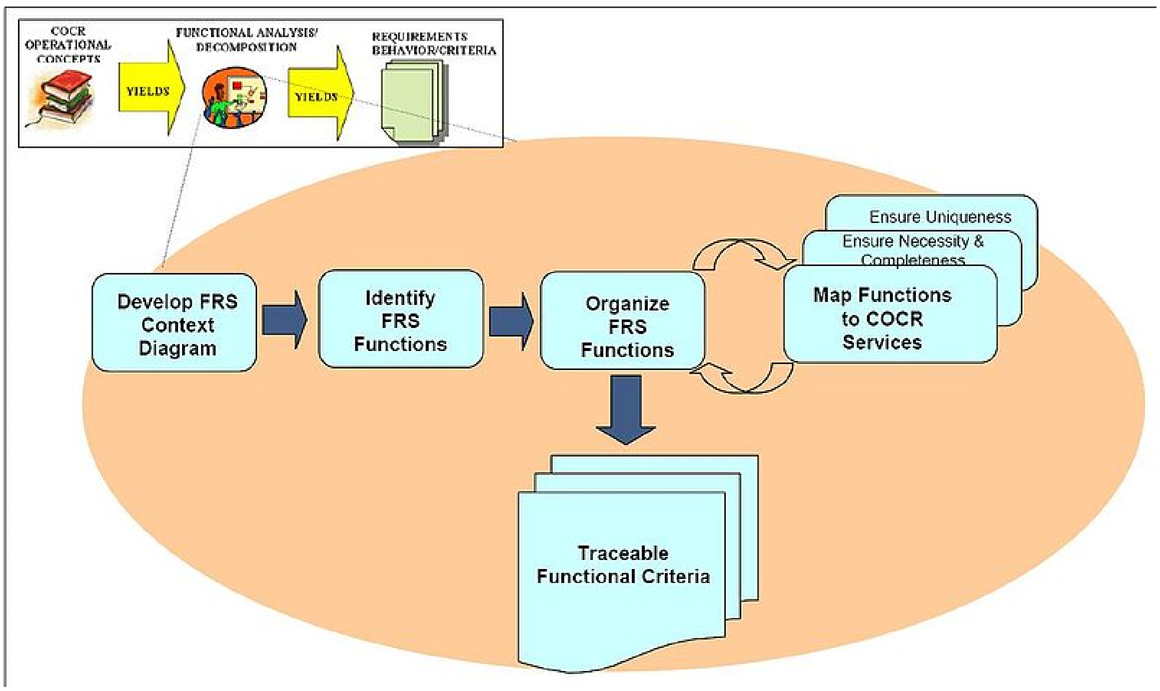
- Class diagram: A class diagram is a type of static structure UML diagram that describes the structure of a system by showing the system's classes, their attributes, and the relationships between the classes. The messages and classes identified through the development of the sequence diagrams can serve as input to the automatic generation of the global class diagram of the system.

Some design principles and strategies

- Dependency injection: The basic idea is that if an object depends upon having an instance of some other object then the needed object is "injected" into the dependent object; for example, being passed a database connection as an argument to the constructor instead of creating one internally.
- Acyclic dependencies principle: The dependency graph of packages or components should have no cycles. This is also referred to as having a directed acyclic graph. For example, package C depends on package B, which depends on package A. If package A also depended on package C, then you would have a cycle.
- Composite reuse principle: Favor polymorphic composition of objects over inheritance.

Chapter 8

Structured Analysis



Example of a Structured Analysis approach.

Structured Analysis (SA) in software engineering and its allied technique, Structured Design (SD), are methods for analyzing and converting business requirements into specifications and ultimately, computer programs, hardware configurations and related manual procedures.

Structured analysis and design techniques are fundamental tools of systems analysis, and developed from classical systems analysis of the 1960s and 1970s.

Objectives of Structured Analysis

Structured Analysis became popular in the 1980's and is still used by many. The analysis consists of interpreting the system concept (or real world) into data and control terminology, that is into data flow diagrams. The flow of data and control from bubble to data store to bubble can be very hard to track and the number of bubbles can get to be extremely large. One approach is to first define events from the outside world that require the system to react, then assign a bubble to that event, bubbles that need to interact are then connected until the system is defined. This can be rather overwhelming and so the bubbles are usually grouped into higher level bubbles. Data Dictionaries are needed to describe the data and command flows and a process specification is needed to capture the transaction/transformation information.

SA and SD were accompanied by notational methods including structure charts, data flow diagrams and data model diagrams, of which there were many variations, including those developed by Tom DeMarco, Ken Orr, Larry Constantine, Vaughn Frick, Ed Yourdon, Steven Ward, Peter Chen, and others.

These techniques were combined in various published System Development Methodologies, including Structured Systems Analysis and Design Method, Profitable Information by Design (PRIDE), Nastec Structured Analysis & Design, SDM/70 and the Spectrum Structured system development methodology.

History

Structured analysis is part of a series of structured methods, that "represent a collection of analysis, design, and programming techniques that were developed in response to the problems facing the software world from the 1960s to the 1980s. In this timeframe most commercial programming was done in Cobol and Fortran, then C and BASIC. There was little guidance on “good” design and programming techniques, and there were no standard techniques for documenting requirements and designs. Systems where getting larger and more complex, and the information system development became harder and harder to do so". As a way to help manage large and complex software.

Since the end 1960 multiple Structured Methods emerged:

- Structured programming in circa 1967 with Edsger Dijkstra.
- Structured Design around 1975 with Larry Constantine, Ed Yourdon and Wayne Stevens.
- Jackson Structured Programming in circa 1975 developed by Michael A. Jackson
- Structured Analysis in circa 1978 with Tom DeMarco, Yourdon, Gane & Sarson, McMenamin & Palmer.
- Structured Analysis and Design Technique (SADT) developed by Douglas T. Ross
- Yourdon Structured Method developed by Edward Yourdon.
- Structured Analysis and System Specification published in 1979 by Tom DeMarco.
- Structured Systems Analysis and Design Method (SSADM) first presented in 1983 developed by the UK Office of Government Commerce.
- IDEF0 based on SADT, developed by Douglas T. Ross in 1985.
- Information Engineering in circa 1990 with James Martin.

According to Hay (1999) "information engineering was a logical extension of the structured techniques that were developed during the 1970's. Structured programming led to structured design, which in turn led to structured systems analysis. These techniques were characterized by their use of diagrams: structure charts for structured design, and data flow diagrams for structured analysis, both to aid in communication between users and developers, and to improve the analyst's and the designer's discipline. During the 1980's, tools began to appear which both automated the drawing of the diagrams, and kept track of the things drawn in a data dictionary". After the example of computer-aided design and computer-aided manufacturing (CAD/CAM), the use of these tools was named Computer-aided software engineering (CASE).

Structured analysis topics

Single abstraction mechanism

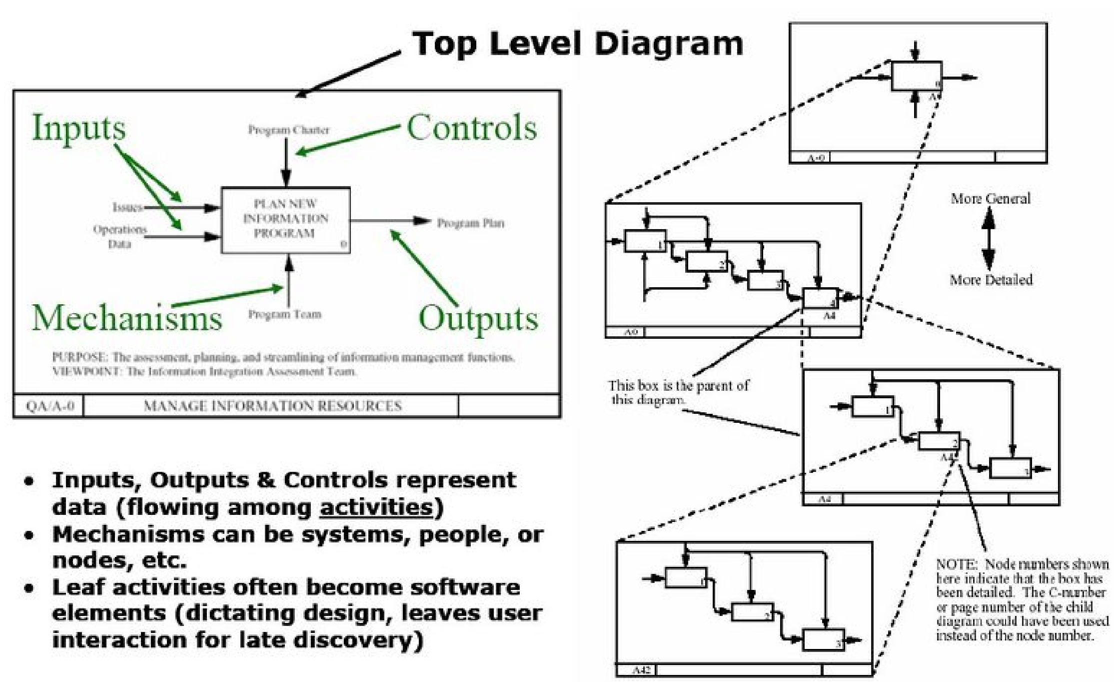


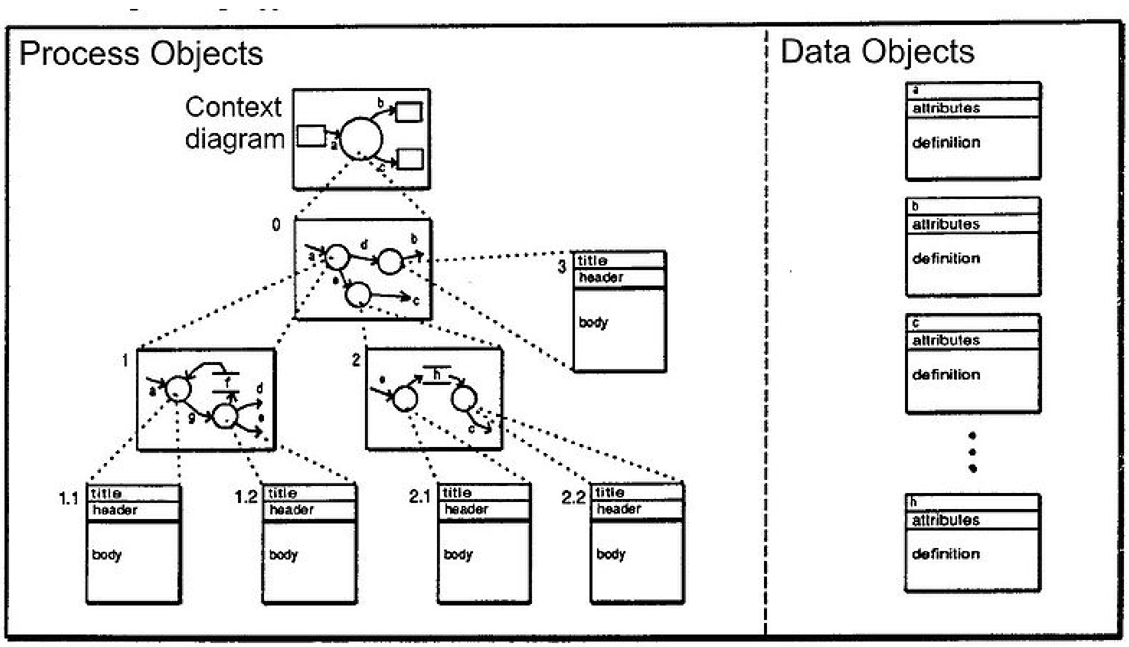
Image:Structured Analysis example.

Structured analysis typically creates a hierarchy employing a single abstraction mechanism. The structured analysis method can employ IDEF (see figure), is process driven, and starts with a purpose and a viewpoint. This method identifies the overall function and iteratively divides functions into smaller functions, preserving inputs, outputs, controls, and mechanisms necessary to optimize processes. Also known as a functional decomposition approach, it focuses on cohesion within functions and coupling between functions leading to structured data.

The functional decomposition of the structured method describes the process without delineating system behavior and dictates system structure in the form of required functions. The method identifies inputs and outputs as related to the activities. One reason for the popularity of structured analysis is its intuitive ability to communicate high-level processes and concepts, whether single system or enterprise levels. Discovering how objects might support functions for commercially prevalent object-oriented development is unclear. In contrast to IDEF, the UML is interface driven with multiple abstraction mechanisms useful in describing service-oriented architectures (SOAs).

Approach

Structured Analysis views a system from the perspective of the data flowing through it. The function of the system is described by processes that transform the data flows. Structured analysis takes advantage of information hiding through successive decomposition (or top down) analysis. This allows attention to be focused on pertinent details and avoids confusion from looking at irrelevant details. As the level of detail increases, the breadth of information is reduced. The result of structured analysis is a set of related graphical diagrams, process descriptions, and data definitions. They describe the transformations that need to take place and the data required to meet a system's functional requirements.



The structured analyse approach develops perspectives on both process objects and data objects.

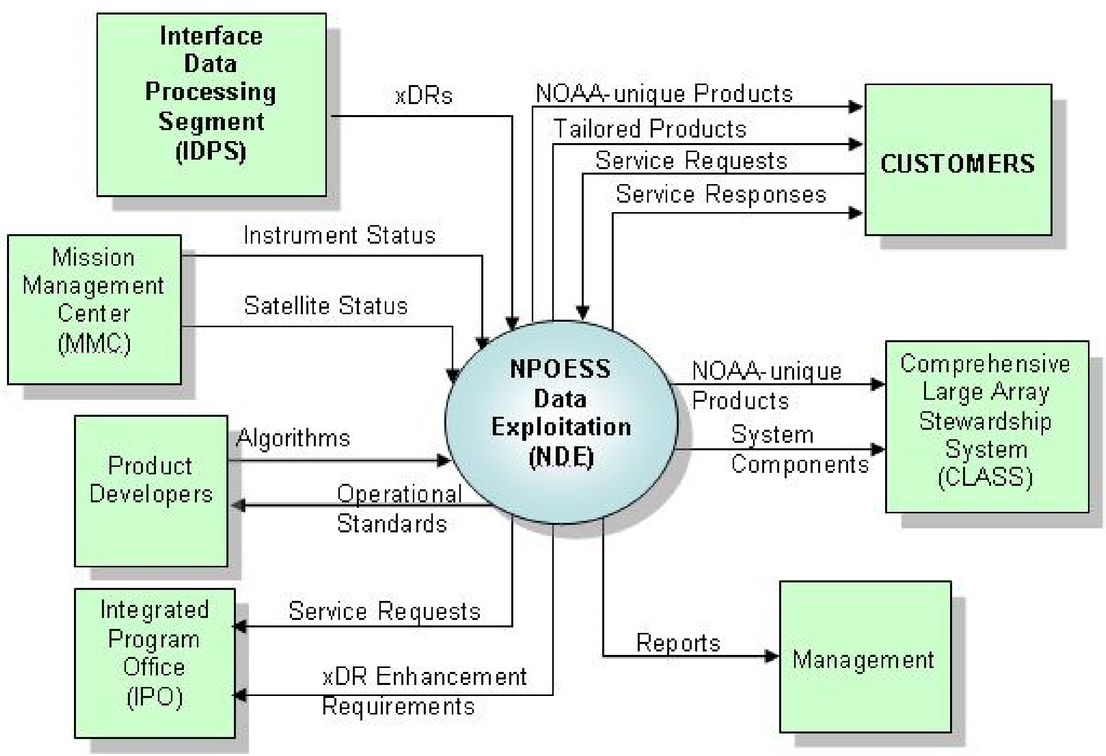
De Marco's approach consists of the following objects (see figure):

- Context diagram
- dataflow diagram,

- process specifications, and
- a data dictionary,

Hereby the Data flow diagrams (DFDs) are directed graphs. The arcs represent data, and the nodes (circles or bubbles) represent processes that transform the data. A process can be further decomposed to a more detailed DFD which shows the subprocesses and data flows within it. The subprocesses can in turn be decomposed further with another set of DFDs until their functions can be easily understood. Functional primitives are processes which do not need to be decomposed further. Functional primitives are described by a process specification (or mini-spec). The process specification can consist of pseudo-code, flowcharts, or structured English. The DFDs model the structure of the system as a network of interconnected processes composed of functional primitives. The data dictionary is a set of entries (definitions) of data flows, data elements, files. and data bases. The data dictionary enmes are partitioned in a topdown manner. They can be referenced in other data dictionary entries and in data flow diagrams.

Context diagram

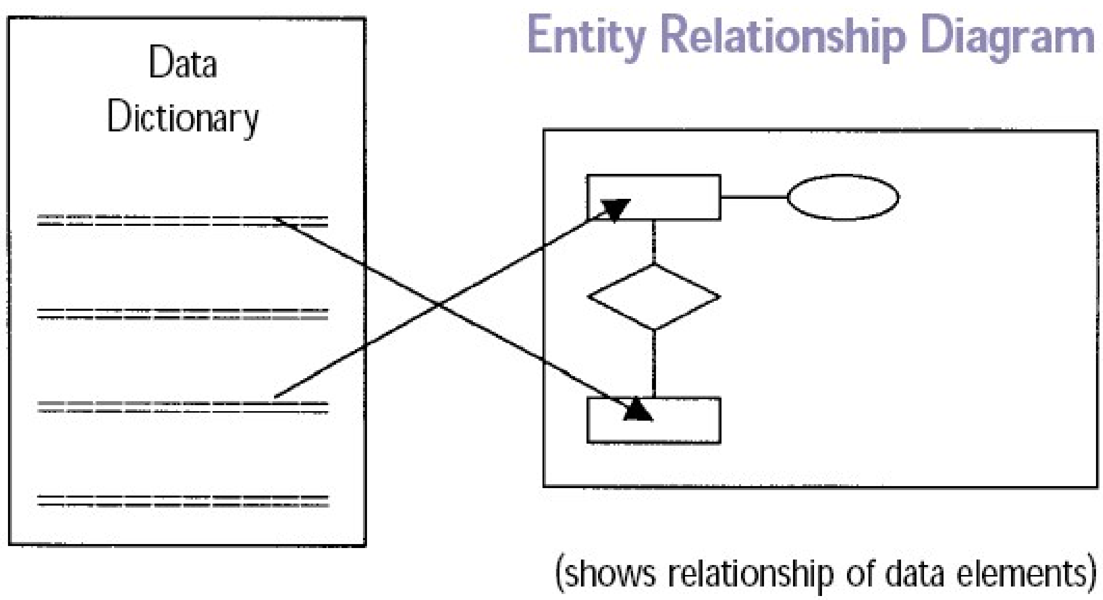


Example of a System context diagram.

Context diagrams are diagrams that represent the actors outside a system that could interact with that system. This diagram is the highest level view of a system, similar to Block Diagram, showing a, possibly software-based, system as a whole and its inputs and outputs from/to external factors.

This type of diagram according to Kossiakoff (2003) usually "pictures the system at the center, with no details of its interior structure, surrounded by all its interacting systems, environment and activities. The objective of a system context diagram is to focus attention on external factors and events that should be considered in developing a complete set of system requirements and constraints". System context diagram are related to Data Flow Diagram, and show the interactions between a system and other actors with which the system is designed to face. System context diagrams can be helpful in understanding the context in which the system will be part of software engineering.

Data dictionary

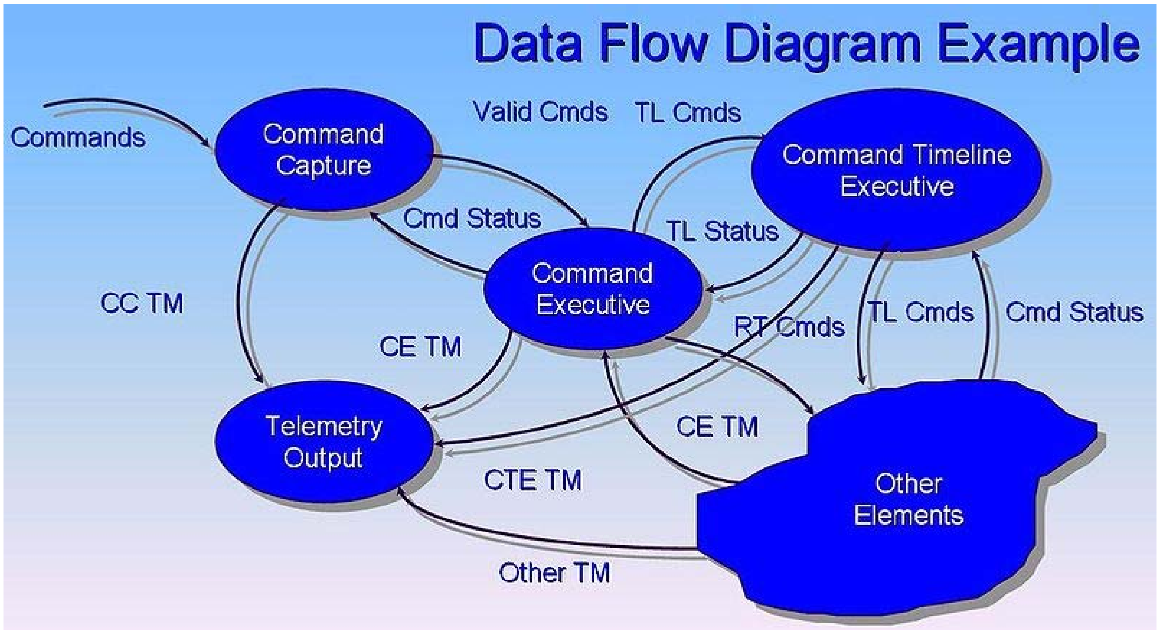


Entity relationship diagram, essential for the design of database tables, extracts, and metadata.

A data dictionary or *database dictionary* is a file that defines the basic organization of a database. A database dictionary contains a list of all files in the database, the number of records in each file, and the names and types of each data field. Most database management systems keep the data dictionary hidden from users to prevent them from accidentally destroying its contents. Data dictionaries do not contain any actual data from the database, only bookkeeping information for managing it. Without a data dictionary, however, a database management system cannot access data from the database.

Database users and application developers can benefit from an authoritative data dictionary document that catalogs the organization, contents, and conventions of one or more databases. This typically includes the names and descriptions of various tables and fields in each database, plus additional details, like the type and length of each data element. There is no universal standard as to the level of detail in such a document, but it is primarily a distillation of metadata about database structure, not the data itself. A data dictionary document also may include further information describing how data elements are encoded. One of the advantages of well-designed data dictionary documentation is that it helps to establish consistency throughout a complex database, or across a large collection of federated databases.

Data Flow Diagrams



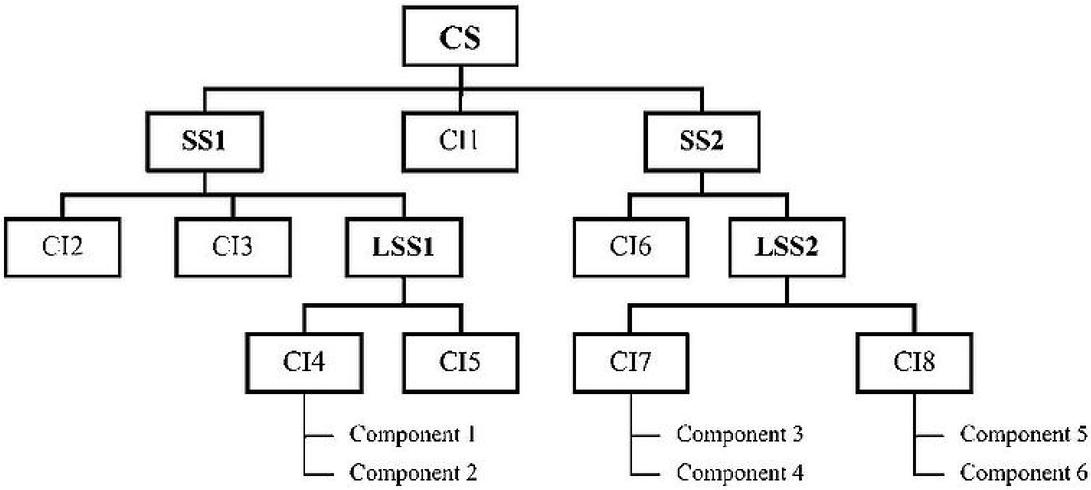
Data Flow Diagram example.

A Data Flow Diagram (DFD) is a graphical representation of the "flow" of data through an information system. It differs from the system flowchart as it shows the flow of data through processes instead of hardware. Data flow diagrams were invented by Larry Constantine, developer of structured design, based on Martin and Estrin's "data flow graph" model of computation.

It is common practice to draw a System Context Diagram first which shows the interaction between the system and outside entities. The DFD is designed to show how a system is divided into smaller portions and to highlight the flow of data between those parts. This context-level Data flow diagram is then "exploded" to show more detail of the system being modeled.

Data flow diagrams (DFDs) are one of the three essential perspectives of Structured Systems Analysis and Design Method (SSADM). The sponsor of a project and the end users will need to be briefed and consulted throughout all stages of a system's evolution. With a dataflow diagram, users are able to visualize how the system will operate, what the system will accomplish, and how the system will be implemented. The old system's dataflow diagrams can be drawn up and compared with the new system's dataflow diagrams to draw comparisons to implement a more efficient system. Dataflow diagrams can be used to provide the end user with a physical idea of where the data they input ultimately has an effect upon the structure of the whole system from order to dispatch to recook. How any system is developed can be determined through a dataflow diagram.

Structure Chart



A Configuration System Structure Chart.

A Structure Chart (SC) is a chart, that shows the breakdown of the configuration system to the lowest manageable levels. This chart is used in structured programming to arrange the program modules in a tree structure. Each module is represented by a box which contains the name of the modules. The tree structure visualizes the relationships between the modules.

In structured analysis structure charts are used to specify the high-level design, or architecture, of a computer program. As a design tool, they aid the programmer in dividing and conquering a large software problem, that is, recursively breaking a problem down into parts that are small enough to be understood by a human brain. The process is called top-down design, or functional decomposition. Programmers use a structure chart to build a program in a manner similar to how an architect uses a blueprint to build a house. In the design stage, the chart is drawn and used as a way for the client and the various software designers to communicate. During the actual building of the program (implementation), the chart is continually referred to as the master-plan.

Structured Design

Structured Design (SD) is concerned with the development of modules and the synthesis of these modules in a so called "module hierarchy". In order to design optimal module structure and interfaces two principles are crucial:

- *Cohesion* which is "concerned with the grouping of functionally related processes into a particular module", and

- *Coupling* relates to "the flow of information, or parameters, passed between modules. Optimal coupling reduces the interfaces of modules, and the resulting complexity of the software".

Page-Jones (1980) has proposed his own approach, which consists of three main objects: structure charts, module specifications and a data dictionary. The structure chart aims to show "the module hierarchy or calling sequence relationship of modules. There is a module specification for each module shown on the structure chart. The module specifications can be composed of pseudo-code or a program design language. The data dictionary is like that of structured analysis. At this stage in the software development lifecycle, after analysis and design have been performed, it is possible to automatically generate data type declarations", and procedure or subroutine templates.

Structured query language

The structured query language (SQL) is a standardized language for querying information from a database. SQL was first introduced as a commercial database system in 1979 and has since been the favorite query language for database management systems running on minicomputers and mainframes. Increasingly, however, SQL is being supported by PC database systems because it supports distributed databases. This enables several users on a computer network to access the same database simultaneously. Although there are different dialects of SQL, it is nevertheless the closest thing to a standard query language that currently exists.

Criticisms

Problems with data flow diagrams have been:

1. choosing bubbles appropriately,
2. partitioning those bubbles in a meaningful and mutually agreed upon manner,
3. the size of the documentation needed to understand the Data Flows,
4. still strongly functional in nature and thus subject to frequent change,
5. though “data” flow is emphasized, “data” modeling is not, so there is little understanding of just what the subject matter of the system is about, and
6. not only is it hard for the customer to follow how the concept is mapped into these data flows and bubbles, it has also been very hard for the designers who must shift the DFD organization into an implementable format

Chapter 9

Shlaer–Mellor Method

The **Shlaer-Mellor** method, developed by Sally Shlaer and Stephen Mellor, is one of a number of object-oriented analysis (OOA) / object-oriented design (OOD) methods which arrived in the late 1980s in response to established weaknesses in the existing structured analysis and structured design (SASD) techniques in use by (primarily) software engineers.

Of these well known problems, Shlaer and Mellor chose to address:

- The complexity of designs generated through the use of SASD.
- The problem of maintaining analysis and design documentation over time.

Background

The general solution taken by OOA/OOD methods to these particular problems with SASD, was to switch from **functional decomposition** to **semantic decomposition**. That is to say, describing the control of a passenger train as *load passengers, close doors, start train, stop train, open doors, unload passengers* becomes a design focused on the behavior of doors, brakes, and engines, and how those "domains" (doors, brakes, etc.) are related and interact. So door behavior, for example, becomes localized in one part of the design rather than being distributed across the design.

Translation v. elaboration

What makes Shlaer-Mellor unique among OOA/OOD methods is the degree to which object-oriented semantic decomposition is taken, the precision of the **Shlaer-Mellor Notation** used to express the analysis, and the defined behavior of that analysis model at run-time. The goal of the Shlaer-Mellor method is to make the documented analysis so precise that it is possible to implement the analysis model directly **by translation** rather than **by elaboration**. In Shlaer-Mellor terminology this is called **recursive design**. In current (2006) terminology, we would say the Shlaer-Mellor method uses a form of Model-driven Architecture (MDA) normally associated with the Unified Modeling Language (UML).

By taking this translative approach, the implementation is always generated (either manually, or more typically, automatically) directly from the analysis. This is not to say that there is *no* design in Shlaer-Mellor, rather that there is considered to be a *virtual machine* that can execute any Shlaer-Mellor analysis model for any particular hardware/software platform combination. This is similar in concept to the virtual machines at the heart of the Java programming language and the Ada programming language, but existing at analysis level rather than at programming level. Once designed and implemented, such a virtual machine is re-usable across a range of applications. Shlaer-Mellor virtual machines are available commercially from a number of tool vendors, notably Kennedy Carter in the UK, and Mentor Graphics in the USA.

Semantic decomposition

In Shlaer-Mellor the result of a semantic decomposition is a collection of (problem) **domains**.

The first level of semantic decomposition found in Shlaer-Mellor is the split between analysis and design models. The analysis domain expresses precisely *what* the system must do, the design domain is a model of how the Shlaer-Mellor virtual machine operates for a particular hardware and software platform. These models are disjoint, the only connection being the notation used to express the analysis model. The analysis domain is considered to be dependent upon the design domain.

The second level of semantic decomposition comes within the analysis domain where system requirements are modelled, and grouped, around specific, disjoint, subject matter ontologies. To return to the earlier train controller example, individual semantic models may be created based on doors, motors, braking system domains. Each grouping is considered, and modelled, independently. The only defined relationship between the groupings are dependencies e.g. a train controller may depend on both door management and motor control. Motor control may depend upon traction and braking systems.

Domain models of doors, motors, and braking systems would typically be considered as generic re-usable **service domains** whereas the train controller domain is likely to be a very product-specific **application domain**. What makes a particular system/product is specific populations of objects in each domain and the **counterpart mappings** defined between the domains (a decelerator object in the motor domain may be an actuator in the braking system). The mapping between objects (and typically events between objects' finite state machines) in two different domains is called a **bridge**.

Precise action language

One of the requirement for automated code generation is to precisely model the actions within the **finite state machines** used to express dynamic behaviour of Shlaer-Mellor objects. Since Shlaer-Mellor uses only **Moore State Models**, this means specifically state actions. Shlaer-Mellor is unique amongst OOA methods in expressing such sequential behavior graphically as **Action Data Flow Diagrams** (ADFDs). The (relatively) trivial behaviour of an action on an ADFD is then expressed textually.

There has never been a universally agreed textual language to express actions within the Shlaer-Mellor community. Shlaer and Mellor's own version was, and is, copyright and available only through their own commercial toolchain. Other tool vendors have similarly copyrighted and controlled their own action languages. This problem has dogged all model-driven architecture to the present day, with even the Unified Modeling Language having no unified action language to define its state actions. The best that has been achieved is the **UML Action Semantics** specification created as Shlaer-Mellor migrated to the UML notation, becoming **Executable UML**. This however describes *what* features an action language must possess, *not* its grammar. Imagine describing the requirements of spoken English without defining a grammar or vocabulary.

In practice, all tools that support Shlaer-Mellor's Recursive Design, or more generally Model Driven Architecture, provide a (vendor specific) precise action language. Usually such action languages do not support the ADFD approach, and the entire state action is written in textual form.

Test and simulation

The translative approach of the Shlaer-Mellor method lends itself to automated test and simulation environments (by switching the target platform during code generation), and this may partly explain the popularity of Shlaer-Mellor and other MDA-based methods when developing *embedded systems*, where testing on target systems e.g. mobile phones or engine management systems, is particularly difficult.

What makes such testing useful and productive is the concept of the Shlaer-Mellor virtual machine. As with most OOA/OOD methods, Shlaer-Mellor is an event-driven, message-passing environment. Onto this generic view, the Shlaer-Mellor virtual machine mandates a prioritised event mechanism built around **Moore State Models**, which allows for concurrent execution of actions in different state machines. Since any implementation of Shlaer-Mellor requires this model to be fully supported, testing under simulation can be a very close model of testing on target platform. Whilst functionality heavily dependent upon timing constraints may be difficult to test, the majority of system behaviour is highly predictable due to the prioritized execution model.

Chapter 10

Object-Oriented Analysis and Design

Object-oriented analysis and design (OOAD) is a software engineering approach that models a system as a group of interacting objects. Each object represents some entity of interest in the system being modeled, and is characterised by its class, its state (data elements), and its behavior. Various models can be created to show the static structure, dynamic behavior, and run-time deployment of these collaborating objects. There are a number of different notations for representing these models, such as the Unified Modeling Language (UML).

Object-oriented analysis (OOA) applies object-modelling techniques to analyze the functional requirements for a system. Object-oriented design (OOD) elaborates the analysis models to produce implementation specifications. OOA focuses on *what* the system does, OOD on *how* the system does it.

Object-oriented systems

An object-oriented system is composed of objects. The behavior of the system results from the collaboration of those objects. Collaboration between objects involves them sending messages to each other. Sending a message differs from calling a function in that when a target object receives a message, it itself decides what function to carry out to service that message. The same message may be implemented by many different functions, the one selected depending on the state of the target object.

The implementation of "message sending" varies depending on the architecture of the system being modeled, and the location of the objects being communicated with.

Object-oriented analysis

Object-oriented analysis (OOA) looks at the problem domain, with the aim of producing a conceptual model of the information that exists in the area being analyzed. Analysis models do not consider any implementation constraints that might exist, such as concurrency, distribution, persistence, or how the system is to be built. Implementation constraints are dealt during object-oriented design (OOD). Analysis is done before the Design.

The sources for the analysis can be a written requirements statement, a formal vision document, interviews with stakeholders or other interested parties. A system may be divided into multiple domains, representing different business, technological, or other areas of interest, each of which are analyzed separately.

The result of object-oriented analysis is a description of *what* the system is functionally required to do, in the form of a conceptual model. That will typically be presented as a set of use cases, one or more UML class diagrams, and a number of interaction diagrams. It may also include some kind of user interface mock-up. The purpose of object oriented analysis is to develop a model that describes computer software as it works to satisfy a set of customer defined requirements.

Object-oriented design

Object-oriented design (OOD) transforms the conceptual model produced in object-oriented analysis to take account of the constraints imposed by the chosen architecture and any non-functional – technological or environmental – constraints, such as transaction throughput, response time, run-time platform, development environment, or programming language.

The concepts in the analysis model are mapped onto implementation classes and interfaces. The result is a model of the solution domain, a detailed description of *how* the system is to be built.

Chapter 11

Architecture Description Language

In the system engineering community, an **Architecture Description Language** (**ADL**) is a language and/or conceptual model used to describe and represent system architectures.

In the software engineering community, an **Architecture Description Language** (**ADL**) is a computer language used to describe and represent software architectures. This means in case of technical architecture, the architecture must be communicated to software developers. With functional architecture, the software architecture is communicated with stakeholders and enterprise engineers. By the software engineering community several ADLs have been developed, such as Acme (developed by CMU), AADL (standardized by SAE), C2 (developed by UCI), Darwin (developed by Imperial College London), and Wright (developed by CMU).

The enterprise modelling and engineering community have also developed architecture description languages catered for at the enterprise level. Examples include ArchiMate (now an Open Group standard), DEMO, ABACUS (developed by the University of Technology, Sydney) etc. These languages do not necessarily refer to software components, etc. Most of them, however, refer to an application architecture as the architecture that is communicated to the software engineers.

Most of the writing below refers primarily to the perspective from the software engineering community.

Introduction

A standard notation (ADL) for representing architectures helps promote mutual communication, the embodiment of early design decisions, and the creation of a transferable abstraction of a system. Architectures in the past were largely represented by box-and-line drawing annotated with such things as the nature of the component, properties, semantics of connections, and overall system behavior. ADLs result from a linguistic approach to the formal representation of architectures, and as such they address its shortcomings. Also important, sophisticated ADLs allow for early analysis and feasibility testing of architectural design decisions.

Characteristics

There is a large variety in ADLs developed by either academic or industrial groups. Many languages were not intended to be an ADL, but they turn out to be suitable for representing and analyzing an architecture. In principle ADLs differ from requirements languages, because ADLs are rooted in the solution space, whereas requirements describe problem spaces. They differ from programming languages, because ADLs do not bind architectural abstractions to specific point solutions. Modeling languages represent behaviors, where ADLs focus on representation of components. However, there are domain specific modeling languages (DSMLs) that focus on representation of components.

Minimal requirements

The language must:

- Be suitable for communicating an architecture to all interested parties
- Support the tasks of architecture creation, refinement and validation
- Provide a basis for further implementation, so it must be able to add information to the ADL specification to enable the final system specification to be derived from the ADL
- Provide the ability to represent most of the common architectural styles
- Support analytical capabilities or provide quick generating prototype implementations

ADLs have in common:

- Graphical syntax with often a textual form and a formally defined syntax and semantics
- Features for modeling distributed systems
- Little support for capturing design information, except through general purpose annotation mechanisms
- Ability to represent hierarchical levels of detail including the creation of substructures by instantiating templates

ADLs differ in their ability to:

- Handle real-time constructs, such as deadlines and task priorities, at the architectural level
- Support the specification of different architectural styles. Few handle object oriented class inheritance or dynamic architectures
- Support analysis
- Handle different instantiations of the same architecture, in relation to product line architectures

Positive elements of ADL

- ADLs represent a formal way of representing architecture
- ADLs are intended to be both human and machine readable
- ADLs support describing a system at a higher level than previously possible
- ADLs permit analysis of architectures – completeness, consistency, ambiguity, and performance
- ADLs can support automatic generation of software systems

Negative elements of ADL

- There is not universal agreement on what ADLs should represent, particularly as regards the behavior of the architecture
- Representations currently in use are relatively difficult to parse and are not supported by commercial tools
- Most ADLs tend to be very vertically optimized toward a particular kind of analysis

Common concepts of architecture

The ADL community generally agrees that Software Architecture is a set of components and the connections among them. But there are different kind of architectures like :

Object Connection Architecture

- Configuration consists of the interfaces and connections of an object-oriented system
- Interfaces specify the features that must be provided by modules conforming to an interface
- Connections represented by interfaces together with call graph
- Conformance usually enforced by the programming language
    - Decomposition - associating interfaces with unique modules
    - Interface conformance - static checking of syntactic rules
    - Communication integrity - visibility between modules

Interface Connection Architecture

- Expands the role of interfaces and connections
    - Interfaces specify both “required” and “provided” features
    - Connections are defined between “required” features and “provided” features
- Consists of interfaces, connections and constraints
    - Constraints restrict behavior of interfaces and connections in an architecture
    - Constraints in an architecture map to requirements for a system

Most ADLs implement an interface connection architecture.

Architecture vs. Design

So what is the difference between architecture and design? Architecture casts non-functional decisions and partitions functional requirements, whereas design specifies or derives functional requirements. The process of defining an architecture may use heuristics or iterative improvements; this may require going a level deeper to validate the choices, so the architect often has to do a high-level design to validate the partitioning.

Examples

Below the list gives the candidates for being the best ADL until now

- Primary candidates
    - ACME / ADML (CMU/USC)
    - Rapide (Stanford)
    - Wright (CMU)
    - Unicon (CMU)
    - ByADL (Build Your ADL) - University of L'Aquila
    - LePUS3 and Class-Z (University of Essex)
    - ABACUS (UTS)
- Secondary candidates
    - Aesop (CMU)
    - MetaH (Honeywell)
    - AADL (SAE) - **A**rchitecture **A**nalysis & **D**esign **L**anguage
    - C2 SADL (UCI)
    - SADL (SRI) - **S**ystem **A**rchitecture **D**escription **L**anguage
- Others
    - Lileanna - **L**ibrary **I**nterconnect **L**anguage **E**xtended with **Ann**otated **A**da
    - Dually: Providing Architectural Languages and Tools Interoperability through Model Transformation Technologies

Approaches to architecture

Approaches to Architecture

- Academic Approach
    - focus on analytic evaluation of architectural models
    - individual models
    - rigorous modeling notations
    - powerful analysis techniques
    - depth over breadth
    - special-purpose solutions
- Industrial Approach
    - focus on wide range of development issues

- families of models
- practicality over rigor
- architecture as the big picture in development
- breadth over depth
- general-purpose solutions

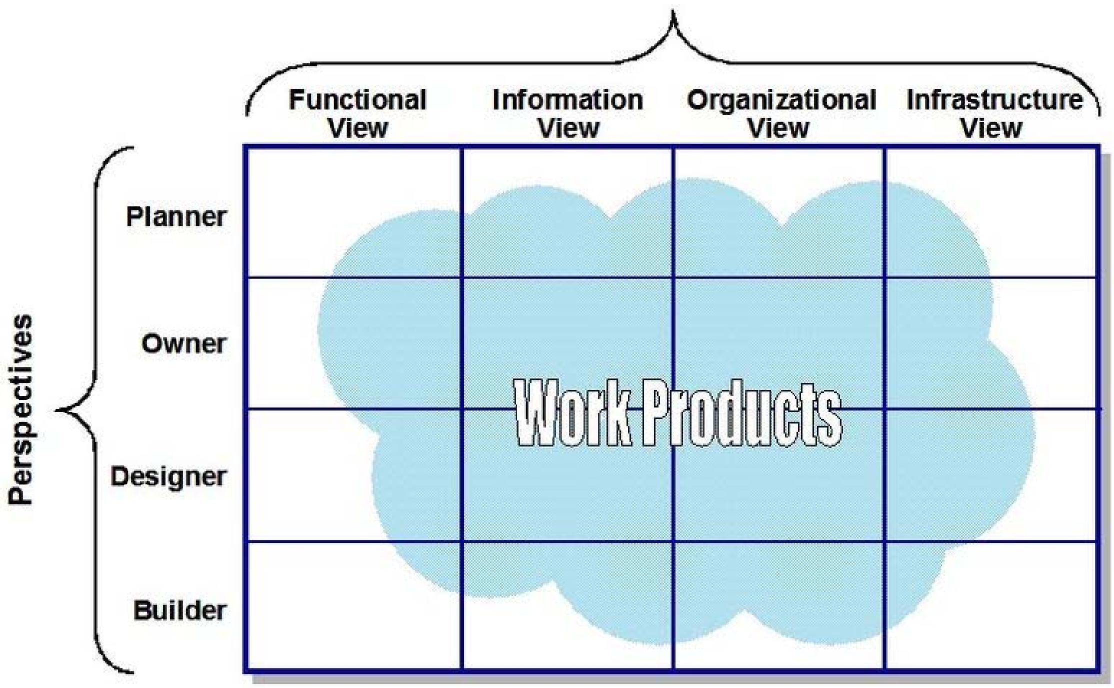
Conclusion

- There is a rich body of research to draw upon
- Much has been learned about representing and analyzing architectures
- Effort is needed now to bring together the common knowledge and put it into practice

Chapter 12

Enterprise Architecture Frameworks

View model



The TEAF Matrix of Views and Perspectives.

A **view model** or *viewpoints framework* in systems engineering, software engineering, and enterprise engineering is a framework which defines a coherent set of *views* to be used in the construction of a system architecture, software architecture, or enterprise architecture. A *view* is a representation of a whole system from the perspective of a related set of concerns. View model provides guidance and rules for structuring, classifying, and organizing architectures.

Since the early 1990s there have been a number of efforts to define standard approaches for describing and analyzing system architectures. Each of the recent Enterprise

Architecture frameworks define a set of views, but these sets are not always called "view models".

Usually one *view* is a very concrete product that presents a specific set of architecture data for a given system. However, the same term is sometimes used to refer to a view *definition*, including the particular viewpoint and the corresponding guidance that defines each concrete view. The term *view model* is related to view definitions.

Overview

The purpose of viewpoints and views is to enable human engineers to comprehend very complex systems, and to organize the elements of the problem and the solution around domains of expertise. In the engineering of physically-intensive systems, viewpoints often correspond to capabilities and responsibilities within the engineering organization.

Most complex system specifications are so extensive that no single individual can fully comprehend all aspects of the specifications. Furthermore, we all have different interests in a given system and different reasons for examining the system's specifications. A business executive will ask different questions of a system make-up than would a system implementer. The concept of viewpoints framework, therefore, is to provide separate viewpoints into the specification of a given complex system in order to facilitate communication with the stakeholders. Each viewpoint satisfies an audience with interest in a particular set of aspects of the system. Each viewpoint may use a specific *viewpoint language* that optimizes the vocabulary and presentation for the audience of that viewpoint. Viewpoint modeling has become an effective approach for dealing with the inherent complexity of large distributed systems.

Current software architectural practices, as described in IEEE Std. 1471, divide the design activity into several areas of concerns, each one focusing on a specific aspect of the system. Examples include the "4+1" view model, the Zachman Framework, TOGAF, DoDAF and, RM-ODP.

History

Since the early 1990s there have been a number of efforts to define standard approaches for describing and analyzing system architectures. Many of these have been funded by the United States Department of Defense, but some have sprung from international or national efforts in ISO or the IEEE. The most relevant of these, the IEEE Recommended Practice for Architectural Description of Software-Intensive Systems (IEEE-Std-1471-2000) provides some very useful definitions and guidelines for what a system architecture is and for the use of viewpoint specifications to address the stakeholder concerns.

The IEEE 1471 specification describes the process for developing architecture descriptions under a number of scenarios, including precedented and unprecedented design, evolutionary design, and capture of design of existing systems. In all of these scenarios the overall process is the same: identify stakeholders, elicit concerns, identify a set of viewpoints to be used, and then apply these viewpoint specifications to develop a set of relevant views of the system. Although IEEE-1471 does mention the possibility of defining system, functional, and technical views (among others), it does not go so far as to define any specific set of views nor what these might be. Eventually in 1996 the ISO Reference Model for Open Distributed Processing (RM-ODP) was published to provide a useful framework for describing the architecture and design of large scale distributed systems.

View model topics

View

A view of a system is a representation of the system from the perspective of a viewpoint. This viewpoint on a system involves a perspective focusing on specific concerns regarding the system, which suppresses details to provide a simplified model having only those elements related to the concerns of the viewpoint. For example, a security viewpoint focuses on security concerns and a security viewpoint model contains those elements that are related to security from a more general model of a system.

A view allows a user to examine a portion of a particular interest area. For example, an Information View may present all functions, organizations, technology, etc. that use a particular piece of information, while the Organizational View may present all functions, technology, and information of concern to a particular organization. In the Zachman Framework views comprise a group of work products whose development requires a particular analytical and technical expertise because they focus on either the "what," "how," "who," "where," "when," or "why" of the enterprise. For example, Functional View work products answer the question "how is the mission carried out?" They are most easily developed by experts in functional decomposition using process and activity modeling. They show the enterprise from the point of view of functions. They also may show organizational and information components, but only as they relate to functions.

Viewpoints

Viewpoint is a systems engineering concept that describes a partitioning of concerns in system restricted to a particular set of concerns. Adoption of a viewpoint is usable so that issues in those aspects can be addressed separately. A good selection of viewpoints also partitions the design of the system into specific areas of expertise.

Viewpoints provide the conventions, rules, and languages for constructing views. A view is a representation of a whole system from the perspective of a point. A view may consist of one or more architectural models. Each such architectural model is developed using the methods established by its associated architectural system, as well as for the system as a whole.

Modeling perspectives

Modeling perspectives is a set of different ways to represent pre-selected aspects of a system. Each perspective has a different focus, conceptualization, dedication and visualization of what the model is representing.

In information systems, the traditional way to distinction between modeling perspectives is structural, functional and behavioral/processual perspectives. This together with rule, object, communication and actor and role perspectives is one way of classifying modeling approaches

Viewpoint model

In any given viewpoint, it is possible to make a model of the system that contains only the objects that are visible from that viewpoint, but also captures all of the objects, relationships and constraints that are present in the system and relevant to that viewpoint. Such a model is said to be a viewpoint model, or a view of the system from that viewpoint.

A given view is a specification for the system at a particular level of abstraction from a given viewpoint. Different levels of abstraction contain different levels of detail. Higher-level views allow the engineer to fashion and comprehend the whole design and identify and resolve problems in the large. Lower-level views allow the engineer to concentrate on a part of the design and develop the detailed specifications.

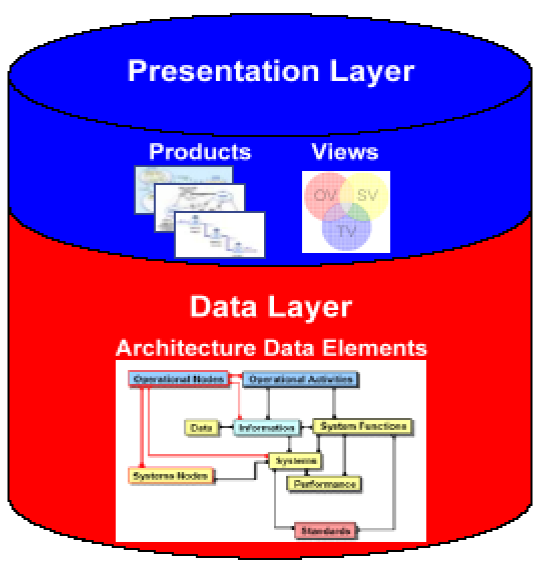


Illustration of the views, products and data in DoDAF Architecture Framework.

In the system itself, however, all of the specifications appearing in the various viewpoint models must be addressed in the realized components of the system. And the specifications for any given component may be drawn from many different viewpoints. On the other hand, the specifications induced by the distribution of functions over specific components and component interactions will typically reflect a different partitioning of concerns than that reflected in the original viewpoints. Thus additional viewpoints, addressing the concerns of the individual components and the bottom-up synthesis of the system, may also be useful.

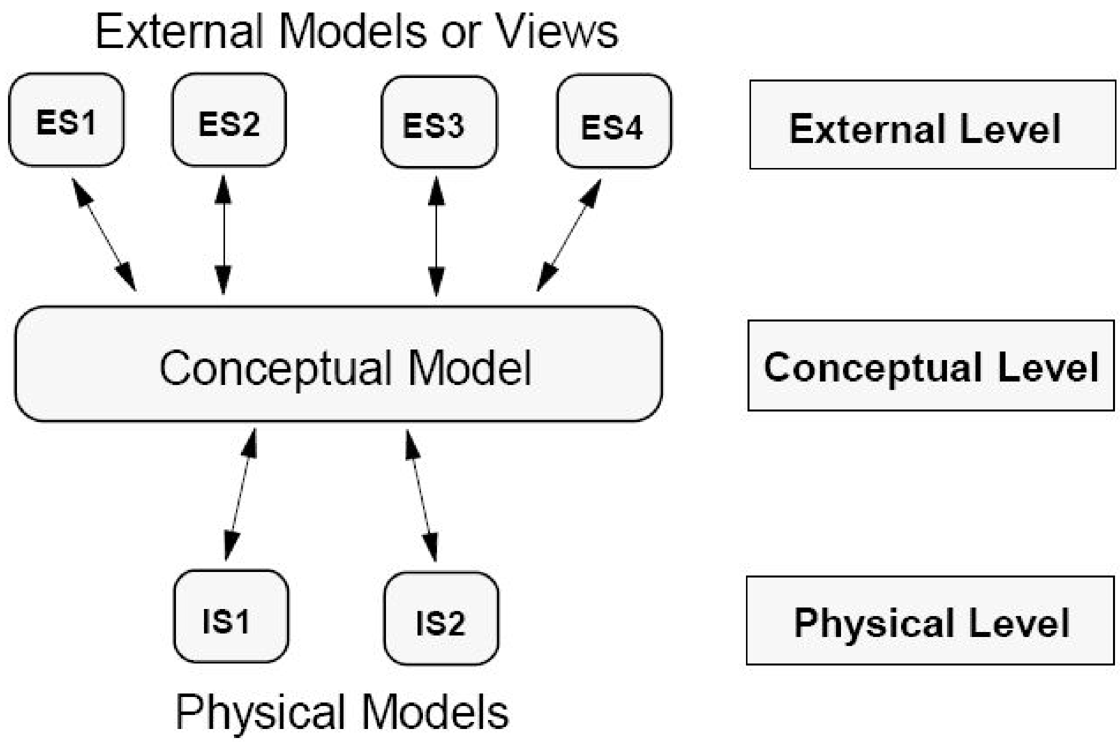
Architecture description

An architecture description is a representation of a system, as of a current or future point in time, in terms of its component parts, how those parts function, the rules and constraints under which those parts function, and how those parts relate to each other and to the environment. In an architecture description the *architecture data* is shared across several views and products.

At the data layer are the architecture data elements and their defining attributes and relationships. At the presentation layer are the products and views that support a visual means to communicate and understand the purpose of the architecture, what it describes, and the various architectural analyses performed. Products provide a way for visualizing architecture data as graphical, tabular, or textual representations. Views provide the ability to visualize architecture data that stem across products, logically organizing the data for a specific or holistic perspective of the architecture.

Types of System View Models

Three schema approach



The notion of a three-schema model was first introduced in 1977 by the ANSI/X3/SPARC three level architecture, which determined three levels to model data.

The Three schema approach for data modeling, introduced in 1977, can be considered one of the first view models. It is an approach to building information systems and systems information management, that promotes the conceptual model as the key to achieving data integration. The Three schema approach defines three schema's and views:

- External schema for user views
- Conceptual schema integrates external schemata
- Internal schema that defines physical storage structures

At the center, the conceptual schema defines the ontology of the concepts as the users think of them and talk about them. The physical schema describes the internal formats of the data stored in the database, and the external schema defines the view of the data presented to the application programs. The framework attempted to permit multiple data models to be used for external schemata.

Over the years, the skill and interest in building information systems has grown tremendously. However, for the most part, the traditional approach to building systems has only focused on defining data from two distinct views, the "user view" and the "computer view". From the user view, which will be referred to as the “external schema,” the definition of data is in the context of reports and screens designed to aid individuals in doing their specific jobs. The required structure of data from a usage view changes with the business environment and the individual preferences of the user. From the computer view, which will be referred to as the “internal schema,” data is defined in terms of file structures for storage and retrieval. The required structure of data for computer storage depends upon the specific computer technology employed and the need for efficient processing of data.

4+1 View Model

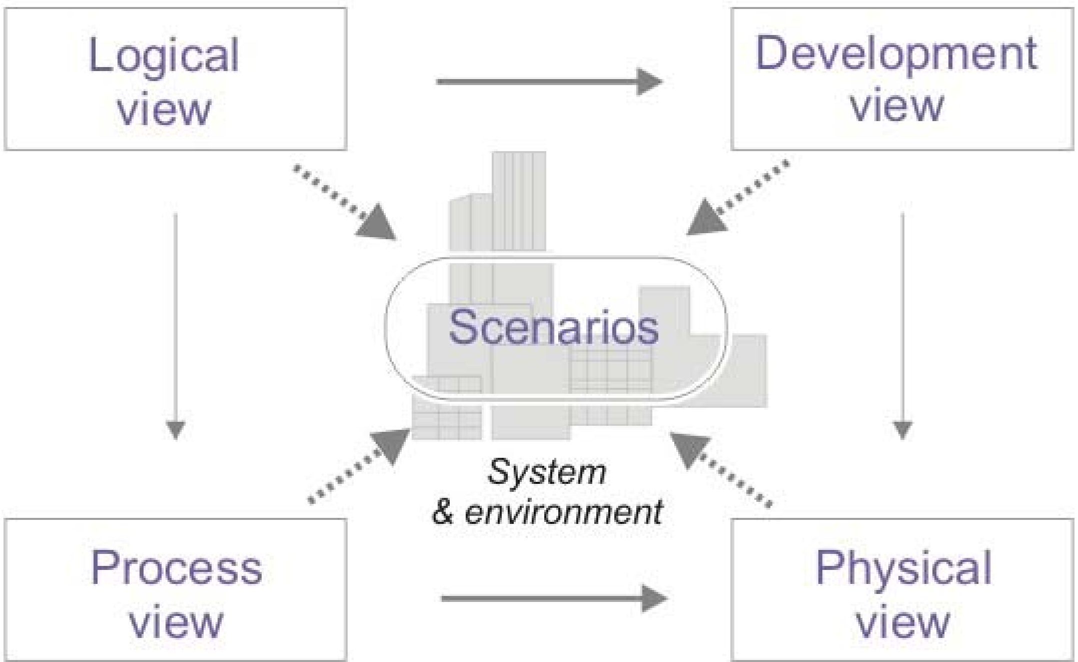


Illustration of the 4+1 Architectural View Model.

4+1 is a view model designed by Philippe Kruchten in 1995 for describing the architecture of software-intensive systems, based on the use of multiple, concurrent views. The views are used to describe the system in the viewpoint of different stakeholders, such as end-users, developers and project managers. The four views of the model are logical, development, process and physical view:

The four views of the model are concerned with :

- *Logical view* : is concerned with the functionality that the system provides to end-users.
- *Development view* : illustrates a system from a programmers perspective and is concerned with software management.

- *Process view* : deals with the dynamic aspect of the system, explains the system processes and how they communicate, and focuses on the runtime behavior of the system.
- *Physical view* : depicts the system from a system engineer's point-of-view. It is concerned with the topology of software components on the physical layer, as well as communication between these components.

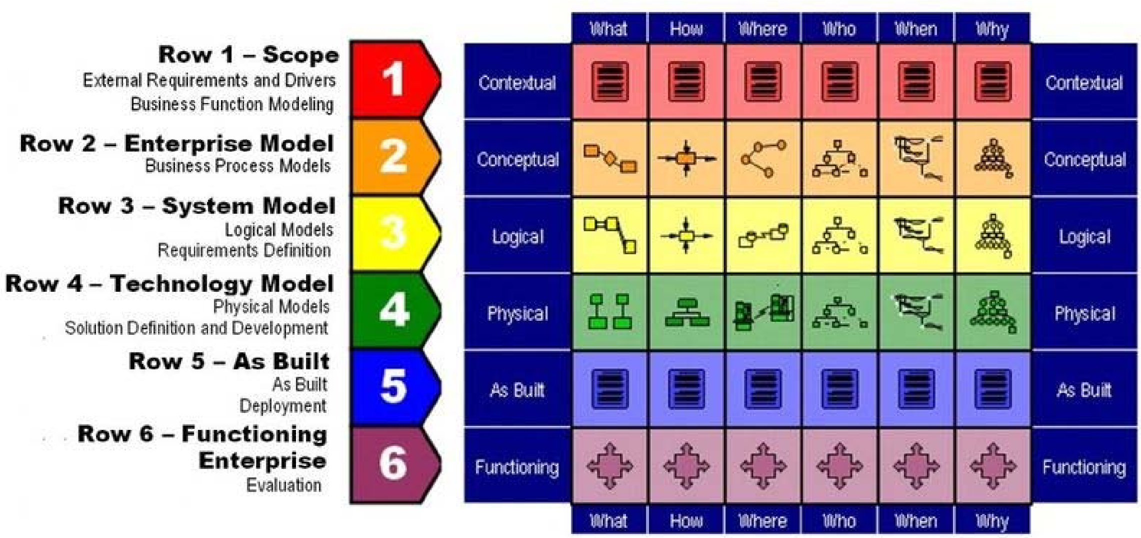
In addition selected use cases or scenarios are utilized to illustrate the architecture. Hence the model contains 4+1 views.

Types of Enterprise Architecture View Models

Enterprise Architecture framework defines how to organize the structure and views associated with an Enterprise Architecture. Because the discipline of Enterprise Architecture and Engineering is so broad, and because enterprises can be large and complex, the models associated with the discipline also tend to be large and complex. To manage this scale and complexity, an Architecture Framework provides tools and methods that can bring the task into focus and allow valuable artifacts to be produced when they are most needed.

Architecture Frameworks are commonly used in Information technology and Information system governance. An organization may wish to mandate that certain models be produced before a system design can be approved. Similarly, they may wish to specify certain views be used in the documentation of procured systems - the U.S. Department of Defense stipulates that specific DoDAF views be provided by equipment suppliers for capital project above a certain value.

Zachman Framework



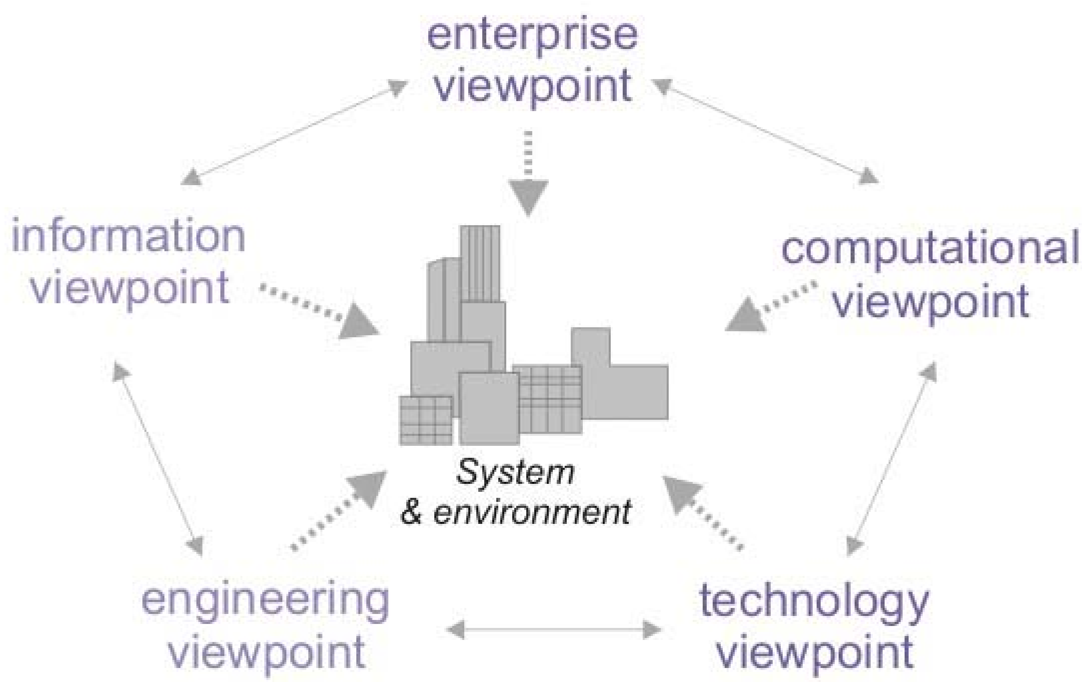
Simplified illustration of the Zachman Framework with an explanation of the rows.

The Zachman Framework, originally conceived by John Zachman at IBM in the 1987, is a framework for enterprise architecture, which provides a formal and highly structured way of viewing and defining an enterprise.

The Framework is used for organizing architectural "artifacts" in a way that takes into account both who the artifact targets (for example, business owner and builder) and what particular issue (for example, data and functionality) is being addressed. These artifacts may include design documents, specifications, and models.

The Zachman Framework is often referenced as a standard approach for expressing the basic elements of enterprise architecture. The Zachman Framework has been recognized by the U.S. Federal Government as having "... received worldwide acceptance as an integrated framework for managing change in enterprises and the systems that support them."

RM-ODP views



The RM-ODP view model, which provides five generic and complementary viewpoints on the system and its environment.

The International Organization for Standardization (ISO) Reference Model for Open Distributed Processing (RM-ODP) specifies a set of viewpoints for partitioning the design of a distributed software/hardware system. Since most integration problems arise in the design of such systems or in very analogous situations, these viewpoints may prove useful in separating integration concerns. The RMODP viewpoints are:

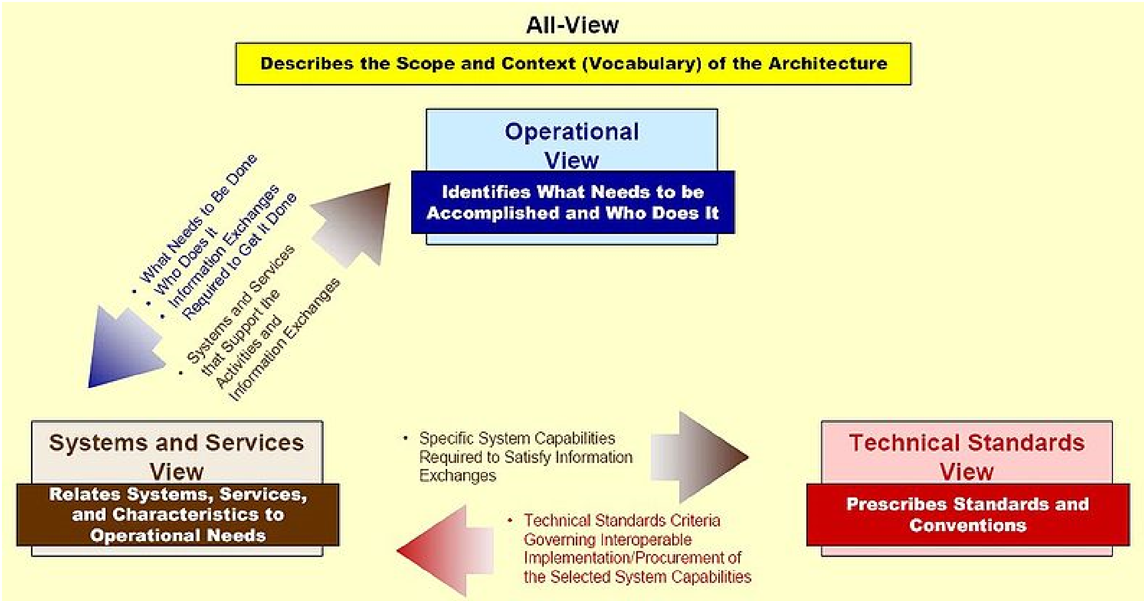
- the *enterprise viewpoint*, which is concerned with the purpose and behaviors of the system as it relates to the business objective and the business processes of the organization
- the *information viewpoint*, which is concerned with the nature of the information handled by the system and constraints on the use and interpretation of that information
- the *computational viewpoint*, which is concerned with the functional decomposition of the system into a set of components that exhibit specific behaviors and interact at interfaces
- the *engineering viewpoint*, which is concerned with the mechanisms and functions required to support the interactions of the computational components
- the *technology viewpoint*, which is concerned with the explicit choice of technologies for the implementation of the system, and particularly for the communications among the components

RMODP further defines a requirement for a design to contain specifications of consistency between viewpoints, including:

- the use of enterprise objects and processes in defining information units
- the use of enterprise objects and behaviors in specifying the behaviors of computational components, and use of the information units in defining computational interfaces
- the association of engineering choices with computational interfaces and behavior requirements
- the satisfaction of information, computational and engineering requirements in the chosen technologies

DoDAF views

The Department of Defense Architecture Framework (DoDAF) defines a standard way to organize an enterprise architecture (EA) or systems architecture into complementary and consistent views. It is especially suited to large systems with complex integration and interoperability challenges, and is apparently unique in its use of "operational views" detailing the external customer's operating domain in which the developing system will operate.



DoDAF linkages among views.

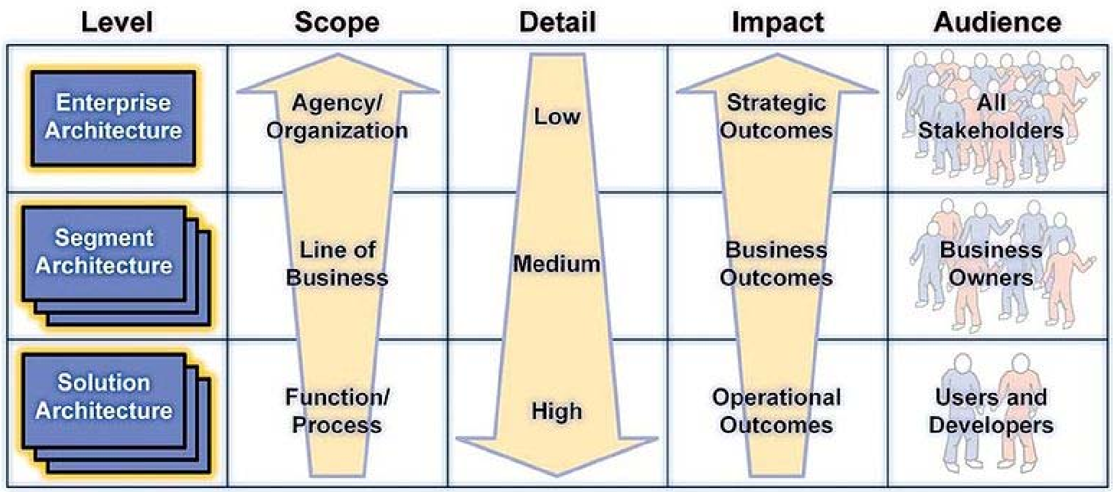
The DoDAF defines a set of products that act as mechanisms for visualizing, understanding, and assimilating the broad scope and complexities of an architecture description through graphic, tabular, or textual means. These products are organized under four views:

- Overarching All View (AV),
- Operational View (OV),
- Systems View (SV), and the
- Technical Standards View (TV).

Each view depicts certain perspectives of an architecture as described below. Only a subset of the full DoDAF viewset is usually created for each system development. The figure represents the information that links the operational view, systems and services view, and technical standards view. The three views and their interrelationships driven – by common architecture data elements – provide the basis for deriving measures such as interoperability or performance, and for measuring the impact of the values of these metrics on operational mission and task effectiveness.

Federal Enterprise Architecture views

In the US Federal Enterprise Architecture enterprise, segment, and solution architecture provide different business perspectives by varying the level of detail and addressing related but distinct concerns. Just as enterprises are themselves hierarchically organized, so are the different views provided by each type of architecture. The Federal Enterprise Architecture Practice Guidance (2006) has defined three types of architecture:



Federal Enterprise Architecture levels and attributes

- Enterprise architecture,
- Segment architecture, and
- Solution architecture.

By definition, Enterprise Architecture (EA) is fundamentally concerned with identifying common or shared assets – whether they are strategies, business processes, investments, data, systems, or technologies. EA is driven by strategy; it helps an agency identify whether its resources are properly aligned to the agency mission and strategic goals and objectives. From an investment perspective, EA is used to drive decisions about the IT investment portfolio as a whole. Consequently, the primary stakeholders of the EA are the senior managers and executives tasked with ensuring the agency fulfills its mission as effectively and efficiently as possible.

By contrast, segment architecture defines a simple roadmap for a core mission area, business service, or enterprise service. Segment architecture is driven by business management and delivers products that improve the delivery of services to citizens and agency staff. From an investment perspective, segment architecture drives decisions for a business case or group of business cases supporting a core mission area or common or shared service. The primary stakeholders for segment architecture are business owners and managers. Segment architecture is related to EA through three principles: structure, reuse, and alignment. First, segment architecture inherits the framework used by the EA, although it may be extended and specialized to meet the specific needs of a core mission area or common or shared service. Second, segment architecture reuses important assets defined at the enterprise level including: data; common business processes and investments; and applications and technologies. Third, segment architecture aligns with elements defined at the enterprise level, such as business strategies, mandates, standards, and performance measures.

Nominal set of views

In search of "Framework for Modeling Space Systems Architectures" Peter Shames and Joseph Skipper (2006) defined a "nominal set of views", Derived from CCSDS RASDS, RM-ODP, ISO 10746 and compliant with IEEE 1471.

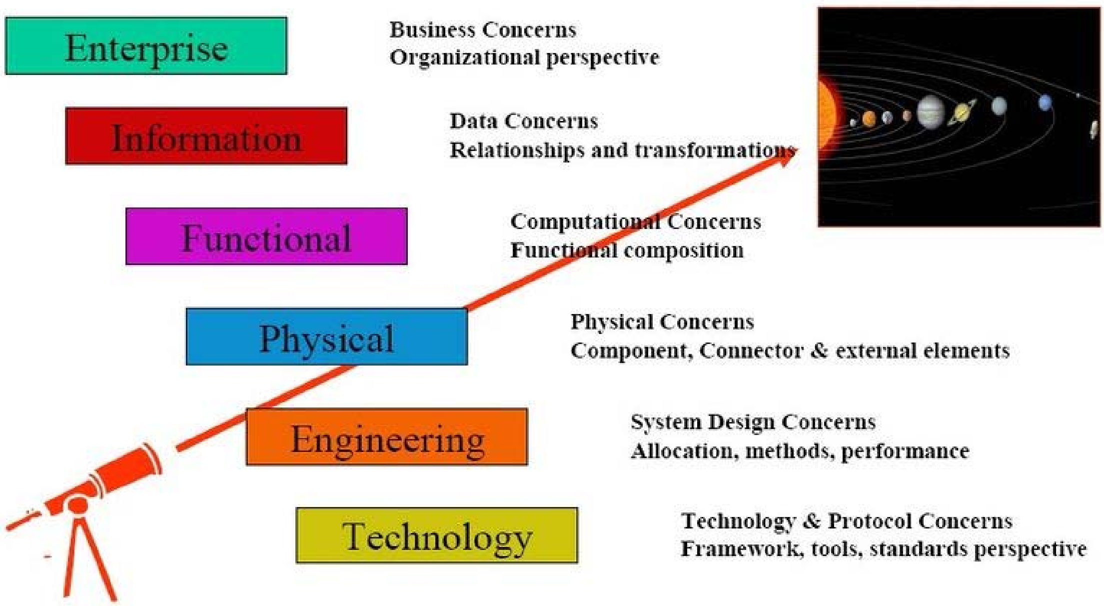


Illustration of the "Nominal set of views".

This "set of views", as described below, is a listing of possible modeling viewpoints. Not all of these views may be used for any one project and other views may be defined as necessary. Note that for some analyses elements from multiple viewpoints may be combined into a new view, possibly using a layered representation.

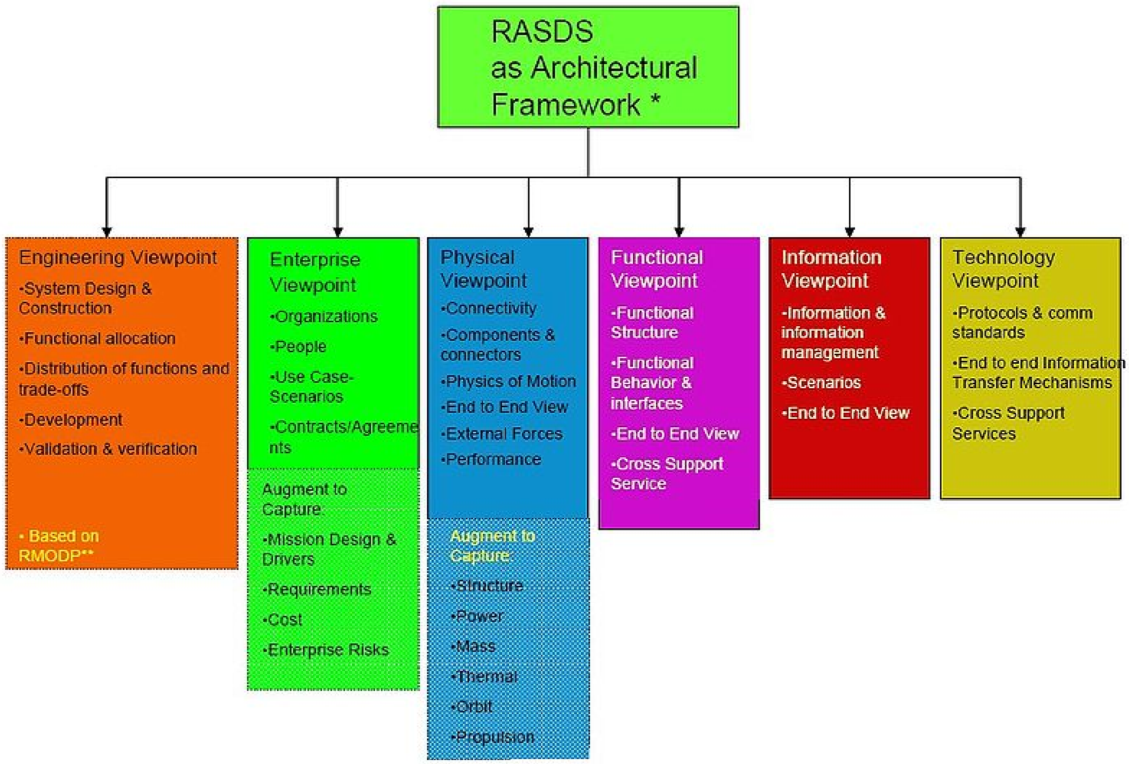
In a latter presentation this nominal set of views was presented as an Extended RASDS Semantic Information Model Derivation. Hereby RASDS stands for Reference Architecture for Space Data Systems.

Enterprise Viewpoint

- Organization view – Includes organizational elements and their structures and relationships. May include agreements, contracts, policies and organizational interactions.
- Requirements view – Describes the requirements, goals, and objectives that drive the system. Says what the system must be able to do.
- Scenario view – Describes the way that the system is intended to be used. Includes user views and descriptions of how the system is expected to behave.

Information viewpoint

- Metamodel view – An abstract view that defines information model elements and their structures and relationships. Defines the classes of data that are created and managed by the system and the data architecture.
- Information view – Describes the actual data and information as it is realized and manipulated within the system. Data elements are defined by the metamodel view and they are referred to by functional objects in other views.



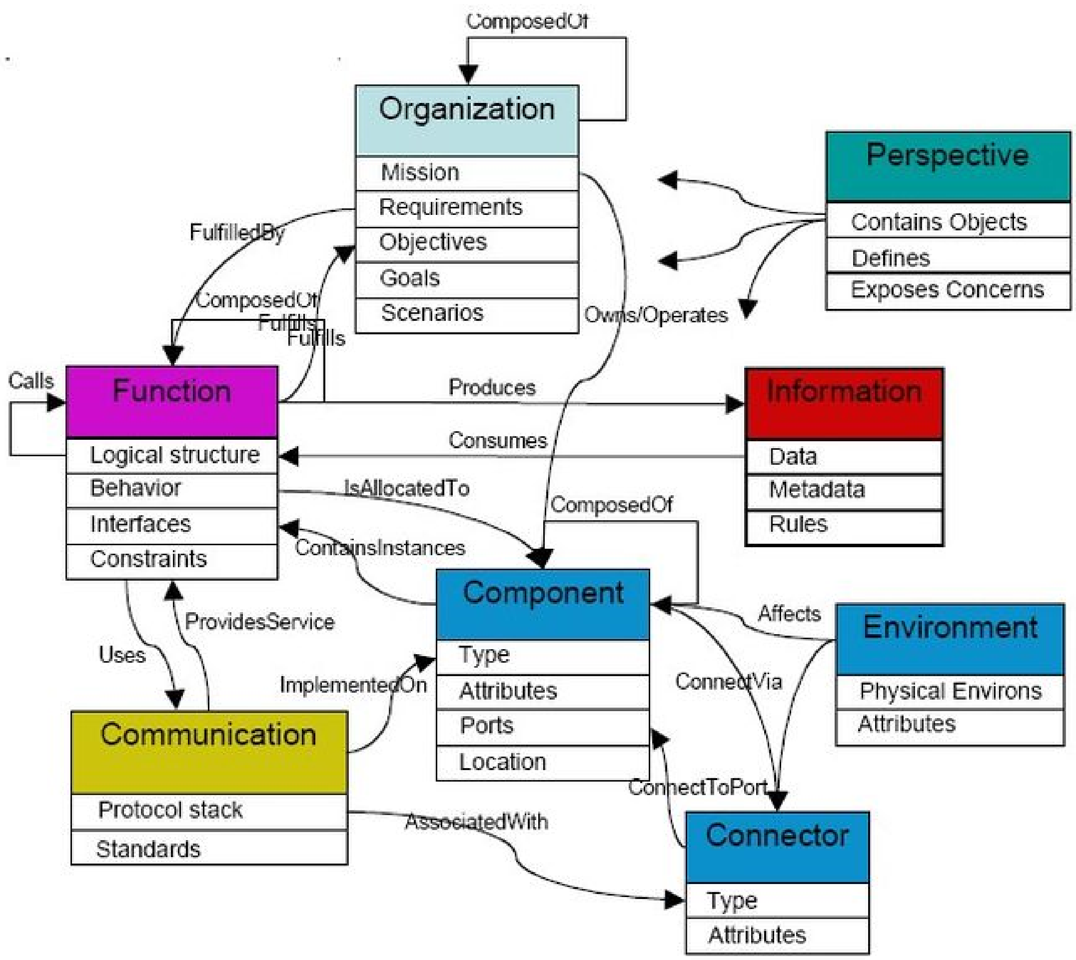
Reference Architecture for Space Data Systems.

Functional viewpoint

- Functional Dataflow view – An abstract view that describes the functional elements in the system, their interactions, behavior, provided services, constraints and data flows among them. Defines which functions the system is capable of performing, regardless of how these functions are actually implemented.
- Functional Control view – Describes the control flows and interactions among functional elements within the system. Includes overall system control interactions, interactions between control elements and sensor / effector elements and management interactions.

Physical viewpoint

- Data System view – Describes instruments, computers, and data storage components, their data system attributes and the communications connectors (busses, networks, point to point links) that are used in the system.
- Telecomm view – Describes the telecomm components (antenna, transceiver), their attributes and their connectors (RF or optical links).
- Navigation view – Describes the motion of the major elements within the system (trajectory, path, orbit), including their interaction with external elements and forces that are outside of the control of the system, but that must be modeled with it to understand system behavior (planets, asteroids, solar pressure, gravity)
- Structural view – Describes the structural components in the system (s/c bus, struts, panels, articulation), their physical attributes and connectors, along with the relevant structural aspects of other components (mass, stiffness, attachment)
- Thermal view – Describes the active and passive thermal components in the system (radiators, coolers, vents) and their connectors (physical and free space radiation) and attributes, along with the thermal properties of other components (i.e. antenna as sun shade)
- Power view – Describes the active and passive power components in the system (solar panels, batteries, RTGs) within the system and their connectors, along with the power properties of other components (data system and propulsion elements as power sinks and structural panels as grounding plane)
- Propulsion view – Describes the active and passive propulsion components in the system (thrusters, gyros, motors, wheels) within the system and their connectors, along with the propulsive properties of other components



MBED Top Level Ontology based on the Nominal set of views.

Engineering viewpoint

- Allocation view – Describes the allocation of functional objects to engineered physical and computational components within the system, permits analysis of performance and used to verify satisfaction of requirements
- Software view - Describes the software engineering aspects of the system, software design and implementation of functionality within software components, select languages and libraries to be used, define APIs, do the engineering of abstract functional objects into tangible software elements. Some functional elements, described using a software language, may actually be implemented as hardware (FPGA, ASIC)
- Hardware views – Describes the hardware engineering aspects of the system, hardware design, selection and implementation of all of the physical components to be assembled into the system. There may be many of these views, each specific to a different engineering discipline.
- Communications Protocol view – Describes the end to end design of the communications protocols and related data transport and data management

services, shows the protocol stacks as they are implemented on each of the physical components of the system.
- Risk view – Describes the risks associated with the system design, processes, and technologies, assigns additional risk assessment attributes to other elements described in the architecture
- Control Engineering view - Analyzes system from the perspective of its controllability, allocation of elements into system under control and control system
- Integration and Test view – Looks at the system from the perspective of what must be done to assemble, integrate and test system and sub-systems, and assemblies. Includes verification of proper functionality, driven by scenarios, in satisfaction of requirements.
- IV&V view – independent validation and verification of functionality and proper operation of the system in satisfaction of requirements. Does system as designed and developed meet goals and objectives.

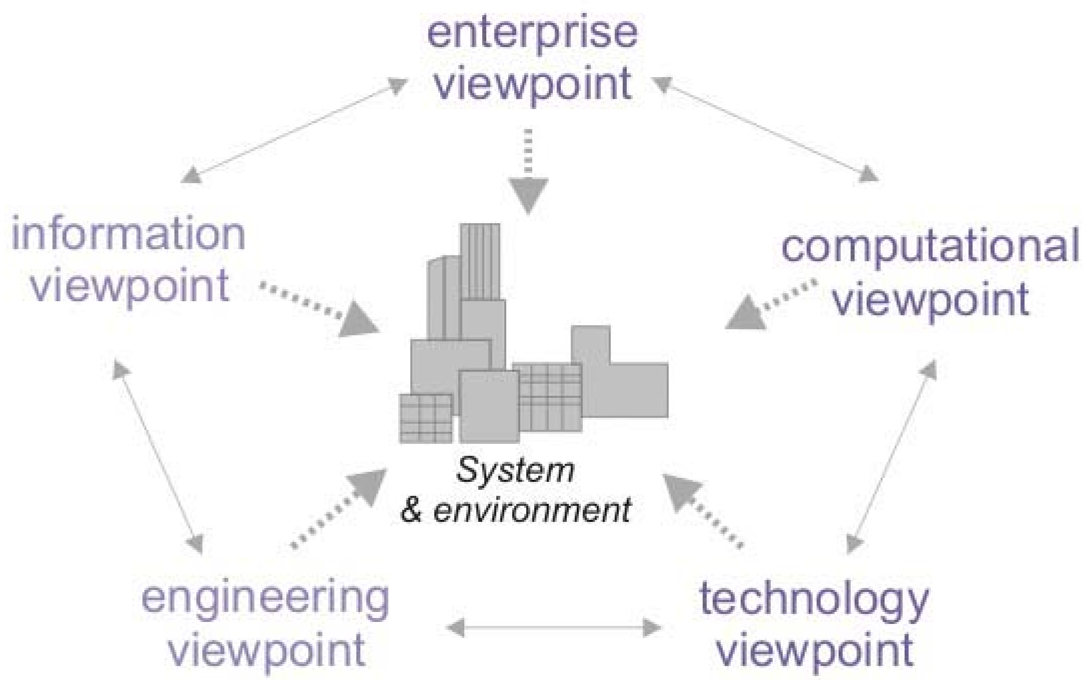
Technology viewpoint

- Standards view – Defines the standards to be adopted during design of the system (e.g. communication protocols, radiation tolerance, soldering). These are essentially constraints on the design and implementation processes.
- Infrastructure view – Defines the infrastructure elements that are to support the engineering, design, and fabrication process. May include data system elements (design repositories, frameworks, tools, networks) and hardware elements (chip fabrication, thermal vacuum facility, machine shop, RF testing lab)
- Technology Development & Assessment view – Includes description of technology development programs designed to produce algorithms or components that may be included in a system development project. Includes evaluation of properties of selected hardware and software components to determine if they are at a sufficient state of maturity to be adopted for the mission being designed.

In contrast to the previous listed view models, this "nominal set of views" lists a whole range of views, possible to develop powerful and extensible approaches for describing a general class of software intensive system architectures.

RM-ODP

Reference Model of Open Distributed Processing (RM-ODP) is a reference model in computer science, which provides a co-ordinating framework for the standardization of open distributed processing (ODP). It supports distribution, interworking, platform and technology independence, and portability, together with an enterprise architecture framework for the specification of ODP systems.



The RM-ODP view model, which provides five generic and complementary viewpoints on the system and its environment.

RM-ODP, also named *ITU-T Rec. X.901-X.904* and *ISO/IEC 10746*, is a joint effort by the International Organization for Standardization (ISO), the International Electrotechnical Commission (IEC) and the Telecommunication Standardization Sector (ITU-T).

Overview

The RM-ODP is a reference model based on precise concepts derived from current distributed processing developments and, as far as possible, on the use of formal description techniques for specification of the architecture. Many RM-ODP concepts, possibly under different names, have been around for a long time and have been rigorously described and explained in exact philosophy (for example, in the works of Mario Bunge) and in systems thinking (for example, in the works of Friedrich Hayek). Some of these concepts -- such as abstraction, composition, and emergence -- have recently been provided with a solid mathematical foundation in category theory.

RM-ODP has four fundamental elements:

- an object modelling approach to system specification;
- the specification of a system in terms of separate but interrelated viewpoint specifications;
- the definition of a system infrastructure providing distribution transparencies for system applications; and

- a framework for assessing system conformance.

The RM-ODP family of recommendations and international standards defines a system of interrelated essential concepts necessary to specify open distributed processing systems and provides a well-developed enterprise architecture framework for structuring the specifications for any large-scale systems including software systems.

History

Much of the preparatory work that led into the adoption of RM-ODP as an ISO standard was carried out by the Advanced Networked Systems Architecture (ANSA) project. This ran from 1984 until 1998 under the leadership of Andrew Herbert (now MD of Microsoft Research in Cambridge), and involved a number of major computing and telecommunication companies. Parts 2 and 3 of the RM-ODP were eventually adopted as ISO standards in 1996. Parts 1 and 4 were adopted in 1998.

RM-ODP Topics

RM-ODP standards

RM-ODP consists of four basic ITU-T Recommendations and ISO/IEC International Standards:

1. Overview : Contains a motivational overview of ODP, giving scoping, justification and explanation of key concepts, and an outline of the ODP architecture. It contains explanatory material on how the RM-ODP is to be interpreted and applied by its users, who may include standard writers and architects of ODP systems.
2. Foundations : Contains the definition of the concepts and analytical framework for normalized description of (arbitrary) distributed processing systems. It introduces the principles of conformance to ODP standards and the way in which they are applied. In only 18 pages, this standard sets the basics of the whole model in a clear, precise and concise way.
3. Architecture : Contains the specification of the required characteristics that qualify distributed processing as open. These are the constraints to which ODP standards must conform. This recommendation also defines RM-ODP viewpoints, subdivisions of the specification of a whole system, established to bring together those particular pieces of information relevant to some particular area of concern.
4. Architectural Semantics : Contains a formalization of the ODP modeling concepts by interpreting many concepts in terms of the constructs of the different standardized formal description techniques.

Viewpoints modeling and the RM-ODP framework

Most complex system specifications are so extensive that no single individual can fully comprehend all aspects of the specifications. Furthermore, we all have different interests in a given system and different reasons for examining the system's specifications. A business executive will ask different questions of a system make-up than would a system implementer. The concept of RM-ODP viewpoints framework, therefore, is to provide separate viewpoints into the specification of a given complex system. These viewpoints each satisfy an audience with interest in a particular set of aspects of the system. Associated with each viewpoint is a viewpoint language that optimizes the vocabulary and presentation for the audience of that viewpoint.

Viewpoint modeling has become an effective approach for dealing with the inherent complexity of large distributed systems. Current software architectural practices, as described in IEEE 1471, divide the design activity into several areas of concerns, each one focusing on a specific aspect of the system. Examples include the "4+1" view model, the Zachman Framework, TOGAF, DoDAF and, of course, RM-ODP.

A viewpoint is a subdivision of the specification of a complete system, established to bring together those particular pieces of information relevant to some particular area of concern during the analysis or design of the system. Although separately specified, the viewpoints are not completely independent; key items in each are identified as related to items in the other viewpoints. Moreover, each viewpoint substantially uses the same foundational concepts (defined in Part 2 of RM-ODP). However, the viewpoints are sufficiently independent to simplify reasoning about the complete specification. The mutual consistency among the viewpoints is ensured by the architecture defined by RM-ODP, and the use of a common object model provides the glue that binds them all together.

More specifically, the RM-ODP framework provides five generic and complementary viewpoints on the system and its environment:

- The *enterprise viewpoint*, which focuses on the purpose, scope and policies for the system. It describes the business requirements and how to meet them.
- The *information viewpoint*, which focuses on the semantics of the information and the information processing performed. It describes the information managed by the system and the structure and content type of the supporting data.
- The *computational viewpoint*, which enables distribution through functional decomposition on the system into objects which interact at interfaces. It describes the functionality provided by the system and its functional decomposition.
- The *engineering viewpoint*, which focuses on the mechanisms and functions required to support distributed interactions between objects in the system. It describes the distribution of processing performed by the system to manage the information and provide the functionality.
- The *technology viewpoint*, which focuses on the choice of technology of the system. It describes the technologies chosen to provide the processing, functionality and presentation of information.

RM-ODP and UML

Currently there is growing interest in the use of UML for system modelling. However, there is no widely agreed approach to the structuring of such specifications. This adds to the cost of adopting the use of UML for system specification, hampers communication between system developers and makes it difficult to relate or merge system specifications where there is a need to integrate IT systems.

Although the ODP reference model provides abstract languages for the relevant concepts, it does not prescribe particular notations to be used in the individual viewpoints. The viewpoint languages defined in the reference model are abstract languages in the sense that they define what concepts should be used, not how they should be represented. This lack of precise notations for expressing the different models involved in a multi-viewpoint specification of a system is a common feature for most enterprise architectural approaches, including the Zachman Framework, the "4+1" model, or the RM-ODP. These approaches were consciously defined in a notation- and representation-neutral manner to increase their use and flexibility. However, this makes more difficult, among other things, the development of industrial tools for modeling the viewpoint specifications, the formal analysis of the specifications produced, and the possible derivation of implementations from the system specifications.

In order to address these issues, ISO/IEC and the ITU-T started a joint project in 2004: "ITU-T Rec. X.906|ISO/IEC 19793: Information technology - Open distributed processing - Use of UML for ODP system specifications". This document (usually referred to as UML4ODP) defines use of the Unified Modeling Language 2 (UML 2; ISO/IEC 19505), for expressing the specifications of open distributed systems in terms of the viewpoint specifications defined by the RM-ODP.

It defines a set of UML Profiles, one for each viewpoint language and one to express the correspondences between viewpoints, and an approach for structuring them according to the RM-ODP principles. The purpose of "UML4ODP" to allow ODP modelers to use the UML notation for expressing their ODP specifications in a standard graphical way; to allow UML modelers to use the RM-ODP concepts and mechanisms to structure their large UML system specifications according to a mature and standard proposal; and to allow UML tools to be used to process viewpoint specifications, thus facilitating the software design process and the enterprise architecture specification of large software systems.

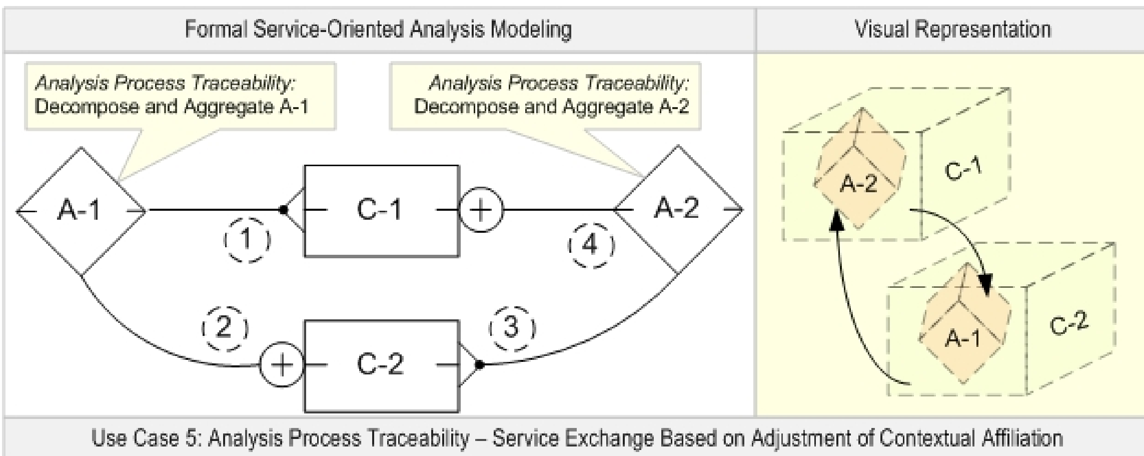
In addition, ITU-T Rec. X.906 | ISO/IEC 19793 enables the seamless integration of the RM-ODP enterprise architecture framework with the Model-Driven Architecture (MDA initiative from the OMG, and with the service-oriented architecture (SOA).

Applications

In addition, there are several projects that have used or currently use RM-ODP for effectively structuring their systems specifications:

- The COMBINE project
- The Reference Architecture for Space Data Systems (RASDS) From the Consultative Committee for Space Data Systems.
- Interoperability Technology Association for Information Processing (INTAP), Japan.
- The Synapses European project.

Service-oriented modeling



Example of a Service-Oriented Modeling Framework (SOMF) Diagram.

Service-oriented modeling is the discipline of modeling business and software systems, for the purpose of designing and specifying service-oriented business systems within a service-oriented architecture.

Any service-oriented modeling methodology typically includes a modeling language that can be employed by both the 'problem domain organization' (the Business), and 'solution domain organization' (the Information Technology Department), whose unique perspectives typically influence the 'service' development life-cycle strategy and the projects implemented using that strategy.

Service-oriented modeling typically strives to create models that provide a comprehensive view of the analysis, design, and architecture of all 'Software Entities' in an organization, which can be understood by individuals with diverse levels of business and technical understanding. Service-oriented modeling typically encourages viewing software entities as 'assets' (service-oriented assets), and refers to these assets collectively as 'services'.

Popular approaches to service-oriented modeling

There are many different approaches that have been proposed for service modeling, including SOMA and SOMF.

Service-oriented modeling and architecture (SOMA)

IBM announced Service-Oriented Modeling and Architecture (SOMA) as the first publicly announced SOA-related methodology in 2004. SOMA refers to the more general domain of service modeling necessary to design and create SOA. SOMA covers a broader scope and implements service-oriented analysis and design (SOAD) through the identification, specification and realization of services, components that realize those services (a.k.a. "service components"), and flows that can be used to compose services.

SOMA includes an analysis and design method that extends traditional object-oriented and component-based analysis and design methods to include concerns relevant to and supporting SOA. It consists of three major phases of identification, specification and realization of the three main elements of SOA, namely, services, components that realize those services (aka service components) and flows that can be used to compose services.

SOMA is an end-to-end SOA Method for the identification, specification, realization and implementation of services (including information services), components, flows (processes/composition). SOMA builds on current techniques in areas such as domain analysis, functional areas grouping, variability-oriented analysis (VOA) process modeling, component-based development, object-oriented analysis and design and use case modeling. SOMA introduces new techniques such as goal-service modeling, service model creation and a service litmus test to help determine the granularity of a service.

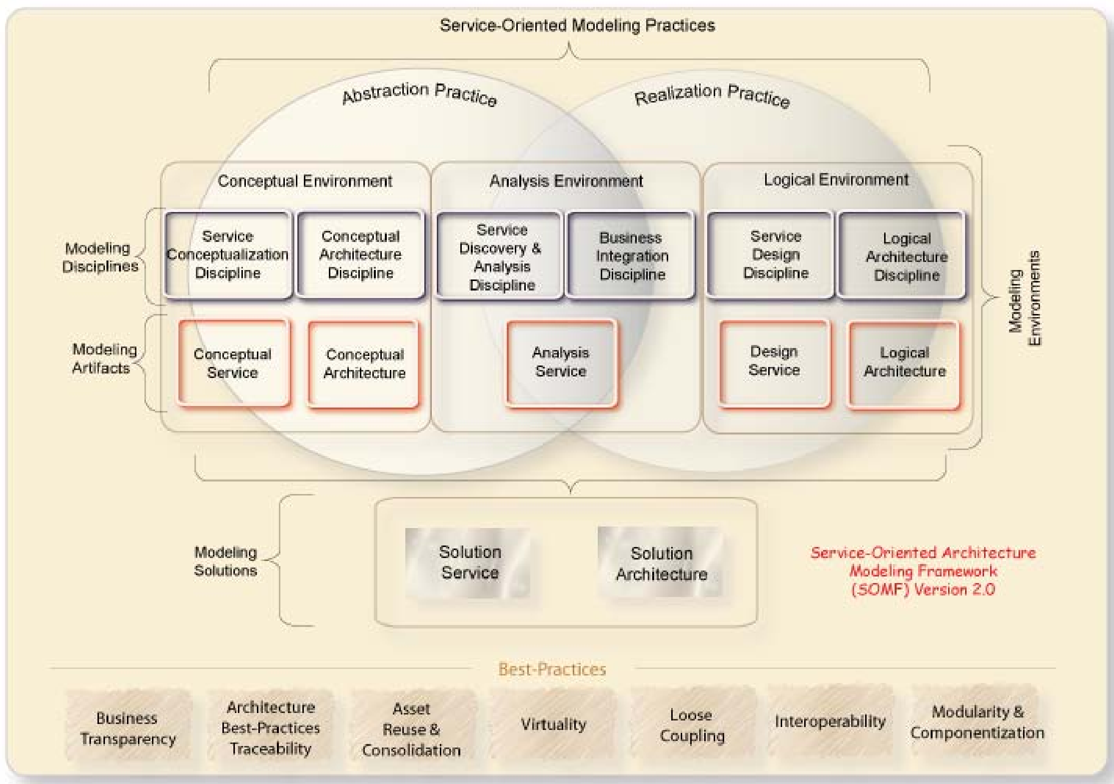
SOMA identifies services, component boundaries, flows, compositions, and information through complementary techniques which include domain decomposition, goal-service modeling and existing asset analysis.

SOMA Life cycle modeling activities

Service-oriented Modeling and Architecture (SOMA) consists of the phases of identification, specification, realization, implementation, deployment and management in which the fundamental building blocks of SOA are identified then refined and implemented in each phase. The fundamental building blocks of SOA consists of services, components, flows and related to them, information, policy and contracts.

Service-oriented modeling framework (SOMF)

The Service-Oriented Modeling Framework (SOMF) has been proposed by author Michael Bell as a holistic and anthropomorphic modeling language for software development that employs disciplines and a universal language to provide tactical and strategic solutions to enterprise problems.The term "holistic language" pertains to a modeling language that can be employed to design any application, business and technological environment, either local or distributed. This universality may include design of application-level and enterprise-level solutions, including SOA landscapes or Cloud Computing environments. The term "anthropomorphic", on the other hand, affiliates the SOMF language with intuitiveness of implementation and simplicity of usage. Furthermore, The SOMF language and its notation has been adopted by Sparx Enterprise Architect modeling platform that enables business architects, technical architects, managers, modelers, developers, and business and technical analysts to pursue the chief SOMF life cycle disciplines.



Service-oriented modeling framework (SOMF) Version 2.0

The service-oriented modeling framework (SOMF) is a service-oriented development life cycle methodology. It offers a number of modeling practices and disciplines that contribute to a successful service-oriented life cycle management and modeling (see image above).

It illustrates the major elements that identify the “what to do” aspects of a service development scheme. These are the modeling pillars that will enable practitioners to craft an effective project plan and to identify the milestones of a service-oriented initiative—either a small or large-scale business or a technological venture.

The provided image thumb (on the right hand side) depicts the four sections of the modeling framework that identify the general direction and the corresponding units of work that make up a service-oriented modeling strategy: practices, environments, disciplines, and artifacts. Remember, these elements uncover the context of a modeling occupation and do not necessarily describe the process or the sequence of activities needed to fulfill modeling goals. These should be ironed out during the project plan – the service-oriented development life cycle strategy – that typically sets initiative boundaries, time frame, responsibilities and accountabilities, and achievable project milestones.

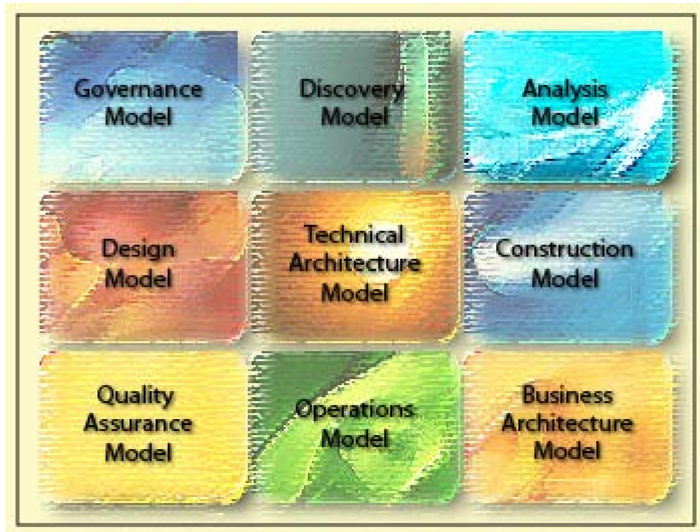
SOMF Language modeling generations

SOMF introduces a transparency model by enabling three major modeling time frames, often named modeling generations:

- Used-to-Be: Design scheme of software components and related environments that were deployed, configured, and used in the past
- As-Is: Design of software components and corresponding environments that are currently being utilized
- To-Be: Design of software components and corresponding environments that will be deployed, configured, and used in the future

These three unique implementation generations can be viewed by SOMF diagrams and their corresponding perspectives to help practitioners to depict business and architectural decisions in the past, current, and future implementations. For example, an architect and a developer can describe the evolution of a system or an application since inception, and explain what were the architecture best practices that drove alterations to these software entities. This capability enables transparency of design and implementation. On the business side, modeling generations can help estimate return on investments and business value. Traceability of business investments and justifications to business initiatives can also be depicted by employing these modeling generations.

SOMF Transformation Models



SOMF Transformation Models

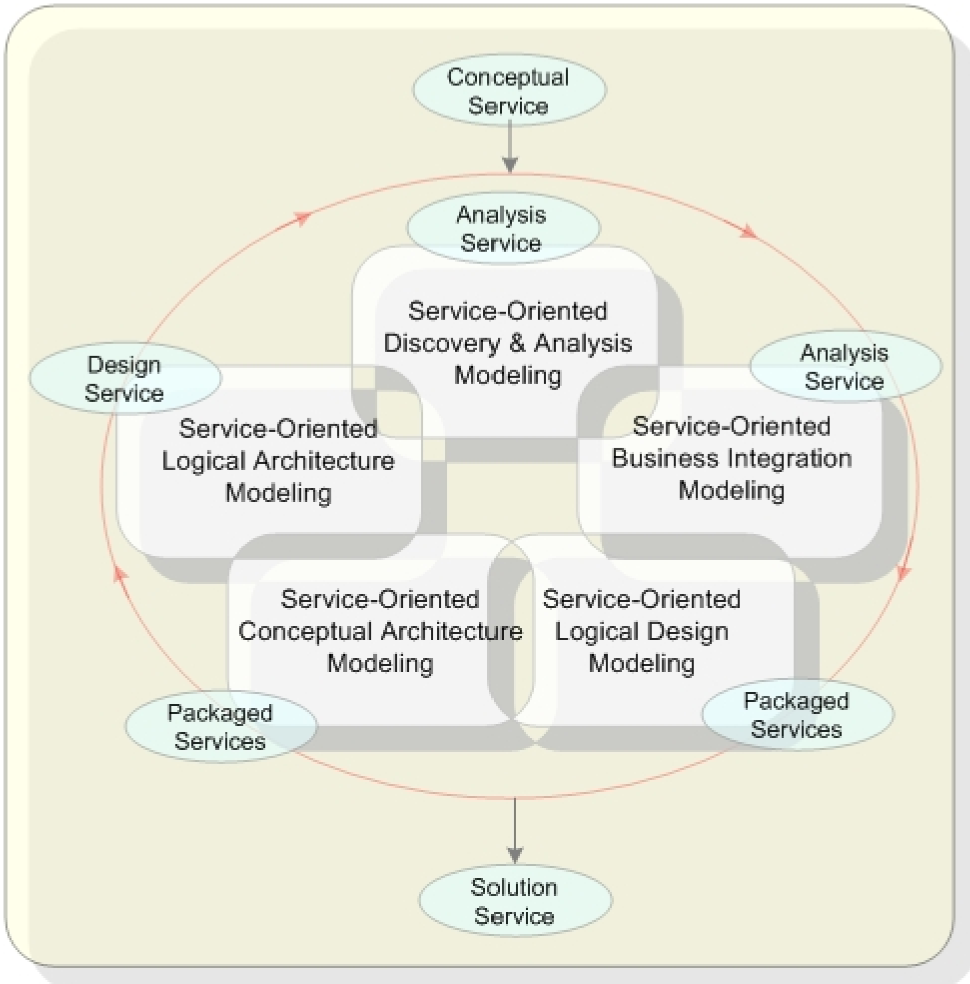
SOMF offers eight models of implementation, also known as "Bell's Transformation Models", as depicted in the displayed image named SOMF Transformation Models. Each of these units of work, namely models, identify the methodology, process, platform, best practices, and disciplines by which a practitioner ought to accomplish a modeling task during a project. The illustrated ninth model is the Governance Model, which should be employed to manage the other eight models.

Consider the overall charter of the SOMF implementation models:

- Discovery Model: This model should be employed when ascertaining new software entities to provide a solution
- Analysis Model: The Analysis Model is devised to inspect a software component's feasibility to offer a solution, help analyzing business and technical requirements, and assist with measuring the success of implementation
- Design Model: Facilitates logical design of software entities; and contributes to component relationship, deployment compositions, and establishment of transactions
- Technical Architecture Model: This model involves three major architecture perspectives: Conceptual Architecture, Logical Architecture, and Physical Architecture
- Construction Model: Assists with modeling practices during the source code implementation phase
- Quality Assurance Model: Certifies software components for production and ensures stability of business and technical continuity
- Operations Model: Enables a stable production environment and assures proper deployment and configuration of software entities
- Business Architecture Model: This model fosters proper integration of contextual and structural business formations with software entities

- Governance Model: Offers best practices, standards, and policies for all SOMF implementation models

SOMF Life cycle modeling activities



SOMF Life Cycle Activities

SOMF is driven by the development process of services. This approach enables business and information technology professionals to focus on deliverables that correspond to a specific service-oriented life cycle stage and event.

The service-oriented modeling framework (SOMF) introduces five major life cycle modeling activities that drive a service evolution during design-time and run-time. At the design-time phase a service originates as a conceptual entity (conceptual service), later it transforms into a unit of analysis (analysis service), next it transitions into a contractual and logical entity (design service), and finally is established as a concrete service (solution service). The following identify the major contributions of the service-oriented modeling activities:

- Service-oriented discovery & analysis modeling: Discover and analyze services for granularity, reusability, interoperability, loose-coupling, and identify consolidation opportunities.
- Service-oriented business integration modeling: Identify service integration and alignment opportunities with business domains' processes (organizations, products, geographical locations)
- Service-oriented logical design modeling: Establish service relationships and message exchange paths. Address service visibility. Craft service logical compositions. Model service transactions
- Service-oriented conceptual architecture modeling: Establish an application or enterprise architectural direction. Depict a technological environment. Craft a technological stack. Identify business ownership.
- Service-oriented logical architecture modeling: Integrate organizational software assets. Establish logical environment dependencies. Foster service reuse, loose coupling and consolidation.

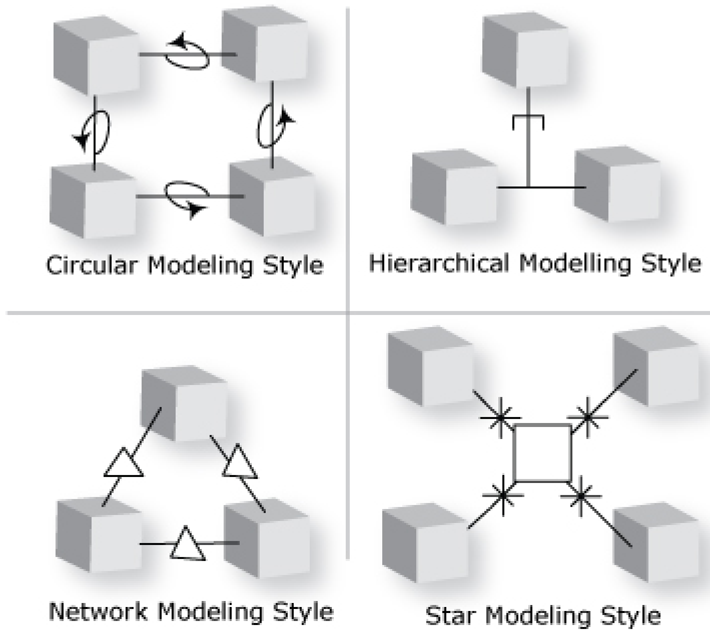
Methodological issues

SOMF Modeling styles

How can a practitioner model a computing environment? In what type of forms can a group of services be arranged to enable an efficient integrated computing landscape? What would be the best message routes between a service consumer and provider? How can interoperability hurdles be mitigated?

SOMF provides four major software modeling styles that are useful throughout a service life cycle (conceptualization, discovery and analysis, business integration, logical design, conceptual and logical architecture). These modeling styles: Circular, Hierarchical, Network, and Star, can assist a software modeler with the following modeling aspects:

- Identify service relationships: contextual and technological affiliations
- Establish message routes between consumers and services
- Provide efficient service orchestration and choreography methods
- Create powerful service transaction and behavioral patterns
- Offer valuable service packaging solutions



SOMF Modeling Styles

In the illustration on the right you will find the major four service-oriented modeling styles that SOMF offers. Each pattern identifies the various approaches and strategies that one should consider employing when modeling a service ecosystem.

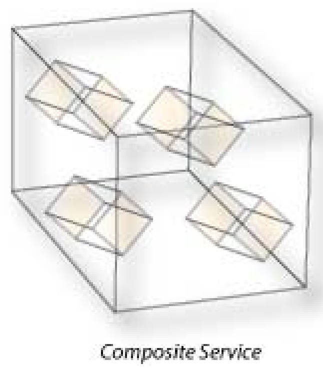
- *Circular Modeling Style*: enables message exchange in a circular fashion, rather than employing a controller to carry out the distribution of messages. The Circular Style also offers a conceptual method to affiliating services.
- *Hierarchical Modeling Style*: offers a relationship pattern between services for the purpose of establishing transactions and message exchange routes between consumers and services. The Hierarchical pattern founds parent/child associations between services.
- *Network Modeling Style*: this pattern establishes “many to many” relationship between services, their peer services, and consumers. The Network pattern accentuates on distributed environments and interoperable computing networks.
- *Star Modeling Style*: the Star pattern advocates arranging services in a star formation, in which the central service passes messages to its extending arms. The Star modeling style is often used in “multi casting” or “publish and subscribe” instances, where “solicitation” or “fire and forget” message styles are involved.

SOMF Modeling assets

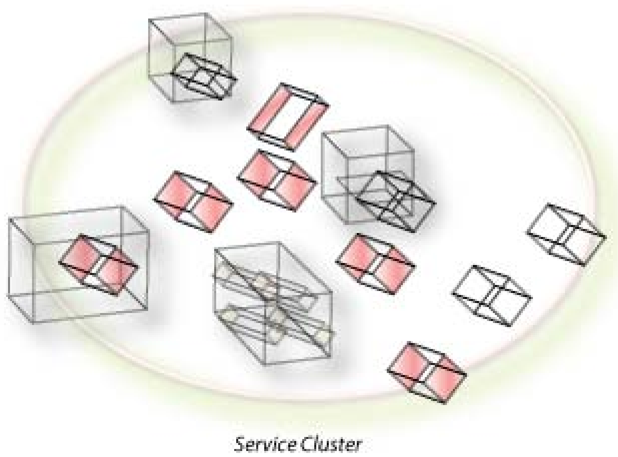
The service-oriented modeling framework (SOMF) introduces three major software formations. These structures are software entities that habitually exist in our computing environments. In addition, the notion of a software component is further abstracted and represented by the universal "service" term, which can represent any organizational software asset, such as an object, a software module, a library component, an application, a business process, a database, a database store procedure or trigger, an ESB, a legacy implementation, a Web service, and more. Again, any of these software entities can be named "service".

Thus, a service is classified by its contextual and structural attributes:

Atomic Service



Composite Service

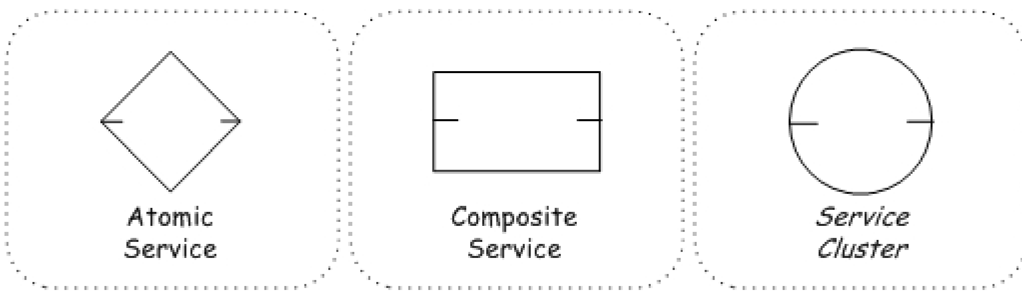


Service Cluster

- *Atomic service*: an indivisible software component that is too granular and executes fewer business or technical functionalities. An atomic formation is also a piece of software that typically is not subject to decomposition analysis activities and its business or technological functionality does not justify breakdown to smaller components. Examples: customer address service and checking account balance service.

- *Composite service*: a composite service structure aggregates smaller and fine-grained services. This hierarchical service formation is characteristically known as coarse-grained entity that encompasses more business or technical processes. A composite service may aggregate atomic or other composite services. Examples: customer checking service that aggregates smaller checking account and savings account services. An application that is composed of sub-systems, an ESB that is composed of routing, orchestration, and data transformation components.

- *Service cluster*: this is a collection of distributed and related services that are gathered because of their mutual business or technological commonalities. A service cluster both affiliates services and combines their offerings to solve a business problem. A cluster structure can aggregate atomic as well as composite formations. Examples: A Mutual Funds service cluster that is composed of related and distributed mutual funds services. A Customer Support service cluster that offers automated bill payments, online account statements, and money transfer capabilities



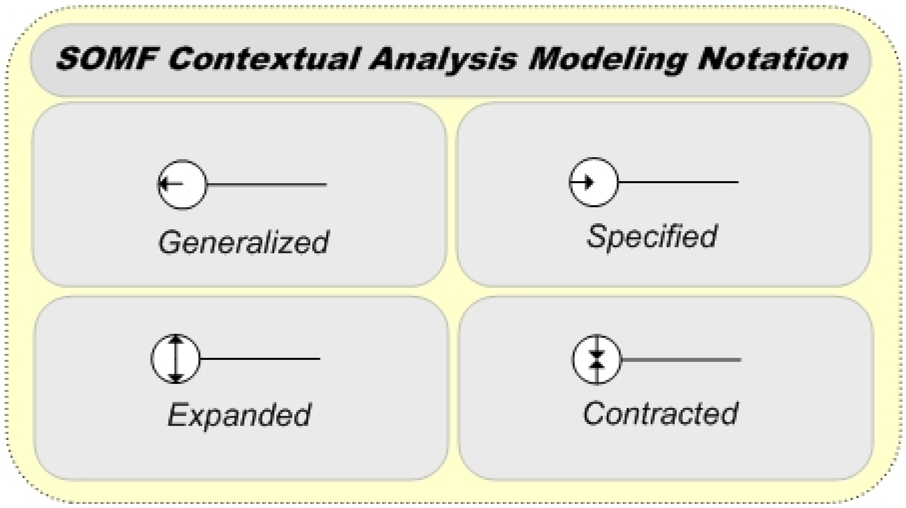
The SOMF also introduces a simple notation to describe the three major service formations, as illustrated in the below image: Atomic Service, Composite Service, and Service Cluster.

SOMF Modeling notation

As previously discussed, each SOMF life cycle discipline also offers a specialized notation. For example, the service-oriented discovery and analysis discipline provides a notation to help model the analysis and identification of services. In addition, during the design phase the SOMF design notation offers symbols to help model a logical design, design composition, and a service transaction model.

Let us take a look at the service-oriented discovery and analysis modeling notation. During the service identification and inspection a practitioner should pursue two types of modeling tasks: (1) Contextual analysis and modeling, and (2) Structural analysis and modeling. These activities are performed to produce a service-oriented analysis proposition.

The below illustration identifies the contextual analysis and modeling operations (represented by analysis symbols)that can be employed to draft an analysis proposition diagram. These operations promote the core service-oriented analysis discipline best practices.

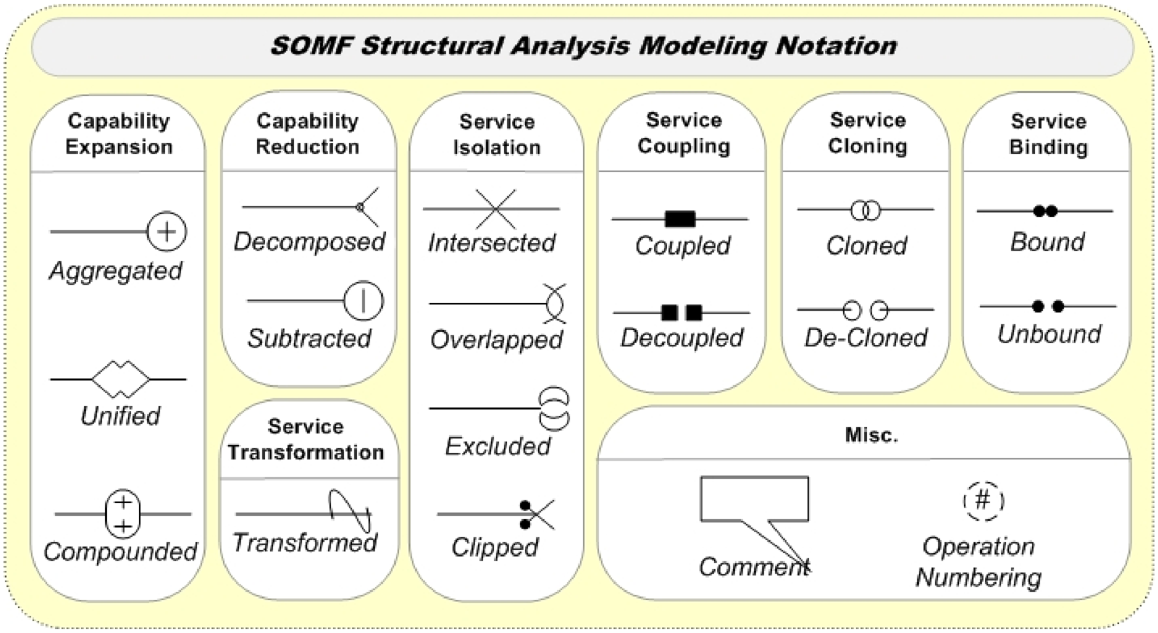


Service-oriented contextual analysis notation - SOMF 2.0

Here is a short description for these symbols:

- *Generalized:* Increases service abstraction level and broadens service offerings
- *Specified:* Decreases service abstraction level and limits service offerings
- *Expanded:* Expands service operations in a distributed environment
- *Contracted:* Trims service operations in a distributed environment

The below illustration, on the other hand, depicts the service-oriented structural analysis and modeling notation.



Service-oriented tructural analysis notation - SOMF 2.0

Here is a short description for these symbols:

- *Aggregated:* Depicts containment of services
- *Unified:* Joins services by creating a new service
- *Compounded:* Groups services that offer collaborative solution
- *Decomposed:* Detaches a child service from its containing parent
- *Subtracted:* Retires a service
- *Transformed:* Converts a service structure to another formation (i.e from Atomic to Composite, etc,)
- *Intersected:* Intersects two or more service clusters
- *Overlapped:* Identifies the overlapping region between two or more service clusters
- *Excluded:* Isolates the overlapping region of a two ore more intersected service clusters
- *Clipped:* Isolates a service from a distributed environment
- *Coupled:* Structurally couples two autonomous services in a distributed environment
- *Decoupled:* Structurally separates two autonomous services in a distributed environment
- *Cloned:* Duplicates an instance of a service by creating a new and identical service
- *De-cloned:* Separates cloned services
- *Bound:* Identifies a contract between two services
- *Unbound:* Identifies a contract cancellation between two services
- *Operation Numbering:* Illustrates the sequence of analysis and modeling operations

- *Comment:* A place to put comments next to each asset or operation

Service-Oriented Conceptualization

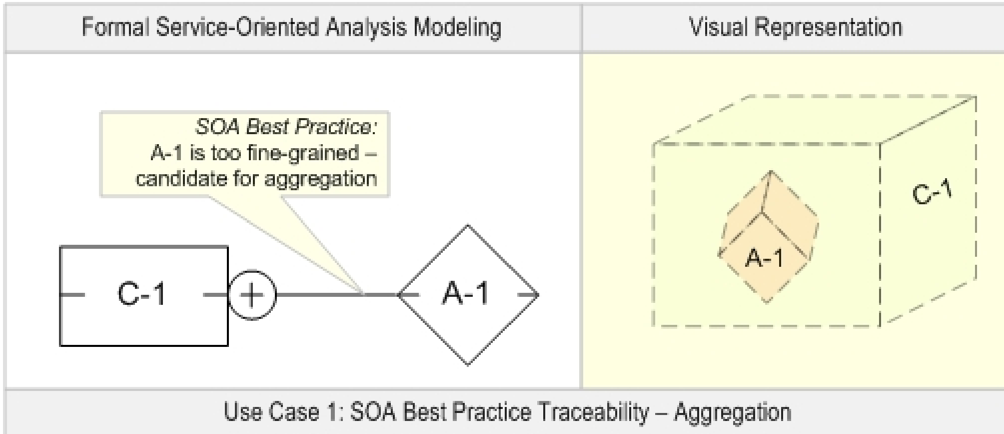
The service-oriented modeling framework (SOMF) advocates that practitioners devise conceptual services to bridge the communication gaps in an organization. These conceptual entities foster the creation of a common language, a vocabulary that can be used during projects to encourage asset reuse and consolidation. One of the most important aspects of the conceptual service paradigm is that it provides direction and defines the scope of a project or a business initiative.

The conceptualization process then identifies six major “tools” that can facilitate the development of conceptual services.

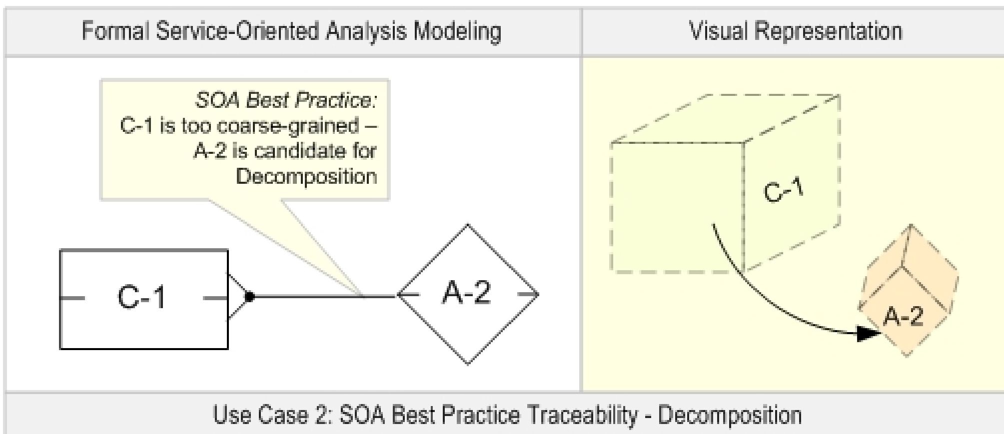
- Concept Attribution: this activity pertains to the collection of software products attributes that both describe service’s features and lead to the discovery of organizational taxonomy
- Concept Classification: the categorization effort contributes to separation of concerns and the establishment of service identities. This is the process of identifying service dissimilarities
- Concept Association: Unlike the classification activity, the association effort enables the discovery of service relationship. These can be business or technological affiliations
- Concept Clustering: this discipline is about grouping related conceptual services that collaboratively provide a solution. Clustering is a conceptual operation that can encompass local, remote, and virtual services
- Concept Generalization: to raise the abstraction level of a conceptual service, the generalization method increases the conceptual scope of a solution. This approach is typically used when a service is coarse-grained (too granular)
- Concept Specification: the specification activity enables architects, modelers, and developers to reduce the abstraction level of a service and thus reduce its granularity scope to finer-grained.

Examples

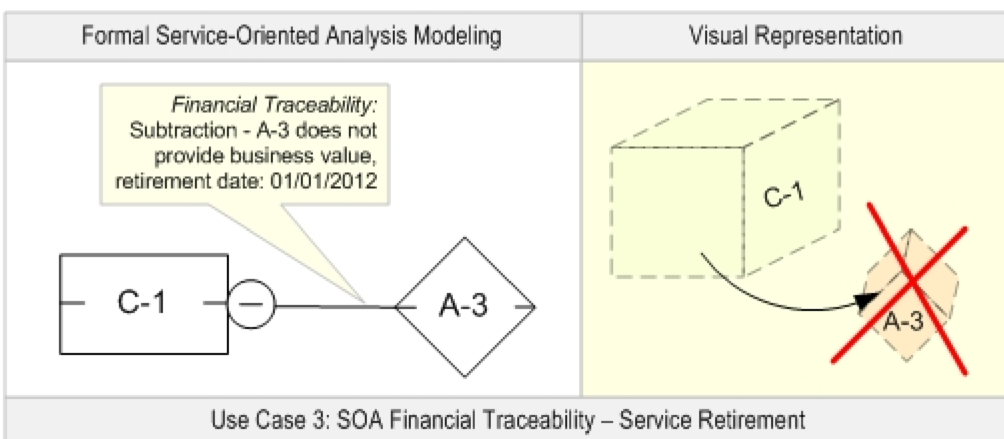
Let us now view a number of service analysis modeling examples.



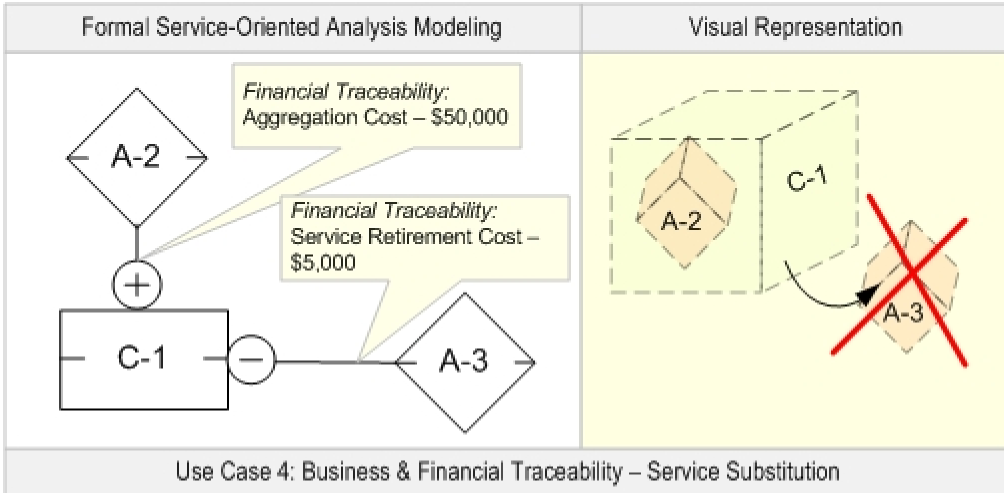
1. Service Aggregation Example



2. Service Decomposition Example



3. Service Subtraction Example



4. Service Substitution Example

- Use Case 1 depicts a simple aggregation case, in which atomic service A-1 is aggregated in composite service C-1 because of SOA best practice reasons.
- Use Case 2 describes service decomposition. Once again, this is because of an SOA best practice rule.
- Use Case 3 illustrates a service retirement (elimination) employing the “subtracted” analysis operation.
- Use Case 4 represents a common business substitution operation. Atomic service A-3 was retired and replaced with atomic service A-2.

Component-based Scalable Logical Architecture

Component-based Scalable Logical Architecture (CSLA) is a software framework created by Rockford Lhotka that provides a standard way to create robust object oriented programs using business objects. Business objects are objects that abstract business entities in an object oriented program. Some examples of business entities include sales orders, employees, or invoices.

Although CSLA itself is free to download, the only documentation the creator provides is his books, which are not free.

CSLA was originally targeted toward Visual Basic 6 in the book *Visual Basic 6.0 Business Objects* by Lhotka ISBN 1-86100-107-X. With the advent of Microsoft .NET, CSLA was completely rewritten from the ground up, with no code carried forward, and called CSLA.NET. This revision took advantage of Web Services and the fully object oriented languages that came with Microsoft .NET (in particular, Visual Basic.NET and C#).

CSLA.NET was expounded in *Expert C# Business Objects* ISBN 1-59059-344-8 and *Expert One-on-One Visual Basic .NET Business Objects* ISBN 1-59059-145-3, both written by Lhotka. Although CSLA and CSLA.NET were originally targeted toward Microsoft programming languages, most of the framework can be applied to most object oriented languages.

Features of CSLA

Smart data

A business object encapsulates all the data and behavior (including persistence logic) associated with the object it represents. For example, an order object will be the only part of the program to load an order, obtain or assign the order's member data (order numbers, etc...), save the order, and so on.

Typed Collections

The framework defines a standard way to create collection objects which represent a collection of objects. This allows an object model to map well to a relational database's data model. For example, an *Account* table could map to an *Accounts* (observe the pluralization) collection object (in reality the object model will differ from the relational model, it is therefore necessary to employ Object-Relational Mapping to resolve the differences). In this case each row in the *Account* table would map to an *Account* business object which is contained in the *Accounts* collection object. This makes it possible to define methods for the collection object that will propagate to its constituent objects. For example, to save all the *Account* objects in an *Accounts* collection, one would type:
`myAccountsCollection.Save();`
The Save() method will call the Save() method on each of its constituent Account objects.

```
// Inside the Accounts class
public void Save(){
    // _accounts will normally be some type of array variable with
class scope
    // that holds all of the Account objects represented in this
collection.
    foreach (Account acct in _accounts){
        acct.Save(); // The actual Account object does the work of
saving itself to the database
    }
}
```

This method of propagating changes to a collection's constituent object can be used for other common methods and property assignments as well(myAccountsCollection.Load(); myAccountsCollection.Active = false;). Business objects can also contain other collection objects, in effect creating a relationship analogous to a parent child relationship between two tables. For example, the *Account* object mentioned above can contain an *Orders* collection object which in turn contains *Order* objects. In this case, the above call to Save() on the *Accounts* collection object could be written to save all of the Account objects as well as all of the orders associated with each of the accounts.

Object persistence

Data creation, retrieval, updates, and deletes (CRUD) are performed by clearly defined methods of the business object associated with the data testing.

Persistence state maintenance

CSLA defines a standard way of allowing a business object to maintain information about its "persistence state". In other words, an object knows when it is new (it represents data that hasn't been saved yet) and when it is dirty (it needs to be saved to the database either because it is new or because its member data has been changed since it was last loaded). Business objects can also be marked for deletion so they can later be deleted (for example when a user has pressed a button confirming his or her intention to delete the rows.)

n-Level undo

This feature makes it possible for an object or collection of objects to maintain a collection of states. This allows the object to easily revert back to previous states. This can be useful when a user wants to undo previous edits multiple times in an application. The feature can also allow a user to redo multiple edits that were previously undone.

Business rule tracking

Allows objects to maintain collections of "broken rule" objects. Broken rules will exist for an object until it is in a valid state, meaning it is ready to be persisted to the database. *BrokenRule* objects are usually associated with validation logic such as ensuring that no alphabetic characters are entered into a phone number field. For example, if an *Account* object has a *PhoneNumber* property, and that property is assigned a phone number with alphabetic characters, the *Account* object's *IsValid* property will become false (making it impossible to save to the database) and then a new *BrokenRule* object will be created and assigned to the Account's *Broken Rules* collection. The rule will disappear when the invalid phone number is corrected making the Account object capable of saving itself to the database.

Extended features of CSLA

Simple UI creation

Using Windows Forms or Web Forms, data-bound controls like DataGrids and ListBoxes can be bound to business objects instead of more generalized database objects like ADO.NET DataSets and DataTables.

Distributed data access

Using Web Services, an object can perform its data access on the client machine or a server. It can also be configured to use manual database transactions or distributed two-phase commit transactions.

Web Services support

Business logic created with the CSLA.NET framework can easily be exposed as a web services to remote consumers. .

Chapter 13

Event-Driven Architecture and Space-Based Architecture

Event-driven architecture

Event-driven architecture (**EDA**) is a software architecture pattern promoting the production, detection, consumption of, and reaction to events.

An *event* can be defined as "a significant change in state". For example, when a consumer purchases a car, the car's state changes from "for sale" to "sold". A car dealer's system architecture may treat this state change as an event to be produced, published, detected and consumed by various applications within the architecture.

This architectural pattern may be applied by the design and implementation of applications and systems which transmit events among loosely coupled software components and services. An event-driven system typically consists of event emitters (or agents) and event consumers (or sinks). Sinks have the responsibility of applying a reaction as soon as an event is presented. The reaction might or might not be completely provided by the sink itself. For instance, the sink might just have the responsibility to filter, transform and forward the event to another component or it might provide a self contained reaction to such event. The first category of sinks can be based upon traditional components such as message oriented middleware while the second category of sinks (self contained online reaction) might require a more appropriate transactional executive framework.

Building applications and systems around an event-driven architecture allows these applications and systems to be constructed in a manner that facilitates more responsiveness, because event-driven systems are, by design, more normalized to unpredictable and asynchronous environments.

Event-driven architecture can complement service-oriented architecture (SOA) because services can be activated by triggers fired on incoming events. This paradigm is particularly useful whenever the sink does not provide any self-contained executive.

SOA 2.0 evolves the implications SOA and EDA architectures provide to a richer, more robust level by leveraging previously unknown causal relationships to form a new event pattern. This new business intelligence pattern triggers further autonomous human or automated processing that adds exponential value to the enterprise by injecting value-added information into the recognized pattern which could not have been achieved previously.

Computing machinery and sensing devices (like sensors, actuators, controllers) can detect state changes of objects or conditions and create events which can then be processed by a service or system. Event triggers are conditions that result in the creation of an event.

Event structure

An event can be made of two parts, the event header and the event body. The event header might include information such as event name, timestamp for the event, and type of event. The event body is the part that describes the fact that has happened in reality. An event body must not be confused with the pattern or the logic that can be applied in reaction to the event itself.

Event flow layers

An event triggered architecture is built on four logical layers. It starts with the sensing of a fact, its technical representation in the form of an event and ends with a non-empty set of reactions to that event.

Event generator

The first logical layer is the event generator, which senses a fact and represents the fact into an event. Since a fact can be almost anything that can be sensed, so can an event generator. As an example, an event generator could be an email client, an E-commerce system or some type of sensor. Converting the different data collected from the sensors to one standardized data form that can be evaluated is a significant problem in the design and implementation of this layer. However, considering that an event is a strongly declarative frame, any transformational operations can be easily applied, thus eliminating the need for a high level of standardization.

Event channel

An event channel is a mechanism whereby the information from an event generator is transferred to the event engine or sink. This could be a TCP/IP connection or any type of input file (flat, XML format, e-mail, etc). Several event channels can be opened at the same time. Usually, because the event processing engine has to process them in near real time, the event channels will be read asynchronously. The events are stored in a queue, waiting to be processed later by the event processing engine.

Event processing engine

The event processing engine is where the event is identified, and the appropriate reaction is selected and executed. This can also lead to a number of assertions being produced. I.e., if the event that comes into the event processing engine is a “product ID low in stock”, this may trigger reactions such as, “Order product ID” and “Notify personnel”.

Downstream event-driven activity

This is where the consequences of the event are shown. This can be done in many different ways and forms; e.g., an email is sent to someone and an application may display some kind of warning on the screen.. Depending on the level of automation provided by the sink (event processing engine) the downstream activity might not be required.

Event processing styles

There are three general styles of event processing: simple, stream, and complex. The three styles are often used together in a mature event-driven architecture.

Simple event processing

Simple event processing concerns events that are directly related to specific, measurable changes of condition. In simple event processing, a notable event happens which initiates downstream action(s). Simple event processing is commonly used to drive the real-time flow of work, thereby reducing lag time and cost.

For example, simple events can be created by a sensor detecting changes in tire pressures or ambient temperature.

Event stream processing

In event stream processing (ESP), both ordinary and notable events happen. Ordinary events (orders, RFID transmissions) are screened for notability and streamed to information subscribers. Stream event processing is commonly used to drive the real-time flow of information in and around the enterprise, which enables in-time decision making.

Complex event processing

Complex event processing (CEP) allows patterns of simple and ordinary events to be considered to infer that a complex event has occurred. Complex event processing evaluates a confluence of events and then takes action. The events (notable or ordinary) may cross event types and occur over a long period of time. The event correlation may be causal, temporal, or spatial. CEP requires the employment of sophisticated event interpreters, event pattern definition and matching, and correlation techniques. CEP is commonly used to detect and respond to business anomalies, threats, and opportunities.

Extreme loose coupling and well distributed

An event driven architecture is extremely loosely coupled and well distributed. The great distribution of this architecture exists because an event can be almost anything and exist almost anywhere. The architecture is extremely loosely coupled because the event itself doesn't know about the consequences of its cause. e.g. If we have an alarm system that records information when the front door opens, the door itself doesn't know that the alarm system will add information when the door opens, just that the door has been opened.

Implementations and examples

Java Swing

The Java Swing API is based on an event driven architecture. This works particularly well with the motivation behind Swing to provide user interface related components and functionality. The API uses a nomenclature convention (e.g. "ActionListener" and "ActionEvent") to relate and organize event concerns. A class which needs to be aware of a particular event simply implements the appropriate listener, overrides the inherited methods, and is then added to the object that fires the event. A very simple example could be:

```
public class FooPanel extends JPanel implements ActionListener {
    public FooPanel() {
        super();

        JButton btn = new JButton("Click Me!");
        btn.addActionListener(this);

        this.add(btn);
    }

    @Override
    public void actionPerformed(ActionEvent ae) {
        System.out.println("Button has been clicked!");
    }
}
```

Alternatively, another implementation choice is to add the listener to the object as an anonymous class. Below is an example.

```
public class FooPanel extends JPanel {
    public FooPanel() {
        super();

        JButton btn = new JButton("Click Me!");
        btn.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ae) {
                System.out.println("Button has been clicked!");
            }
        });
```

Space-based architecture

Space-Based Architecture (**SBA**) is a software architecture pattern for achieving linear scalability of stateful, high-performance applications using the tuple space paradigm. It follows many of the principles of Representational State Transfer (REST), service-oriented architecture (SOA) and Event-driven architecture (EDA), as well as elements of grid computing. With a space-based architecture, applications are built out of a set of self-sufficient units, known as processing-units (PU). These units are independent of each other, so that the application can scale by adding more units.

The SBA model is closely related to other patterns that have been proved successful in addressing the application scalability challenge, such as Shared-Nothing Architecture, used by Google, Amazon.com and other well-known companies. The model has also been applied by many firms in the securities industry for implementing scalable electronic securities trading applications.

Components of Space-Based Architecture

An application built on the principles of space-based architecture typically has the following components:

- **Processing Unit** — the unit of scalability and fail-over. Normally, a processing unit is built out of a POJO (Plain Old Java Object) container, such as that provided by the Spring Framework.

- **Virtual Middleware** — a common runtime and clustering model, used across the entire middleware stack. The core middleware components in a typical SBA architecture are:

Component	Description
Messaging Grid	Handles the flow of incoming transaction as well as the communication between services
Data Grid	Manages the data in distributed memory with options for synchronizing that data with an underlying database
Processing Grid	Parallel processing component based on the master/worker pattern (also known as a blackboard pattern) that enables parallel processing of events among different services

- **POJO-Driven Services Model** — A lightweight services model that can take any standard Java implementation and turn it into a loosely coupled distributed service. The model is ideal for interaction with services that run within the same processing-unit.

- **SLA-Driven Container** — The SLA-driven container enables the deployment of the application on a dynamic pool of machines based on Service Level Agreements. SLA definitions include the number of instances that need to run in order to comply with the application scaling and fail-over policies, as well as other policies.

Chapter 14

Software Architect and Software Architectural Model

Software architect

Software architect is a general term with many accepted definitions, which refers to a broad range of roles. Generally-accepted terminology and certifications began appearing in connection with this role near the beginning of the 21st century.

History

With the increased popularity of multi-tier application development, the choices of how an application can be built have also increased. Given that expansion, the risk that a software development project may inadvertently create an end product that in essence already exists has grown markedly. A new 'Software architect' role became necessary during software development.

The software architect concept began to take hold when object oriented programming (OOP) was coming into more widespread use (in the late 1990s and early years of the 21st century). OOP allowed ever-larger and more complex applications to be built, which in turn required increased high-level application and system oversight.

The main responsibilities of a software architect include:

- Limiting the choices available during development by
    - choosing a standard way of pursuing application development
    - creating, defining, or choosing an application framework for the application
- Recognizing potential reuse in the organization or in the application by
    - Observing and understanding the broader system environment
    - Creating the component design
    - Having knowledge of other applications in the organization

Software architects can also:

- Subdivide a complex application, during the design phase, into smaller, more manageable pieces
- Grasp the functions of each component within the application
- Understand the interactions and dependencies among components
- Communicate these concepts to developers

In order to perform these responsibilities effectively, software architects often use Unified Modeling Language and OOP. UML has become an important tool for software architects to use in communicating the overall system design to developers and other team members, comparable to the drawings made by building architects.

Duties

Despite the lack of an accepted overall definition, the role of software architect generally has certain common traits:

Design

The architect makes high-level design choices much more often than low-level choices. In addition, the architect may sometimes dictate technical standards, including coding standards, tools, or platforms, so as to advance business goals rather than to place arbitrary restrictions on the choices of developers. Note that software architects rarely deal with the physical architecture of the hardware environment, confining themselves to the design methodology of the code.

Communication

Architects also have to communicate effectively, not only to understand the business needs, but also to advance their own architectural vision. They can do so verbally, in writing, and through various software architectural models that specialize in communicating architecture.

Types of software architects

The enterprise architect handles business-related software decisions that frequently can involve multiple software systems within an organization, spanning several projects teams, and often at more than one site. The Enterprise Architect may seldom see or interact with source code.

An application architect works with a single software application. This may be a full- or a part-time role. The application architect is almost always an active software developer.

Other similar titles in use, but without consensus on their exact meaning, include:

- Solutions Architect, which may refer to a person directly involved in advancing a particular business solution needing interactions between multiple applications. May also refer to an Application Architect.
- System Architect (singular), which is often used as a synonym for Application Architect.
- Systems Architect (plural), which is often used as a synonym for Enterprise Architect or Solutions Architect.

The table below indicates many of the differences between various kinds of software architects:

Architect Type	Strategic Thinking	System Interactions	Communication	Design
Enterprise Architect	Across Projects	Highly Abstracted	Across Organization	Minimal, High Level
Solutions Architect	Focused on solution	Very Detailed	Multiple Teams	Detailed
Application Architect	Component re-use, maintainability	Centered on single Application	Single Project	Very Detailed

In the software industry, as the table above suggests, the various versions of architect do not always have the same goals.

Architect metaphor

The term "software architect" came into being because of the perceived similarities between the creation of software and the creation of buildings. Although a simplified construction metaphor may be flawed, the term is still meaningful in the sense that it describes the "design" aspect of the job.

The use of any form of the word 'architect' is regulated by 'title acts' in many states in the UK and the US, and a person must be licensed as a building architect to use it. Enforcement of these laws may be lax, if it is immediately clear that the title refers to the computer industry and not the building industry.

Ivory towers

When architects become too disconnected from the actual developers, they are often dismissively termed "architards", "architecture astronauts", or "Ivory Tower Architects". Architects may think this term is often incorrectly used by developers who do not have the experience or knowledge to comprehend the architecture.

Application or solutions architects work at a level of detail that demands involvement in actual coding, and will function best with a substantial background in software development. A school of thought holds that enterprise architects should also have a development background, so as to avoid the issues that can arise from an ivory-tower approach.

Software Architectural Model

An **Architectural Model** (in software) is a rich and rigorous diagram, created using available standards, in which the primary concern is to illustrate a specific set of tradeoffs inherent in the structure and design of a system or ecosystem. Software architects use architectural models to communicate with others and seek peer feedback.

An architectural model is an expression of a viewpoint in software architecture. There are many definitions of software architecture.

Key elements of the definition of a software architectural model:

- **rich**: for the viewpoint in question, there should be sufficient information to describe the area in detail. The information should not be lacking or vague. The goal is to minimize misunderstandings, not perpetuate them.
- **rigorous**: the architect has applied a specific methodology to create this particular model, and the resulting model 'looks' a particular way. Here's the test of rigorousness: If two architects, in different cities, were describing the same thing, the resulting diagrams would be nearly identical (with the possible exception of visual layout, to a point).
- **diagram**: in general, a model may refer to *any* abstraction that simplifies something for the sake of addressing a particular viewpoint. This definition specifically subclasses 'Architectural Models' to the subset of model descriptions that are represented as diagrams.
- **standards**: standards work when everyone knows them and everyone uses them. This allows a level of communication that cannot be achieved when each diagram is substantially different from another. UML is the most often quoted standard.
- **primary concern**: it is easy to be too detailed by including many different needs in a single diagram. This should be avoided. It is better to draw multiple diagrams, one for each viewpoint, than to draw a 'mega diagram' that is so rich in content that it requires a two-year course of study to understand it. Remember this: when building houses, the architect delivers many different diagrams. Each is used differently. Frequently the final package of plans will include diagrams with the floor plan many times: framing plan, electrical plan, heating plan, plumbing, etc. They don't just say: it's a floor plan so 100% of the information that CAN go on a floor plan should be put there. The plumbing subcontractor doesn't need the details that the electrician cares about.
- **illustrate**: the idea behind creating a model is to communicate and seek valuable feedback. The goal of the diagram should be to answer a specific question and to share that answer with others to (a) see if they agree, and (b) guide their work. Rule of thumb: know what it is you want to say, and whose work you intend to influence with it.

- **specific set of tradeoffs**: the Architecture Tradeoff Analysis Method (ATAM) methodology describes a process whereby software architecture can be peer-reviewed for appropriateness. ATAM does this by starting with a basic notion: there is no such thing as a 'one-size-fits-all' design. We can create a generic design, but then we need to alter it to specific situations based on the business requirements. In effect, we make tradeoffs. The diagram should make those specific tradeoffs visible. Therefore, before an architect creates a diagram, he or she should be prepared to describe, in words, which tradeoffs they are attempting to illustrate in this model.
- **tradeoffs inherent in the structure and design**: a component is not a tradeoff. Tradoffs rarely translate into an image on the diagram. Tradeoffs are the first principles that produced the design models. When an architect wishes to describe or defend a particular tradeoff, the diagram can be used to defend the position.
- **system or ecosystem**: modeling in general can be done at different levels of abstraction. It is useful to model the architecture of a specific application, complete with components and interactions. It is also reasonable to model the systems of applications needed to deliver a complete business process (like order-to-cash). It is not commonly useful, however, to view the model of a single component and its classes as software architecture. At that level, the model, while valuable in its own right, illustrates design much more so than architecture.

Chapter 15

Functional Software Architecture and COLA (Software Architecture)

Functional Software Architecture

A **Functional Software Architecture** (**FSA**) is an architectural model that identifies enterprise functions, interactions and corresponding IT needs. These functions can be used as reference by different domain experts to develop IT-systems as part of a co-operative information-driven enterprise. In this way both software engineers and enterprise architects are able to create an information-driven, integrated organizational environment.

Overview

When an integrated software system needs to be developed and implemented normally a number of tasks and corresponding responsibilities can be divided:

1. Strategic management and business consultants set objectives in relation to a more efficient/effective business process.
2. Enterprise engineers come up with a design of a more efficient business process and a request for a certain information system in the form of an Enterprise Architecture.
3. Software engineers come up with the design of this information system, which describes the components and structural features of the system by use of a certain Architecture Description Language (ADL).
4. Computer programmers code the different modules and actually implement the system.

The described work division is in reality much more complex and also involves more actors but it outlines the involvement of people with different backgrounds in creating a software system that enables the organization to reach business objectives. A wide variety of material produced by different actors within this system development process needs to be exchanged between, and understood by, multiple actors.

Especially in the field of software engineering many tools (A4 Tool, CAME, ARIS), languages (ACME, Rapide, UML) and methods (DSDM, RUP, ISPL) are developed and extensively used. Also, the transition between the software engineers (step 3) and computer programmers (step 4) is already highly formalized by, for instance, object-oriented development.

Setting strategic objectives (step 1) and the corresponding search for business opportunities and weaknesses is a subject extensively discussed and investigated for more than hundred years. Concepts like Business process reengineering, Product software market analysis, Requirements analysis are commonly known and extensively used in this context. These strategic inputs must be used for the development of a good enterprise design (step 2), which can then be used for software design and implementation respectively.

Recent studies have shown that these enterprise architectures can be developed by a number of different methods and techniques. Before these methods and techniques are discussed in detail a definition of an Enterprise architecture is given:

An Enterprise Architecture is a strategic information asset base, which defines the mission, the information necessary to perform the mission and the technologies necessary to perform the mission, and the transitional processes for implementing new technologies in response to the changing mission needs.

This definition emphasizes the use of the architecture as a rich strategic information source for the improvement of business processes and development of needed information systems. If defined, maintained, and implemented effectively, these institutional blueprint assist in optimizing the interdependencies and interrelationships among an organizations business operations and the underlying IT that support operations.

Having read the definition of a Functional Software Architecture at the beginning of this entry we can see a Functional Software Architecture as a type of Enterprise Architecture that can be used as a rich reference for the development of an integrated information system. Naming it a Functional Software Architecture encourages practitioners to use it as a strategic input for a technical architecture. A formal mapping between a Functional Software Architecture and a type of ADL is therefore needed. In this way the formal use and reuse of enterprise architectures as strategic input for software architectures can be realized.

Development of an FSA

As the boundary of an enterprise is extended, it becomes increasingly important that a common “big picture” of needed business, people and IT system activities is developed and shared by all the parties involved. A Functional Software Architecture does this by breaking down the organization in business functions and corresponding IT needs. In this way the enterprise engineer provides a rich schematic reference that can be used by the software engineer in the development of these IT-systems.

The development of a Functional Software Architecture can be done by a number of (combined) methods and techniques. Filling in the “gap” between the enterprise engineers and software engineers through the use of different combinations of methods and techniques will be the main objective. However, this objective can only be reached when combined methods result in clear and rich functional software architectures that are developed and used by both parties.

Optimizing the internal and external business processes through process reengineering is one of the main objectives an enterprise can have in times of high external pressure. A business process involves value creating activities with certain inputs and outputs, which are interconnected and thereby jointly contribute to the final outcome (product or service) of the process. Process reengineering covers a variety of perspectives of how to change the organization. It is concerned with the redesign of strategic, value adding processes, systems, policies and organizational structures to optimize the processes of an organization.

Modeling the business

Within the area of enterprise engineering formal methodologies, methods and techniques are designed, tested and extensively used in order to offer organizations reusable business process solutions:

- Computer Integrated Manufacturing Open Systems Architecture (CIMOSA) methodology
- Integrated DEFinition (IDEF) methodology
- Petri Nets
- Unified Modeling Language (UML) or Unified Enterprise Modeling Language (UEML)
- Enterprise Function Diagrams (EFD)

These methodologies/techniques and methods are all more or less suited in modeling the enterprise and its underlying processes. So, which of them are suited for the further development of Information Technology systems that are needed for effective and efficient (re)designed processes? More important, why using a time consuming enterprise methodology when information and software engineers can’t or won’t use the unclear results in the development of efficiency enabling IT systems? Before we can give the answers to these questions some short descriptions of the listed methods above are given.

Computer Integrated Manufacturing Open Systems Architecture

CIMOSA provides templates and interconnected modeling constructs to encode business, people and IT aspects of enterprise requirements. This is done from multiple perspectives: Information view, Function view, Resource view and Organization view.

These constructs can further be used to structure and facilitate the design and implementation of detailed IT systems.

The division in different views makes it a clarifying reference for enterprise and software engineers. It shows information needs for different enterprise functionalities (activities, processes, operations) and corresponding resources. In this way it can easily be determined which IT-system will fulfill the information needs in a certain activity and process.

Integrated DEFinition

IDEF is a structured modeling technique, which was first developed for the modeling of manufacturing systems. It was already being used by the U.S. Airforce in 1981. Initially it had 4 different notations to model an enterprise from a certain viewpoint. These were IDEF0, IDEF1, IDEF2 and IDEF3 for functional, data, dynamic and process analysis respectively. In the past decades a number of tools and techniques for the integration of the notations are developed in an incremental way.

IDEF clearly shows how a business process flows through a variety of decomposed business functions with corresponding information inputs, outputs and actors. Like CIMOSA, it also uses different enterprise views. Moreover, IDEF can be easily transformed into UML-diagrams for the further development of IT systems. These positive characteristics make it a powerful method for the development of Functional Software Architectures.

Petri Nets

Petri Nets are known tools to model manufacturing systems. They are highly expressive and provide good formalisms for the modeling of concurrent systems. The most advantageous properties are that of simple representation of states, concurrent system transitions and capabilities to model the duration of transitions.

Petri Nets therefore can be used to model certain business processes with corresponding state and transitions or activities with in and outputs. Moreover, Petri Nets can be used to model different software systems and transitions between these systems. In this way programmers use it as a schematic coding reference.

In recent years a number of attempts have shown that Petri Nets can contribute to the development of business process integration. One of these is the Model Blue methodology, which is developed by IBM Chinese Research Laboratory and outlines the importance of model driven business integration as an emerging approach for building integrated platforms. A mapping between their Model Blue business view and an equivalent Petri Net is also shown, which indicates that their research closes the gap between business and IT. However, instead of Petri Nets they rather use their own Model Blue IT view, which can be derived from their business view through a transformation engine.

Unified Modeling Language

UML is a broadly accepted modeling language for the development of software systems and applications. The, so called, “object oriented community” also tries to use UML for enterprise modeling purposes. They emphasize the use of enterprise objects or business objects from which complex enterprise systems are made. A collection of these objects and corresponding interactions between them can represent a complex business system or process. Where Petri Nets focus on the interaction and states of objects, UML focuses more on the business objects themselves. Sometimes these are called the “enterprise building blocks”, which includes resources, processes, goals, rules and metamodels. Despite the fact that UML in this way can be used to model an integrated software system it has been argued that the reality of business can be modeled with a software modeling language. In reaction the object oriented community makes business extensions for UML and adapts the language. UEML is derived from UML and is proposed as a business modeling language. The question remains if this business transformation is the right thing to do. It was earlier said that UML in combination with other “pure’ business methods can be a better alternative.

Enterprise Function Diagrams

EFD is a used modeling technique for the representation of enterprise functions and corresponding interactions. Different business processes can be modeled in these representations through the use of “function modules” and triggers. A starting business process delivers different inputs to different functions. A process flowing through all the functions and sub-functions creates multiple outputs. Enterprise Function Diagrams hereby give a very easy-to-use and detailed representation about a business process and corresponding functions, inputs, outputs and triggers. In this way EFD has many similarities with IDEF0 diagrams, which also represent in a hierarchical way business processes as a combination of functions and triggers. Difference is that an EFD places the business functions in an organization hierarchical perspective, which outlines the downstream of certain processes in the organization. On the contrary, IDEF0 diagrams show responsibilities of certain business functions through the use of arrows. Also, IDEF0 has a clear representation of inputs and outputs of every (sub)function.

EFD possibly could be used as a business front-end to a software modeling language like UML. The major resemblance with IDEF as a modeling tool indicates that it can be done. However, more research is needed to improve the EFD technique in such a way that formal mappings to UML can be made. about the complementary use of IDEF and UML has contributed to the acceptance of IDEF as business-front end. A similar study should be done with EFD and UML.

COLA (software architecture)

Introduction

COLA stands for Combined Object Lambda Architecture, and is a system for experimenting with software design currently being investigated by the Viewpoints Research Institute. A COLA is a self-describing language in two parts, an object system which is implemented in terms of objects, and a functional language to describe the computation to perform.

Since a COLA is written in itself, the whole environment (when bootstrapped) can be rewritten and extended by programming with the COLA; in other words, it does not require more knowledge to rewrite a COLA than it does to write a program to run in it (as opposed to running Python code in CPython for example, which requires knowledge of C in order to reprogram the language).

This flexibility has led to the work-in-progress COLA called 'idst' becoming the implementation vehicle of choice for the Viewpoints Research Institute's research into 'reinventing programming', since it allows rapid creation and modification of new programming languages for study.

Description

A COLA is designed to be the simplest possible language which can be described in itself, so that the implementation exactly describes itself. In order to do this the structure of the environment is separated from the semantics of the computation performed.

The object system describes the structure of a prototype-based Object Oriented environment. This is implemented in terms of objects and message passing, which is in fact the same system it is describing. This allows modification of the system by using the same object oriented knowledge used to write any other application.

This object system is turned into a useful programming language by complementing it with a functional language describing what each object's methods do. The methods called from the object language are closures running a functional programming language.

Combined together, these two parts form a complete prototype-based Object Oriented programming language which is entirely self-hosting.

Natural Language Analogy

In order to illustrate the concept we can consider an analogy in natural language, say English. To define the whole of English for someone who speaks a foreign language would be a monumental task, especially since it would have to be done over and over again for each foreign language we're coming from. However, we could instead define a simpler subset of English as a base which is just expressive enough to understand definitions given in English. For example, such a subset would not need a word for "giraffe", since it could be added later with a statement like "A giraffe is a herbivore with a long neck." Similarly the definitions of herbivore, neck and long can be added later with other statements, and so on. This way we can remove every part of English which we do not need in our subset.

The bits we keep are those which are needed to understand definitions and statements (so that we can expand the language later), along with everything needed to define those, and so on. What we end up with is a self-contained language, written in itself (a subset of English) and capable of being expanded with statements like the giraffe one above.

Any English speaker is thus capable of changing the language itself as easily as they speak (since it's defined in English) by rewriting, overriding or bypassing the statements given in the base, turning the language into anything (including existing ones).

Also, anyone can become an English speaker simply by having this base translated into their native tongue (a more tractable problem than translating the whole of English). Once they know this subset then they know enough English to understand other statements like the giraffe one, and thus grow their knowledge to the whole language through English sentences (which can be reused by everyone, regardless of their first language). This is analogous to the bootstrapping and compatibility of a COLA.

The way a COLA such as idst implements this can be thought of as defining words using other words (the object system) separately to defining the grammar (the functional language).

Features

A COLA can be used in two ways:

Due to their flexible and extensibility it is possible to make COLAs compatible with many ABIs, which allows integration into existing libraries (for example, those written in C) whilst maintaining the ability to mutate the COLA into another (perhaps custom) language.

A completely COLA-based computer system, whilst capable of implementing the operating system, libraries, applications and other levels of a traditional computer system, allows these distinctions to blur or disappear if the end-user wishes. Every aspect of the computer system, since it is written in a COLA (including the COLA itself), can be overridden, mutated, bypassed, etc. just as the local datastructures and functions in a traditional program can. There is also flexibility in how code is run, since there is a choice of interpreting, static compilation, dynamic compilation, in fact if the COLA is given a suitable backend object then it can even reprogram FPGA's to run arbitrary sections of the system.

Current Implementation

Idst

Ian Piumarta's 'idst' system (the name is currently in flux) is a work-in-progress implementation of a COLA. It consists of several components, such as the Id object model, the Jolt function language and the Pepsi object oriented language. Pepsi was bootstrapped by writing two Pepsi compilers, one in C++ and one in Pepsi, then compiling the latter with the former, then in with itself. This made Pepsi self-hosting, and the C++ version was discarded.

Chapter 16

IEEE 1471 and Representational State Transfer

IEEE 1471

IEEE 1471 is an IEEE Standard for describing the *architecture of a software-intensive system*, also known as software architecture or system architecture.

Overview

IEEE 1471 is the short name for a standard formally known as ANSI/IEEE 1471-2000, *Recommended Practice for Architecture Description of Software-Intensive Systems.* Within IEEE parlance, this is a *Recommended Practice*, the least normative of the kinds of IEEE standards. In 2007 this standard was adopted by ISO/IEC JTC1/SC7 as ISO/IEC 42010:2007, *Systems and Software Engineering -- Recommended practice for architectural description of software-intensive systems*.

It has long been recognized that "architecture" has a strong influence over the life cycle of a system. However, until relatively recently, hardware issues have tended to dominate architectural thinking, and software aspect, when considered at all, were often the first to be compromised under the pressures of development. IEEE 1471 was created to provide a basis for thinking about the architecture of software-intensive systems.

The IEEE 1471's contributions can be summarised as follows (in this list, items in *italics* are terms defined by and used in the standard):

- It provides definitions and a meta-model for the description of *architecture*
- It states that an *architecture* should address a system's *stakeholders concerns*
- It asserts that *architecture description*s are inherently multi-view, no single *view* adequately captures all stakeholder concerns
- It separates the notion of *view* from *viewpoint,* where a *viewpoint* identifies the set of *concerns* and the *representations/modeling techniques*, etc. used to describe the *architecture* to address those *concerns* and a *view* is the result of applying a viewpoint to a particular system.

- It establishes content requirements for architecture descriptions and the idea that a *conforming architecture description* has a 1-to-1 correspondence between its *viewpoints* and its *views*.
- It provides guidance for capturing *architecture rationale* and identifying inconsistencies/unresolved issues between the *views* within an *architecture description*

IEEE 1471-2000 provides informative annexes that relate the concepts in IEEE 1471 to architecture concepts in other standards, including RM-ODP and IEEE 12207.

History

In August 1995, the IEEE Software Engineering Standards Committee (SESC) chartered an IEEE Architecture Planning Group (APG) *to set direction for incorporating architectural thinking into IEEE standards*. In April 1996, the Architecture Working Group (AWG) was created to implement the recommendations made by APG to the SESC. The AWG was chaired by Basil Sherlund, vice-chairs Ronald Wade, David Emery, the specification was edited by Rich Hilliard. The AWG had 25 members. Drafts of the specification were balloted and commented on by 130 international reviewers. In September 2000, the IEEE-SA Standards Board approved the specification as IEEE Std 1471-2000.

In 2006, ISO/IEC Joint Technical Committee 1 (JTC1), Information technology/Subcommittee SC 7, Software and systems engineering, adopted the specification as ISO/IEC 42010, under a special “fast-track procedure”, in parallel with its approval by national bodies of ISO and IEC. A coordinated revision of this standard by ISO/IEC JTC1/SC7/WG42 and IEEE CS commenced in 2006, following the successful ISO/IEC fast-track ballot and in line with the IEEE standard 5-year review of the standard.

The purpose of an architecture description

According to IEEE 1471 an architecture description can be used for the following:

- Expression of the system and its evolution
- Communication among the system stakeholders
- Evaluation and comparison of architectures in a consistent manner
- Planning, managing, and executing the activities of system development
- Expression of the persistent characteristics and supporting principles of a system to guide acceptable change
- Verification of a system implementation’s compliance with an architectural description
- Recording contributions to the body of knowledge of software-intensive systems architecture

IEEE Terminology related to software architecture

According to IEEE Standard Glossary of Software Engineering Terminology the following definitions are used:

- *architect*: The person, team, or organization responsible for designing systems architecture.
- *architectural description* (AD): A collection of products to document an architecture.
- *architecture*: The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.
- *designing*: The activities of defining, documenting, maintaining, improving, and certifying proper implementation of an architecture.
- *system*: A collection of components organized to accomplish a specific function or set of functions. The term *system* encompasses individual applications, systems in the traditional sense, subsystems, systems of systems, product lines, product families, whole enterprises, and other aggregations of interest.
- *system stakeholder*: An individual, team, or organization (or classes thereof) with interests in, or concerns relative to, a system.
- *view*: A representation of a whole system from the perspective of a related set of concerns.
- *viewpoint*: A specification of the conventions for constructing and using a view. A pattern or template from which to develop individual views by establishing the purposes and audience for a view and the techniques for its creation and analysis.

IEEE Conceptual Framework for Architecture Description

IEEE 1471 uses the following conceptual framework .

1. A system's environment, or *context'*, can influence that system. The environment can include other systems that interact with the system of interest, either directly via interfaces or indirectly in other ways. The environment determines the boundaries that define the *scope* of the system of interest relative to other systems.
2. A system has one or more *stakeholders*. Each stakeholder typically has interests in, or concerns relative to, that system.
3. *Concerns* are those interests which pertain to the system's development, its operation or any other aspects that are critical or otherwise important to one or more stakeholders. Concerns include system considerations such as performance, reliability, security, distribution, and evolvability.
4. A system exists to fulfill one or more *missions* in its environment. A *mission* is a use or operation for which a system is intended by one or more stakeholders to meet some set of *objectives*.
5. Every system has an *architecture*, whether understood or not; whether recorded or conceptual. An architecture can be recorded by an *architectural description*.

6. An architectural description is organized into one or more constituents called (architectural) *views*. Each *view* addresses one or more of the concerns of the system stakeholders. A *view* is a partial expression of a system's architecture with respect to a particular *viewpoint*.
7. A *viewpoint* establishes the conventions by which a view is created, depicted and analyzed. In this way, a view *conforms* to a viewpoint. The viewpoint determines the languages (including notations, model, or product types) to be used to describe the view, and any associated modeling methods or analysis techniques to be applied to these representations of the view. These languages and techniques are used to yield results relevant to the concerns addressed by the viewpoint.
8. An architectural description *selects* one or more viewpoints for use. The *selection of viewpoints* is typically based on consideration of the stakeholders to whom the AD is addressed and their concerns. A *viewpoint definition* may originate with an AD, or it may have been defined elsewhere (a *library viewpoint*).
9. A view may consist of one or more *architectural models*. Each such architectural model is developed using the methods established by its associated architectural viewpoint. An architectural model may participate in more than one view.

Conformance to the IEEE 1471

IEEE 1471 defines a set of normative requirements for conforming architecture descriptions, including the following:

- AD identification, version, and overview information (clause 5.1)
- Identification of the system stakeholders and their concerns judged to be relevant to the architecture (clause 5.2)
- Specifications of each viewpoint that has been selected to organize the representation of the architecture and the rationale for those selections (clause 5.3)
- One or more architectural views (clause 5.4)
- A record of all known inconsistencies among the architectural description's required constituents (clause 5.5)
- A rationale for selection of the architecture (clause 5.6)

Representational State Transfer

Representational State Transfer (**REST**) is a style of software architecture for distributed hypermedia systems such as the World Wide Web. The term *Representational State Transfer* was introduced and defined in 2000 by Roy Fielding in his doctoral dissertation. Fielding is one of the principal authors of the Hypertext Transfer Protocol (HTTP) specification versions 1.0 and 1.1.

Conforming to the REST constraints is referred to as being 'RESTful'.

History

The REST architectural style was developed in parallel with the HTTP/1.1 protocol, based on the existing design of HTTP/1.0. The largest known implementation of a system conforming to the REST architectural style is the World Wide Web. REST exemplifies how the Web's architecture emerged by characterizing and constraining the macro-interactions of the four components of the Web, namely origin servers, gateways, proxies and clients, without imposing limitations on the individual participants. As such, REST essentially governs the proper behavior of participants.

Concept

REST-style architectures consist of clients and servers. Clients initiate requests to servers; servers process requests and return appropriate responses. Requests and responses are built around the transfer of representations of resources. A resource can be essentially any coherent and meaningful concept that may be addressed. A representation of a resource is typically a document that captures the current or intended state of a resource.

At any particular time, a client can either be in transition between application states or "at rest". A client in a rest state is able to interact with its user, but creates no load and consumes no per-client storage on the set of servers or on the network.

The client begins sending requests when it is ready to make the transition to a new state. While one or more requests are outstanding, the client is considered to be in transition. The representation of each application state contains links that may be used next time the client chooses to initiate a new state transition.

REST was initially described in the context of HTTP, but is not limited to that protocol. RESTful architectures can be based on other Application Layer protocols if they already provide a rich and uniform vocabulary for applications based on the transfer of meaningful representational state. RESTful applications maximize the use of the pre-existing, well-defined interface and other built-in capabilities provided by the chosen network protocol, and minimize the addition of new application-specific features on top of it.

HTTP examples

HTTP, for example, has a very rich vocabulary in terms of verbs (or "methods"), URIs, Internet media types, request and response codes, etc. REST uses these existing features of the HTTP protocol, and thus allows existing layered proxy and gateway components to perform additional functions on the network such as HTTP caching and security enforcement. An abbreviated list of claimed REST Examples is available.

SOAP RPC contrast

SOAP RPC over HTTP, on the other hand, encourages each application designer to define a new and arbitrary vocabulary of nouns and verbs (for example `getUsers()`, `savePurchaseOrder(...)`), usually overlaid onto the HTTP 'POST' verb. This disregards many of HTTP's existing capabilities such as authentication, caching and content type negotiation, and may leave the application designer re-inventing many of these features within the new vocabulary. Examples of doing so may include the addition of methods such as `getNewUsersSince(Date date)`, `savePurchaseOrder(string customerLogon, string password, ...)`.

Constraints

The REST architectural style describes the following six constraints applied to the architecture, while leaving the implementation of the individual components free to design:

Client–server
: Clients are separated from servers by a uniform interface. This separation of concerns means that, for example, clients are not concerned with data storage, which remains internal to each server, so that the portability of client code is improved. Servers are not concerned with the user interface or user state, so that servers can be simpler and more scalable. Servers and clients may also be replaced and developed independently, as long as the interface is not altered.

Stateless
: The client–server communication is further constrained by no client context being stored on the server between requests. Each request from any client contains all of the information necessary to service the request, and any session state is held in the client. The server can be stateful; this constraint merely requires that server-side state be addressable by URL as a resource. This not only makes servers more visible for monitoring, but also makes them more reliable in the face of partial or network failures as well as further enhancing their scalability.

Cacheable
: As on the World Wide Web, clients are able to cache responses. Responses must therefore, implicitly or explicitly, define themselves as cacheable or not to prevent clients reusing stale or inappropriate data in response to further requests. Well-managed caching partially or completely eliminates some client–server interactions, further improving scalability and performance.

Layered system
: A client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary along the way. Intermediary servers may improve system scalability by enabling load balancing and by providing shared caches. They may also enforce security policies.

Code on demand (optional)

Servers are able temporarily to extend or customize the functionality of a client by transferring logic to it that it can execute. Examples of this may include compiled components such as Java applets and client-side scripts such as JavaScript.

Uniform interface

The uniform interface between clients and servers, discussed below, simplifies and decouples the architecture, which enables each part to evolve independently. The four guiding principles of this interface are detailed below.

The only optional constraint of REST architecture is code on demand. If a service violates any other constraint, it cannot strictly be referred to as RESTful.

Complying with these constraints, and thus conforming to the REST architectural style, will enable any kind of distributed hypermedia system to have desirable emergent properties, such as performance, scalability, simplicity, modifiability, visibility, portability and reliability.

Guiding principles of the interface

The uniform interface that any REST interface must provide is considered fundamental to the design of any REST service.

Identification of resources

Individual resources are identified in requests, for example using URIs in web-based REST systems. The resources themselves are conceptually separate from the representations that are returned to the client. For example, the server does not send its database, but rather, perhaps, some HTML, XML or JSON that represents some database records expressed, for instance, in Swedish and encoded in UTF-8, depending on the details of the request and the server implementation.

Manipulation of resources through these representations

When a client holds a representation of a resource, including any metadata attached, it has enough information to modify or delete the resource on the server, provided it has permission to do so.

Self-descriptive messages

Each message includes enough information to describe how to process the message. For example, which parser to invoke may be specified by an Internet media type (previously known as a MIME type). Responses also explicitly indicate their cacheability.

Hypermedia as the engine of application state

If it is likely that the client will want to access related resources, these should be identified in the representation returned, for example by providing their URIs in sufficient context, such as hypertext links.

Key goals

Key goals of REST include:

- Scalability of component interactions
- Generality of interfaces
- Independent deployment of components
- Intermediary components to reduce latency, enforce security and encapsulate legacy systems

REST has been applied to describe the desired web architecture, helped to identify existing problems, to compare alternative solutions, and to ensure that protocol extensions would not violate the core constraints that make the Web successful.

Fielding describes REST's effect on scalability thus:

REST's client–server separation of concerns simplifies component implementation, reduces the complexity of connector semantics, improves the effectiveness of performance tuning, and increases the scalability of pure server components. Layered system constraints allow intermediaries—proxies, gateways, and firewalls—to be introduced at various points in the communication without changing the interfaces between components, thus allowing them to assist in communication translation or improve performance via large-scale, shared caching. REST enables intermediate processing by constraining messages to be self-descriptive: interaction is stateless between requests, standard methods and media types are used to indicate semantics and exchange information, and responses explicitly indicate cacheability.

Central principle

An important concept in REST is the existence of resources (sources of specific information), each of which is referenced with a global identifier (e.g., a URI in HTTP). In order to manipulate these resources, *components* of the network (user agents and origin servers) communicate via a standardized interface (e.g., HTTP) and exchange *representations* of these resources (the actual documents conveying the information). For example, a resource that represents a circle may accept and return a representation that specifies a center point and radius, formatted in SVG, but may also accept and return a representation that specifies any three distinct points along the curve (since this also uniquely identifies a circle) as a comma-separated list.

Any number of *connectors* (e.g., clients, servers, caches, tunnels, etc.) can mediate the request, but each does so without "seeing past" its own request (referred to as "layering," another constraint of REST and a common principle in many other parts of information and networking architecture). Thus, an application can interact with a resource by knowing two things: the identifier of the resource and the action required—it does not need to know whether there are caches, proxies, gateways, firewalls, tunnels, or anything else between it and the server actually holding the information. The application does, however, need to understand the format of the information (*representation*) returned, which is typically an HTML, XML or JSON document of some kind, although it may be an image, plain text, or any other content.

RESTful web services

A RESTful web service (also called a RESTful web API) is a simple web service implemented using HTTP and the principles of REST. It is a collection of resources, with three defined aspects:

- the base URI for the web service, such as `http://example.com/resources/`
- the MIME type of the data supported by the web service. This is often JSON, XML or YAML but can be any other valid MIME type.
- the set of operations supported by the web service using HTTP methods (e.g., POST, GET, PUT or DELETE).

The following table shows how the HTTP verbs are typically used to implement a web service.

RESTful Web Service HTTP methods

Resource	GET	PUT	POST	DELETE
Collection URI, such as **`http://example.com/resources/`**	**List** the URIs and perhaps other details of the collection's members.	**Replace** the entire collection with another collection.	**Create** a new entry in the collection. The new entry's URL is assigned automatically and is usually returned by the operation.	**Delete** the entire collection.
Element URI, such as **`http://example.com/resources/142`**	**Retrieve** a representation of the addressed member of the collection, expressed in an appropriate Internet media type.	**Update** the addressed member of the collection, or if it doesn't exist, **create** it.	Treat the addressed member as a collection in its own right and **create** a new entry in it.	**Delete** the addressed member of the collection.

The PUT and DELETE methods are idempotent methods. The GET method is a safe method (no side-effect, implies idempotence as well).

Unlike SOAP-based web services, there is no "official" standard for RESTful web services. This is because REST is an architecture, unlike SOAP, which is a protocol. Even though REST is not a standard, a RESTful implementation such as the Web can use standards like HTTP, URL, XML, PNG, etc.

Public implementations

REST can be found in a number of places on the public Web:

- The Atom Publishing Protocol for publishing to blogs is considered a canonical RESTful protocol.
- Sun Microsystems' Cloud API is a good example of resource media type documentation.
- The Open Services for Lifecycle Collaboration (OSLC) initiative is establishing a RESTful approach to integrating software development artifacts.
- CouchDB is a document-oriented database written in Erlang that provides a RESTful JSON API that can be accessed from any environment that allows HTTP requests.
- Microsoft's Canonical REST Entity Service.

Framework implementations

- .NET Open-Source - OpenRasta
- ColdFusion - ColdFusion on Wheels
- Ext JS
- Java Jt Design Pattern Framework, Wink , Restlet, JBoss RESTEasy, Jersey, Apache CXF, NetKernel, Apache Sling, Restfulie
- Microsoft's Azure Services Platform
- Microsoft's WCF Data Services
- Perl Catalyst REST
- PHP Symfony, Zend Framework, CakePHP, CodeIgniter, Sapphire, FRAPI
- Python django-piston, django-rest-interface django-restapi, web.py, lazr.restful
- REST microkernel and software platform NetKernel
- Ruby Ruby on Rails, Sinatra, Restfulie
- TurboGears2 provides a RestController

Outside of the Web

Software that may interact with a number of different kinds of objects or devices can do so by virtue of a uniform, agreed interface.

CMIP

The Common Management Information Protocol (CMIP) was designed to allow the control of network resources by presenting their manageable characteristics as object attributes. The objects have parent-child relationships that are identified using distinguished names and attributes, which are read and modified by a set of CRUD operations. The notable non-restful aspect of CMIP is the M_ACTION operation although, wherever possible, designers of management information bases (MIBs) would typically endeavour to represent controllable and stateful aspects of network equipment through attributes.