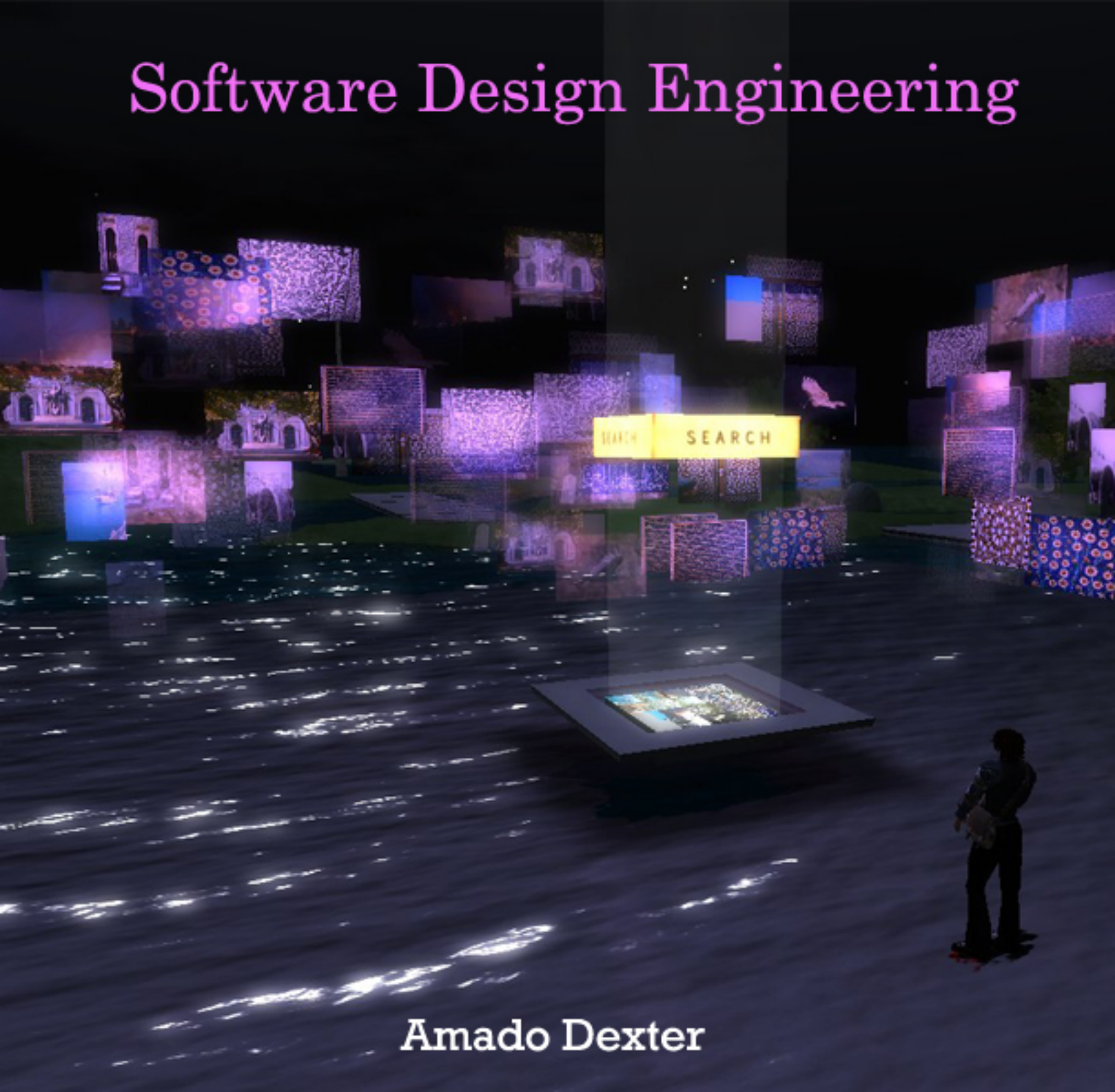
# Software Design Engineering




Amado Dexter

First Edition, 2012

ISBN 978-81-323-3102-5



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

# Table of Contents

# Introduction

**Software design** is a process of problem-solving and planning for a software solution. After the purpose and specifications of software are determined, software developers will design or employ designers to develop a plan for a solution. It includes low-level component and algorithm implementation issues as well as the architectural view.

## *Overview*

The software requirements analysis (SRA) step of a software development process yields specifications that are used in software engineering. If the software is "semiautomated" or user centered, software design may involve user experience design yielding a story board to help determine those specifications. If the software is completely automated (meaning no user or user interface), a software design may be as simple as a flow chart or text describing a planned sequence of events. There are also semi-standard methods like Unified Modeling Language and Fundamental modeling concepts. In either case some documentation of the plan is usually the product of the design.

A software design may be platform-independent or platform-specific, depending on the availability of the technology called for by the design.

## *Software Design Topics*

### Design concepts

The design concepts provide the software designer with a foundation from which more sophisticated methods can be applied. A set of fundamental design concepts has evolved. They are:

1. Abstraction - Abstraction is the process or result of generalization by reducing the information content of a concept or an observable phenomenon, typically in order to retain only information which is relevant for a particular purpose.
2. Refinement - It is the process of elaboration. A hierarchy is developed by decomposing a macroscopic statement of function in a stepwise fashion until programming language statements are reached. In each step, one or several

instructions of a given program are decomposed into more detailed instructions. Abstraction and Refinement are complementary concepts.
3. Modularity - Software architecture is divided into components called modules.
4. Software Architecture - It refers to the overall structure of the software and the ways in which that structure provides conceptual integrity for a system. A good software architecture will yield a good return on investment with respect to the desired outcome of the project, e.g. in terms of performance, quality, schedule and cost.
5. Control Hierarchy - A program structure that represent the organization of a program components and implies a hierarchy of control.
6. Structural Partitioning - The program structure can be divided both horizontally and vertically. Horizontal partitions define separate branches of modular hierarchy for each major program function. Vertical partitioning suggests that control and work should be distributed top down in the program structure.
7. Data Structure - It is a representation of the logical relationship among individual elements of data.
8. Software Procedure - It focuses on the processing of each modules individually
9. Information Hiding - Modules should be specified and designed so that information contained within a module is inaccessible to other modules that have no need for such information.

## Design considerations

There are many aspects to consider in the design of a piece of software. The importance of each should reflect the goals the software is trying to achieve. Some of these aspects are:

- **Compatibility** - The software is able to operate with other products that are designed for interoperability with another product. For example, a piece of software may be backward-compatible with an older version of itself.
- **Extensibility** - New capabilities can be added to the software without major changes to the underlying architecture.
- **Fault-tolerance** - The software is resistant to and able to recover from component failure.
- **Maintainability** - The software can be restored to a specified condition within a specified period of time. For example, antivirus software may include the ability to periodically receive virus definition updates in order to maintain the software's effectiveness.
- **Modularity** - the resulting software comprises well defined, independent components. That leads to better maintainability. The components could be then implemented and tested in isolation before being integrated to form a desired software system. This allows division of work in a software development project.
- **Packaging** - Printed material such as the box and manuals should match the style designated for the target market and should enhance usability. All compatibility information should be visible on the outside of the package. All components

required for use should be included in the package or specified as a requirement on the outside of the package.
- **Reliability** - The software is able to perform a required function under stated conditions for a specified period of time.
- **Reusability** - the software is able to add further features and modification with slight or no modification.
- **Robustness** - The software is able to operate under stress or tolerate unpredictable or invalid input. For example, it can be designed with a resilience to low memory conditions.
- **Security** - The software is able to withstand hostile acts and influences.
- **Usability** - The software user interface must be usable for its target user/audience. Default values for the parameters must be chosen so that they are a good choice for the majority of the users.

## Modeling language

A modeling language is any artificial language that can be used to express information or knowledge or systems in a structure that is defined by a consistent set of rules. The rules are used for interpretation of the meaning of components in the structure. A modeling language can be graphical or textual. Examples of graphical modelling languages for software design are:

- Business Process Modeling Notation (BPMN) is an example of a Process Modeling language.
- EXPRESS and EXPRESS-G (ISO 10303-11) is an international standard general-purpose data modeling language.
- Extended Enterprise Modeling Language (EEML) is commonly used for business process modeling across a number of layers.
- Flowchart is a schematic representation of an algorithm or a stepwise process,
- Fundamental Modeling Concepts (FMC) modeling language for software-intensive systems.
- IDEF is a family of modeling languages, the most notable of which include IDEF0 for functional modeling, IDEF1X for information modeling, and IDEF5 for modeling ontologies.
- Jackson Structured Programming (JSP) is a method for structured programming based on correspondences between data stream structure and program structure
- LePUS3 is an object-oriented visual Design Description Language and a formal specification language that is suitable primarily for modelling large object-oriented (Java, C++, C#) programs and design patterns.
- Unified Modeling Language (UML) is a general modeling language to describe software both structurally and behaviorally. It has a graphical notation and allows for extension with a Profile (UML).
- Alloy (specification language) is a general purpose specification language for expressing complex structural constraints and behavior in a software system. It provides a concise language based on first-order relational logic.

- Systems Modeling Language (SysML) is a new general-purpose modeling language for systems engineering.

## Design patterns

A software designer or architect may identify a design problem which has been solved by others before. A template or pattern describing a solution to a common problem is known as a design pattern. The reuse of such patterns can speed up the software development process, having been tested and proved in the past.

## Usage

Software design documentation may be reviewed or presented to allow constraints, specifications and even requirements to be adjusted prior to programming. Redesign may occur after review of a programmed simulation or prototype. It is possible to design software in the process of programming, without a plan or requirement analysis, but for more complex projects this would not be considered a professional approach. A separate design prior to programming allows for multidisciplinary designers and Subject Matter Experts (SMEs) to collaborate with highly-skilled programmers for software that is both useful and technically sound.

**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 | No | No | N/A |

| | | | | |
|---|---|---|---|---|
| | pool and thread pool patterns. | | | |
| Prototype | Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype. | Yes | No | N/A |
| 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 | No | Yes | N/A |

| | | | | |
|---|---|---|---|---|
| | makes the subsystem easier to use. | | | |
| Flyweight | Use sharing to support large numbers of fine-grained objects efficiently. | Yes | No | N/A |
| 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 the object to be restored to this state later. | Yes | No | N/A |
| 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 control. The goal is to introduce concurrency, by using asynchronous method invocation and a scheduler for handling requests. | Yes | N/A |
| 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.<br><br>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*

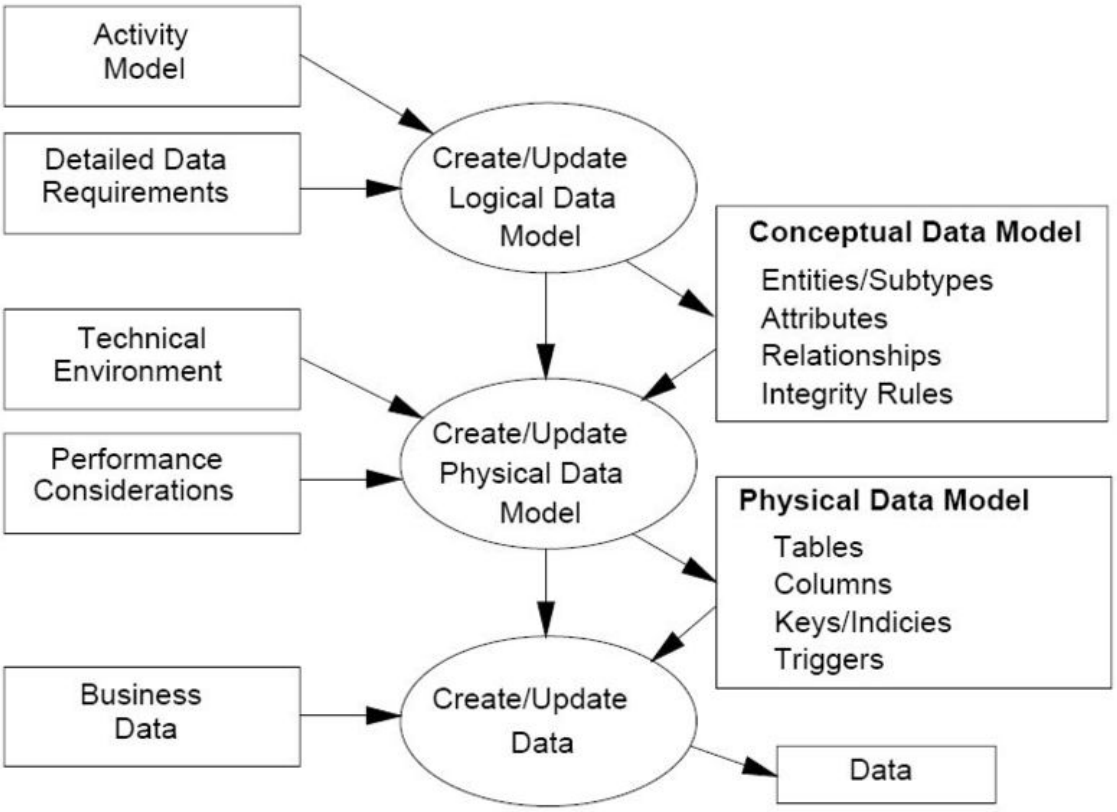
There are many common ways of designing computer software modules and their communications, among them:

- 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)

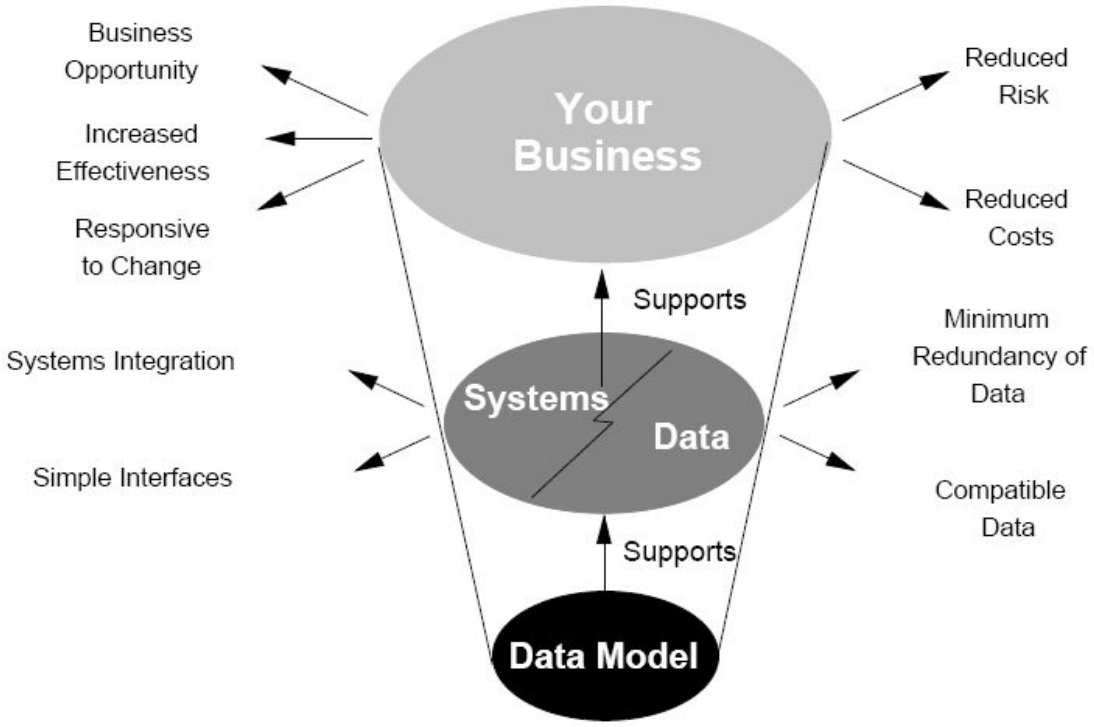
Data modeling may be performed during various types of projects and in multiple phases of projects. Data models are progressive; there is no such thing as the final data model for a business or application. Instead a data model should be considered a living document that will change in response to a 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.

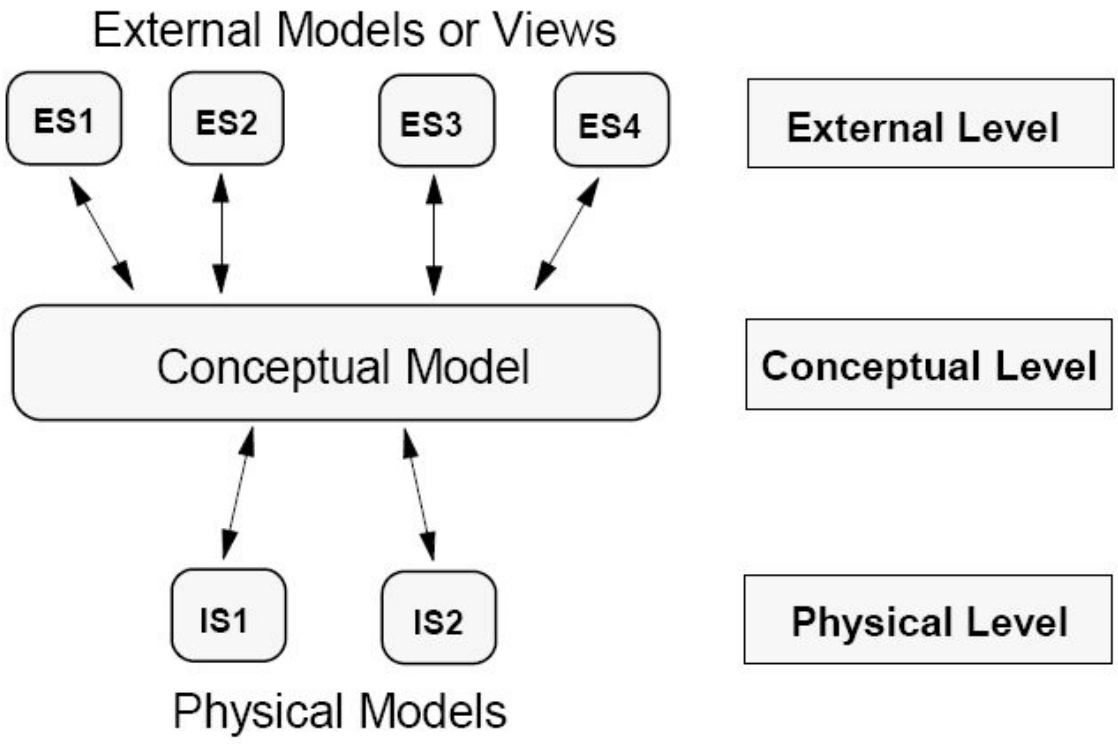
Data models support data and computer systems by providing the definition and format of data. If this is done consistently across systems then compatibility of data can be achieved. If the same data structures are used to store and access data then different applications can share data. The results of 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.

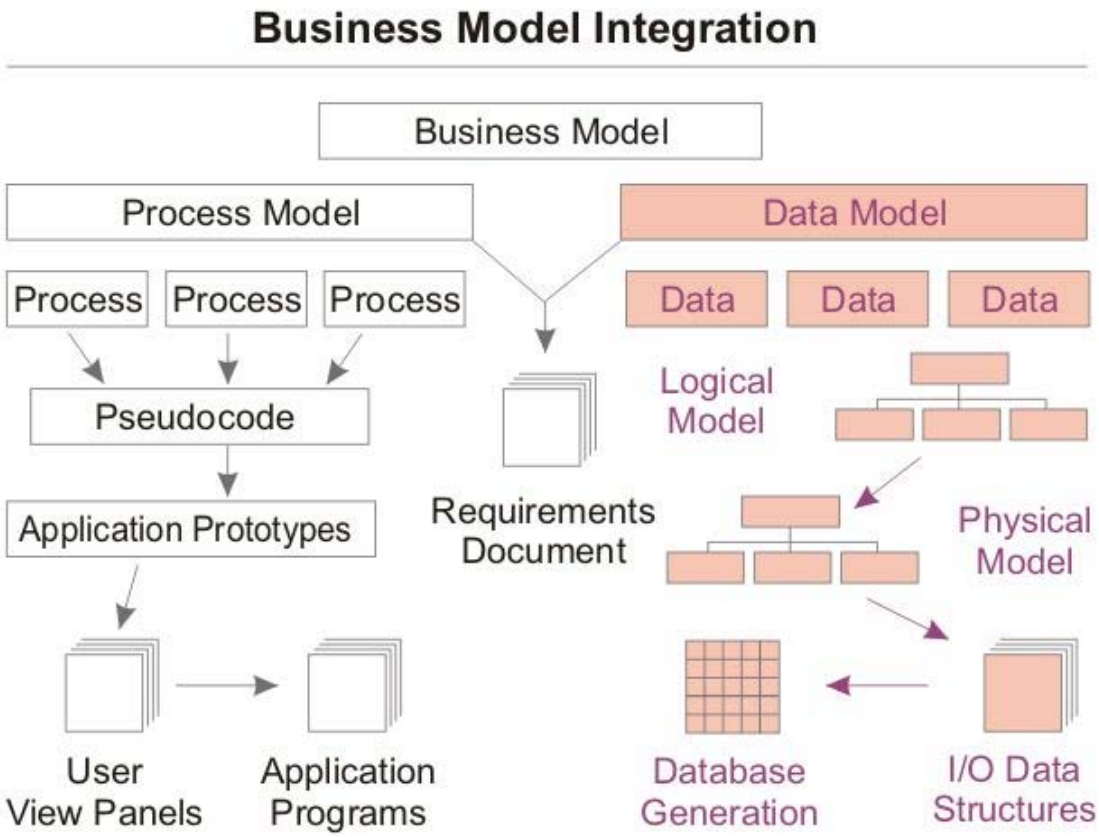
A data model *instance* may be one of three kinds according to ANSI in 1975:

- 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

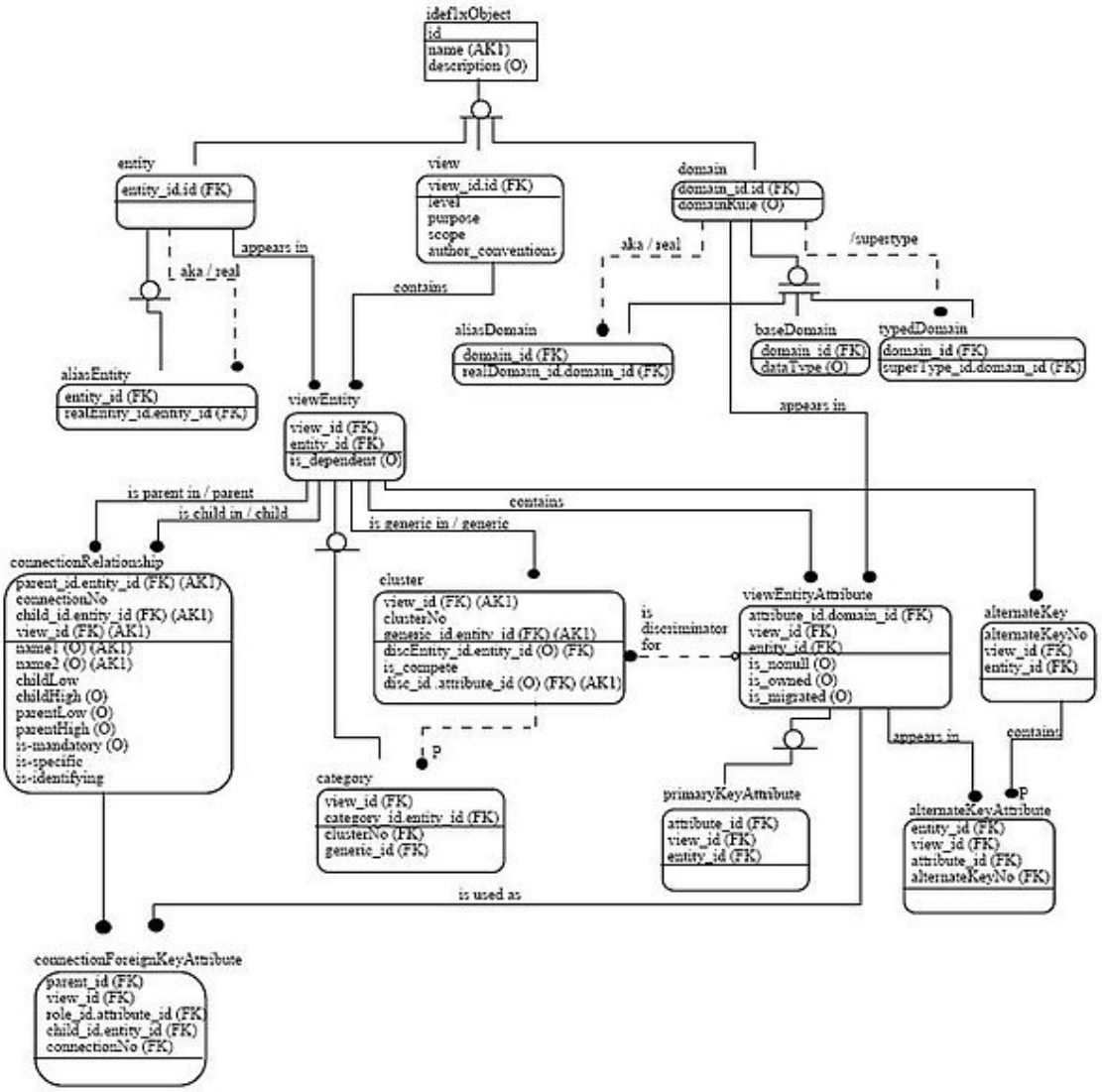
Data models represent information areas of interest. While there are many ways to create data models, according to Len Silverston (1997) only two modeling methodologies stand out, top-down and bottom-up:

- 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.

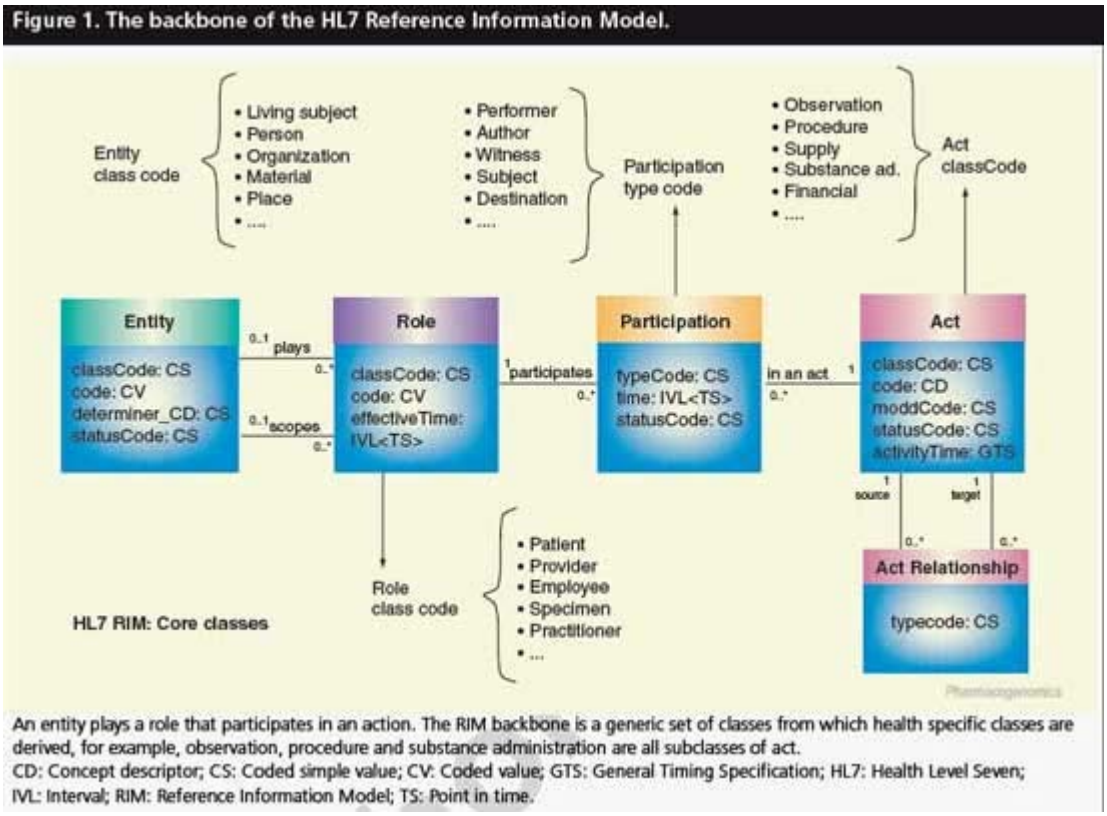
There are several notations for data modeling. The actual model is frequently called "Entity relationship model", because it depicts data in terms of the entities and relationships described in the data. An entity-relationship model (ERM) is an abstract conceptual representation of structured data. Entity-relationship 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



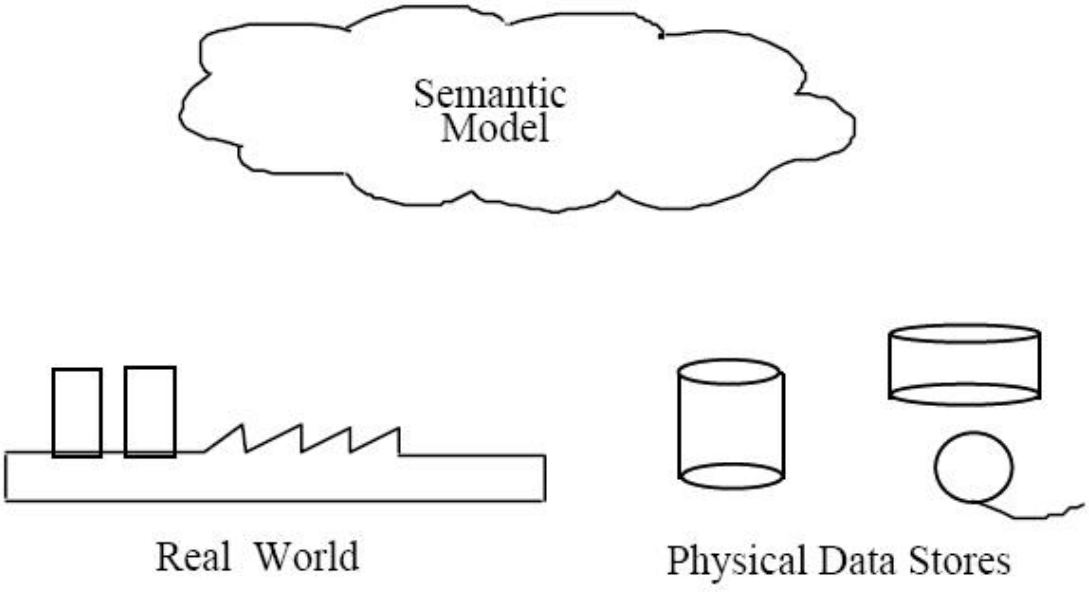

Example of a Generic data model.

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.

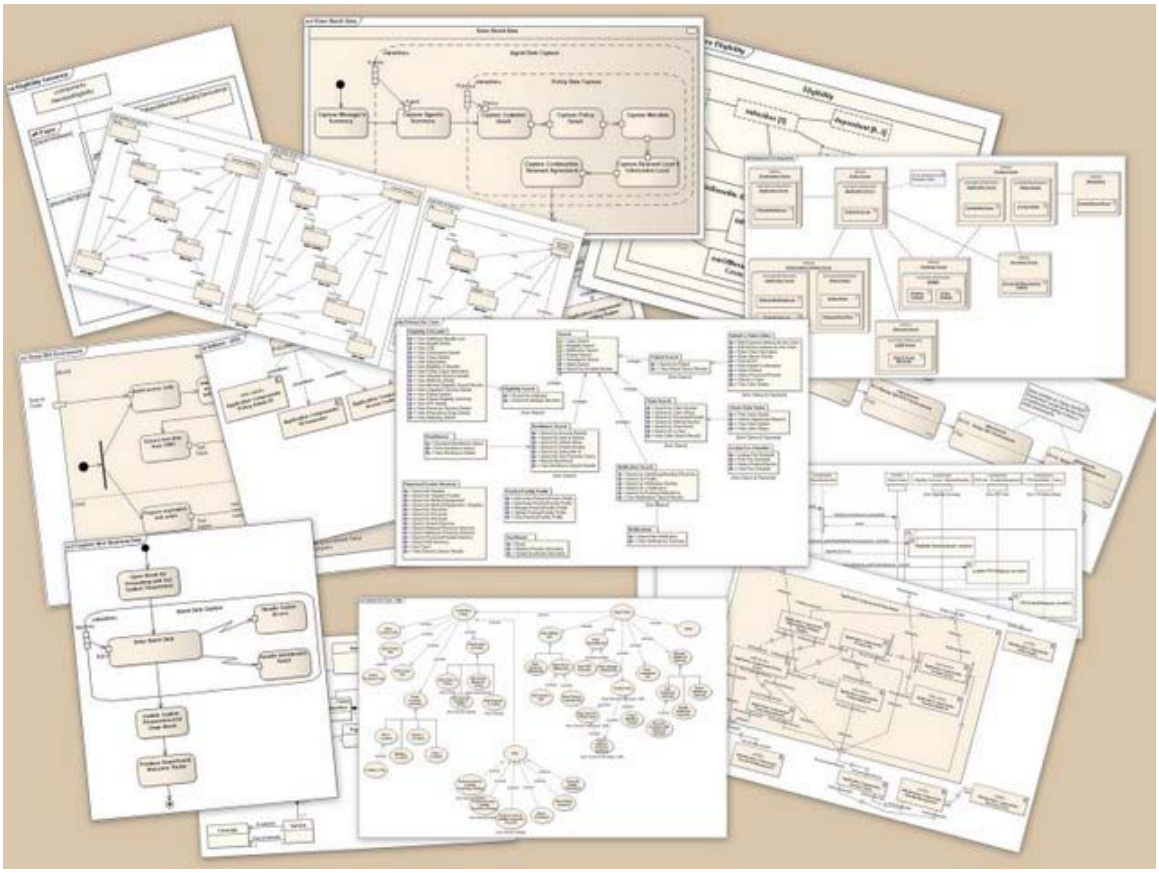
A semantic data model can be used to serve many purposes, such as:.

- 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*

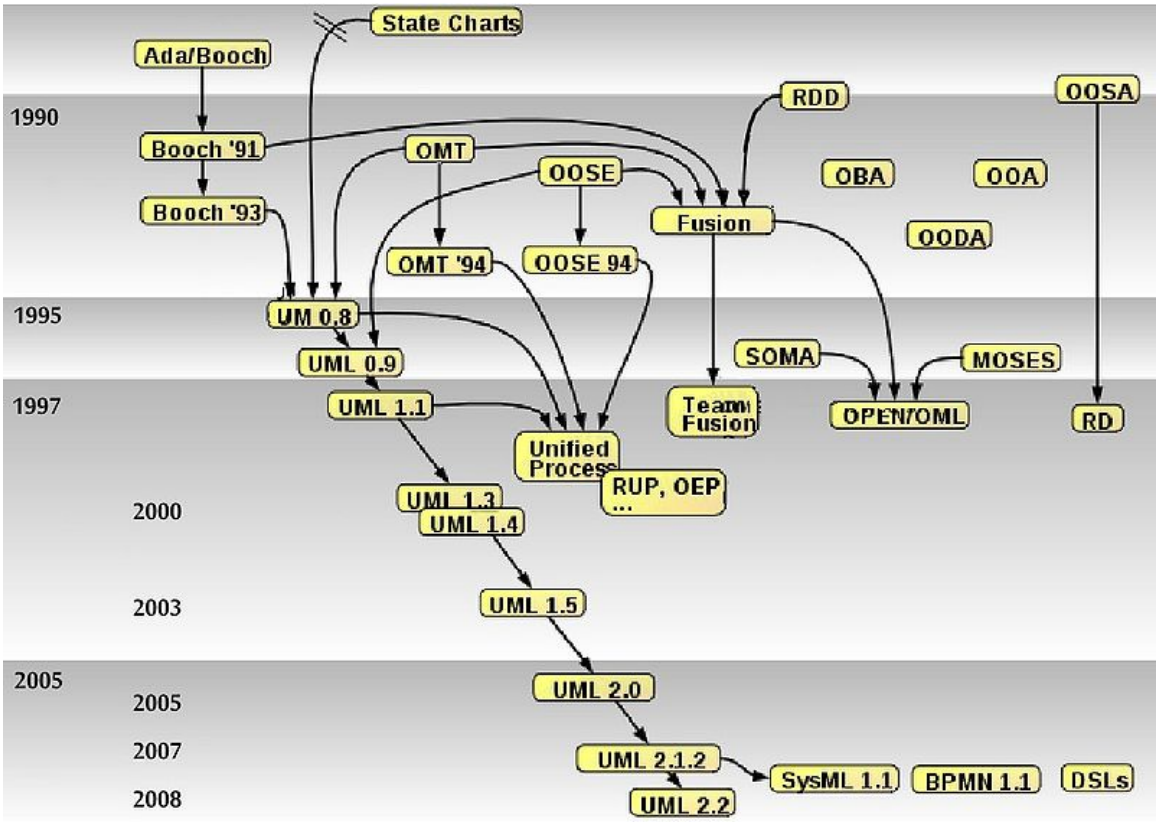
The Unified Modeling Language (UML) is used to specify, visualize, modify, construct and document the artifacts of an object-oriented software-intensive system under development. UML offers a standard way to visualize a system's architectural blueprints, including elements such as:

- 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:

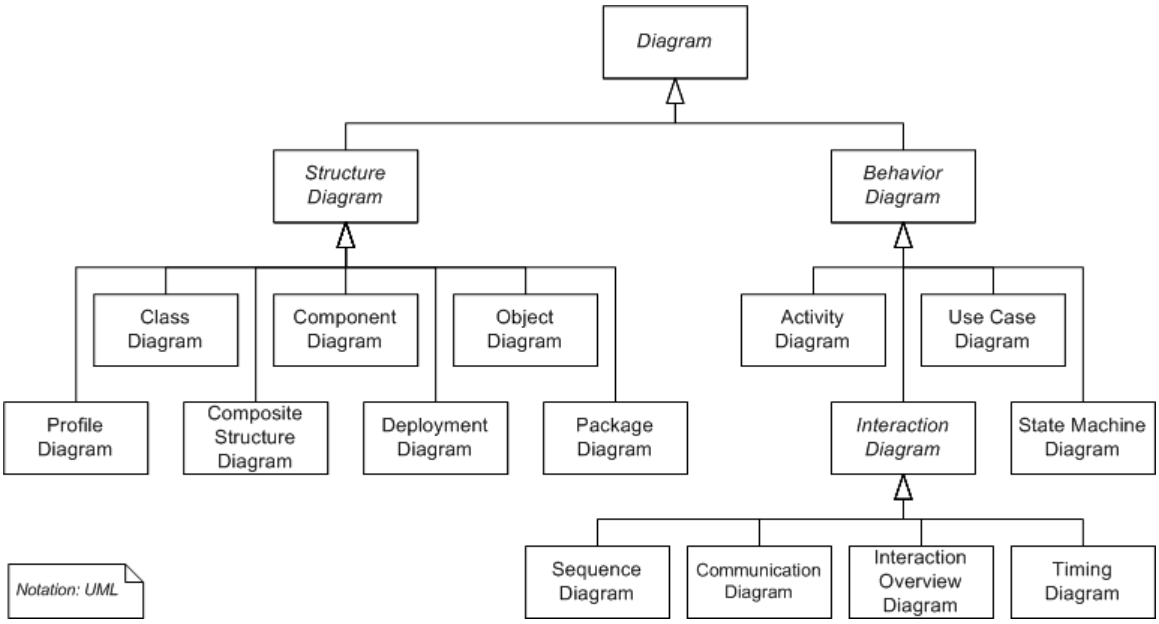
- Static (or *structural*) view: emphasizes the static structure of the system using objects, attributes, operations and relationships. The structural view includes class diagrams and composite structure diagrams.
- Dynamic (or *behavioral*) view: emphasizes the dynamic behavior of the system by showing 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.

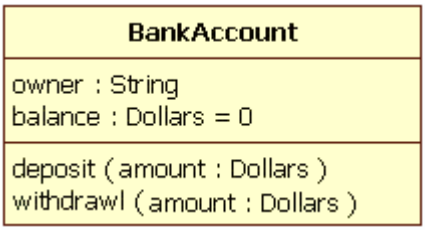
In keeping with the tradition of engineering drawings, a comment or note explaining usage, constraint, or intent is allowed in a UML diagram.

## Structure diagrams

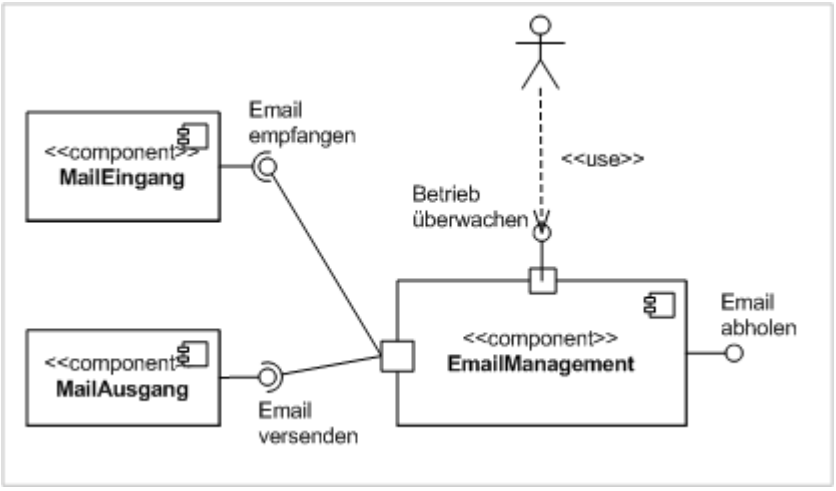
Structure diagrams emphasize the things that must be present in the system being modeled. Since structure diagrams represent the structure, they are used extensively in documenting the software architecture of software systems.

- 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.

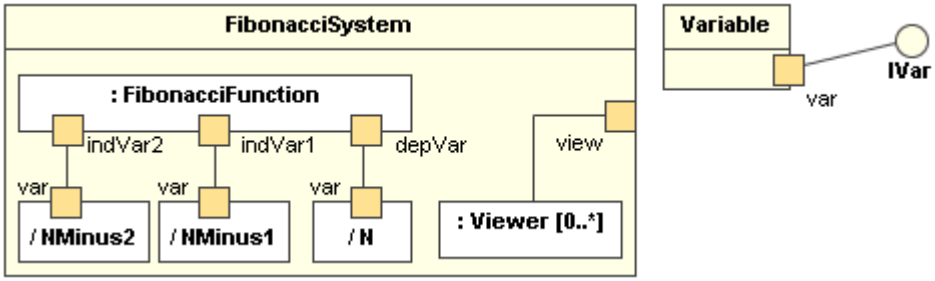
- Profile diagram: operates at the metamodel level to show stereotypes as classes with the <<stereotype>> stereotype, and profiles as packages with the <<profile>> stereotype. The extension relation (solid line with closed, filled arrowhead) indicates what metamodel element a given stereotype is extending.



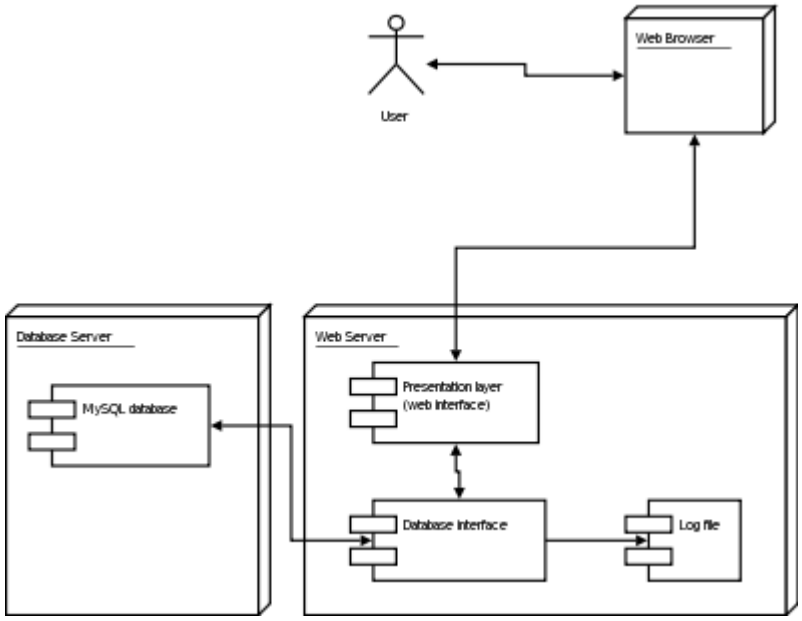

Class diagram



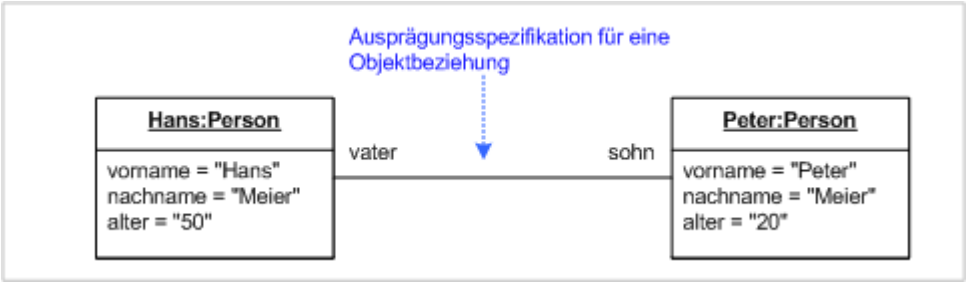

Component diagram



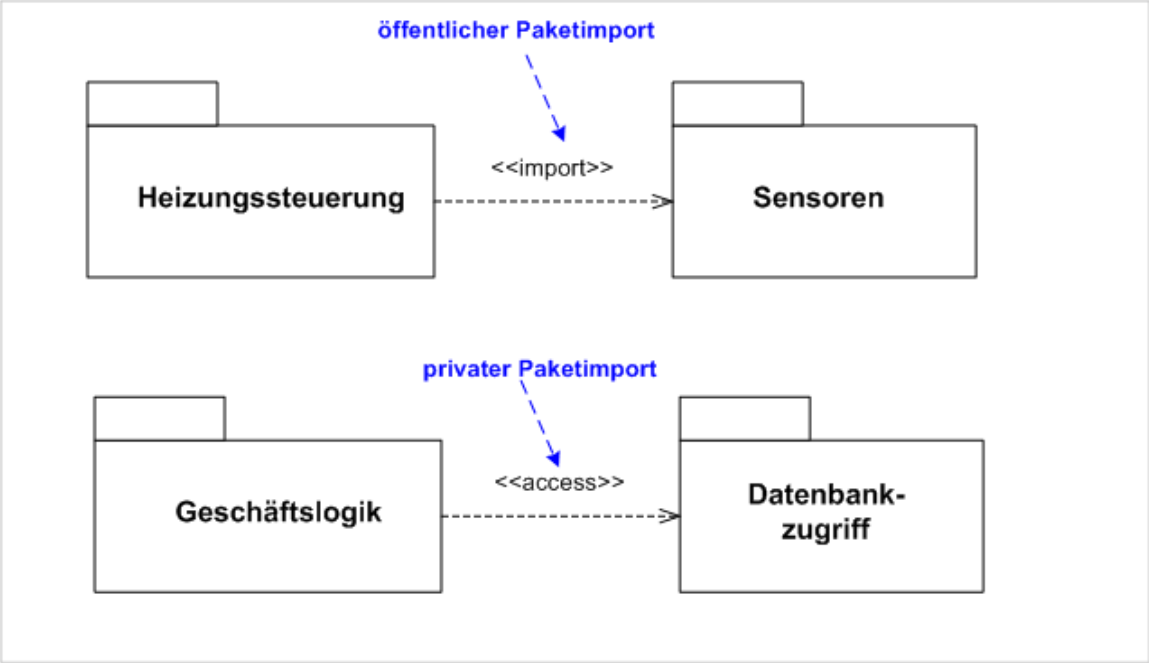

Composite structure diagrams




Deployment diagram




Object diagram

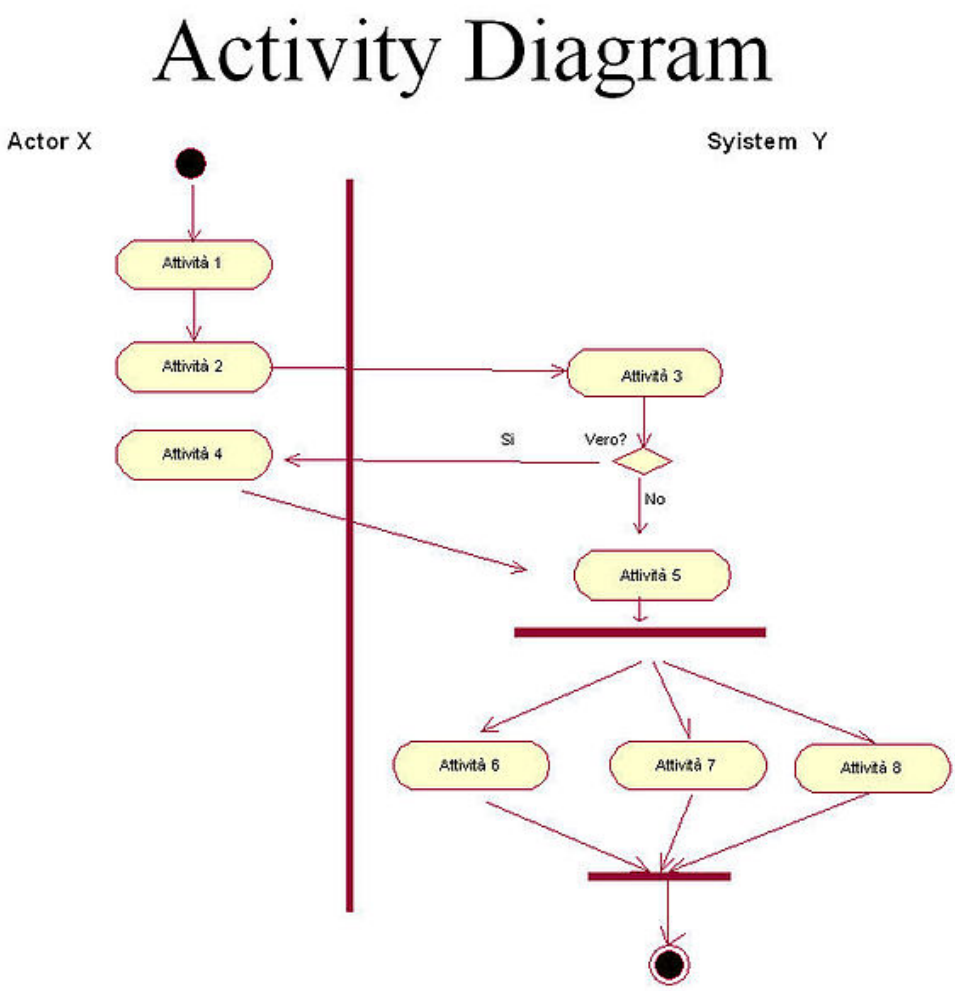



Package diagram

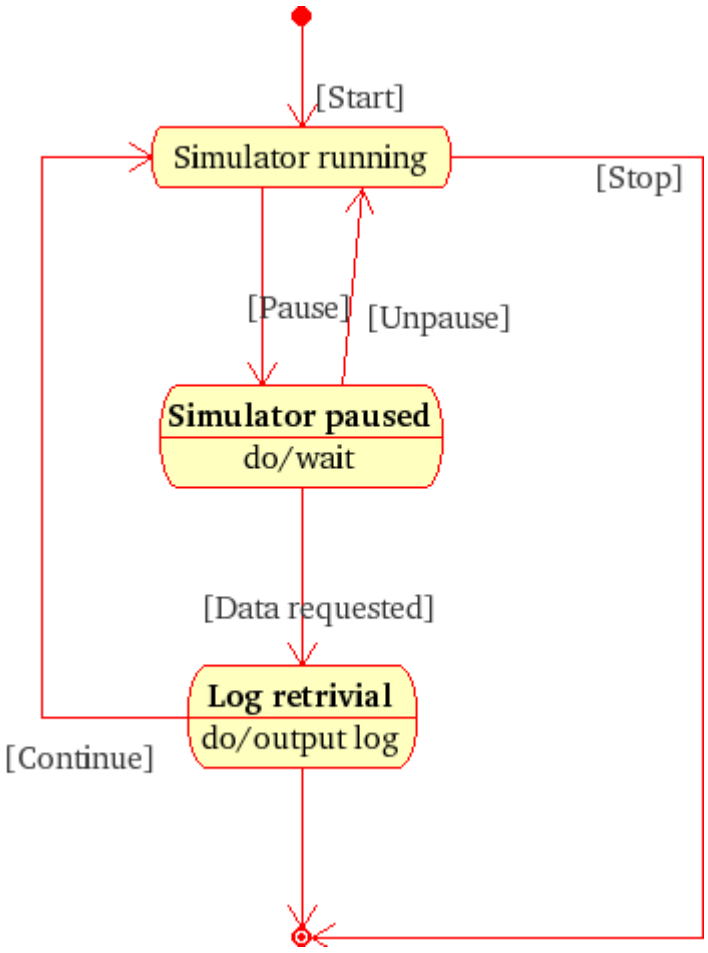
## Behaviour diagrams

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.

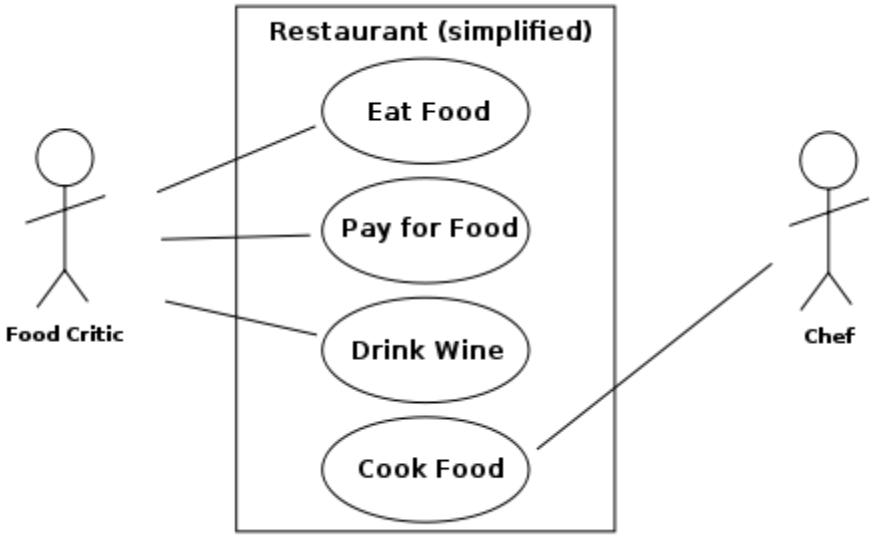
- Activity diagram: describes the business and operational step-by-step workflows of components in a system. An activity diagram shows the overall flow of control.
- UML state machine diagram: describes the states and state transitions of the system.
- Use case diagram: describes the functionality provided 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

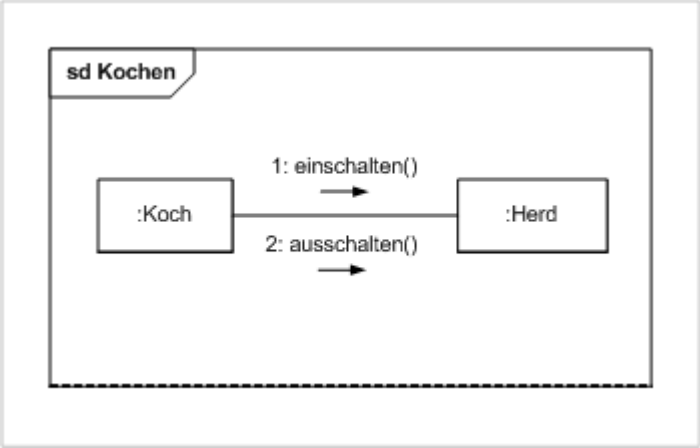



Use case diagram

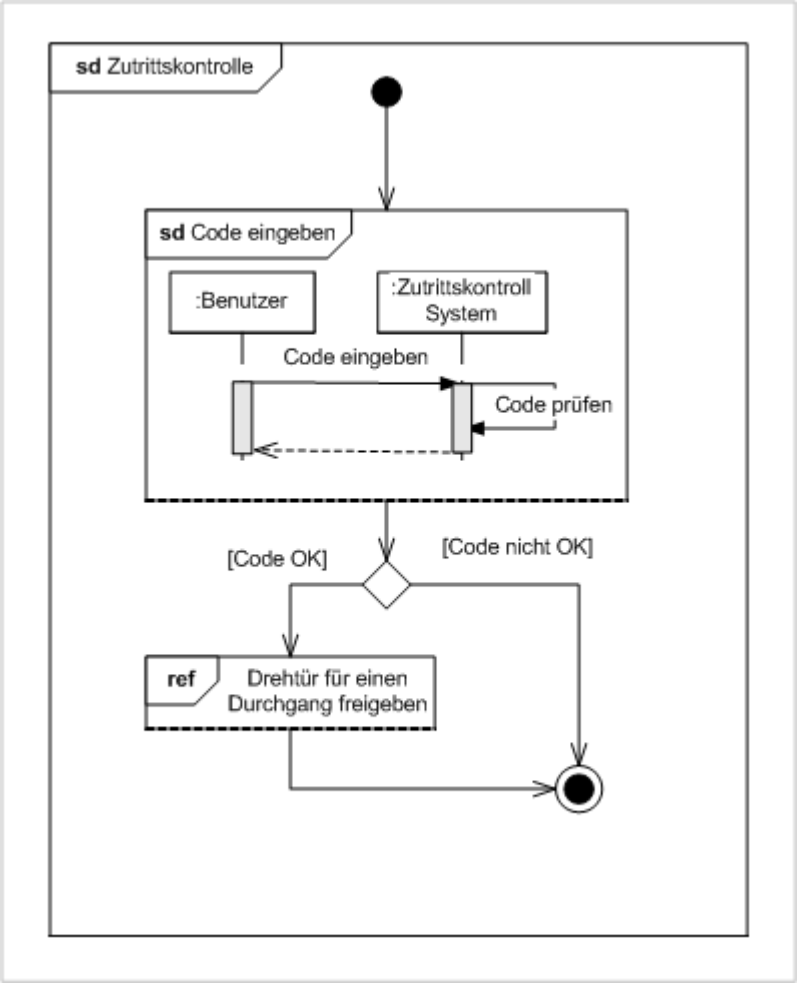
## Interaction diagrams

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

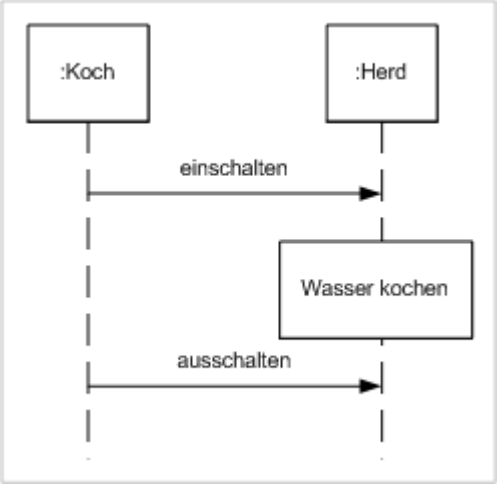
- Communication diagram: shows the interactions between objects or parts in terms of sequenced messages. They represent a combination of information taken from Class, Sequence, and Use Case Diagrams describing both the static structure and dynamic behavior of a system.
- Interaction overview diagram: 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

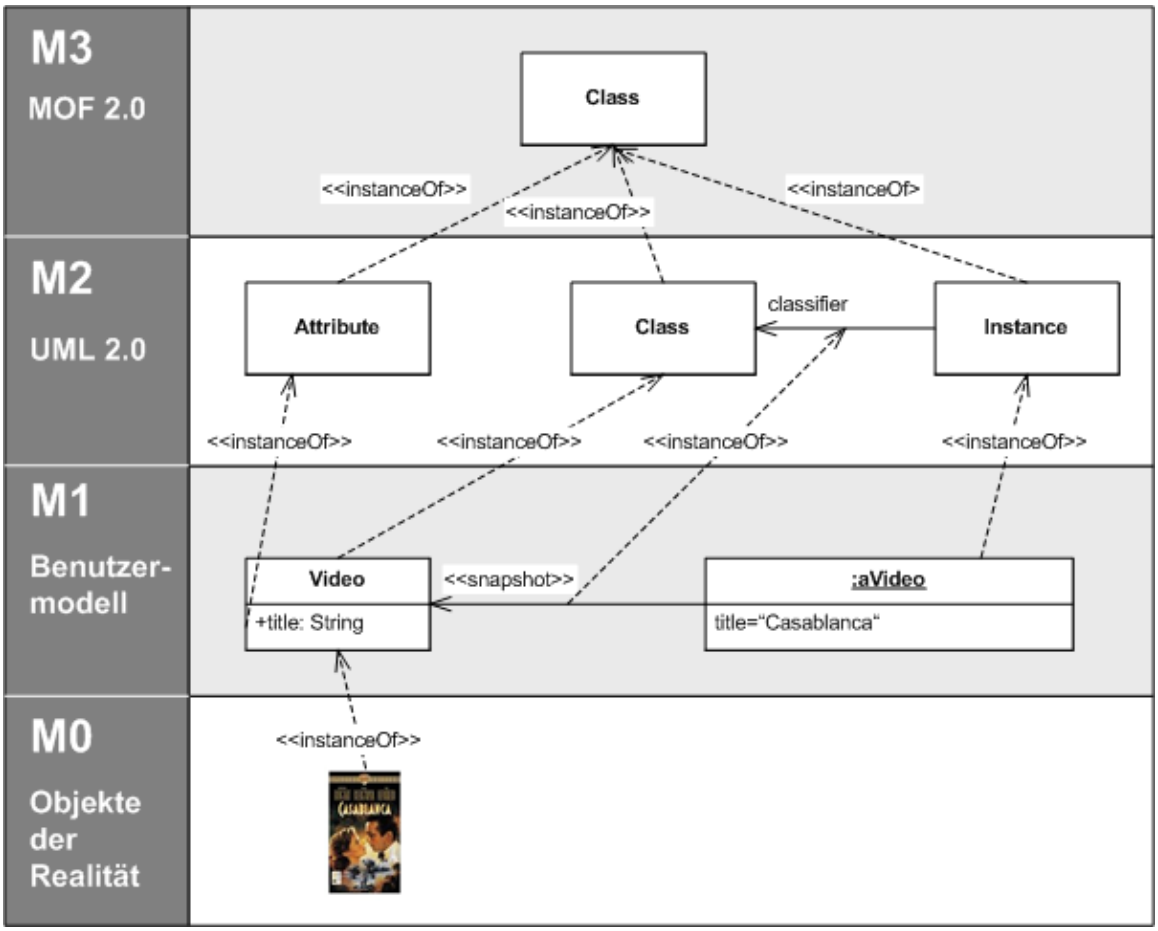



Illustration of the Meta-Object Facility.

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.

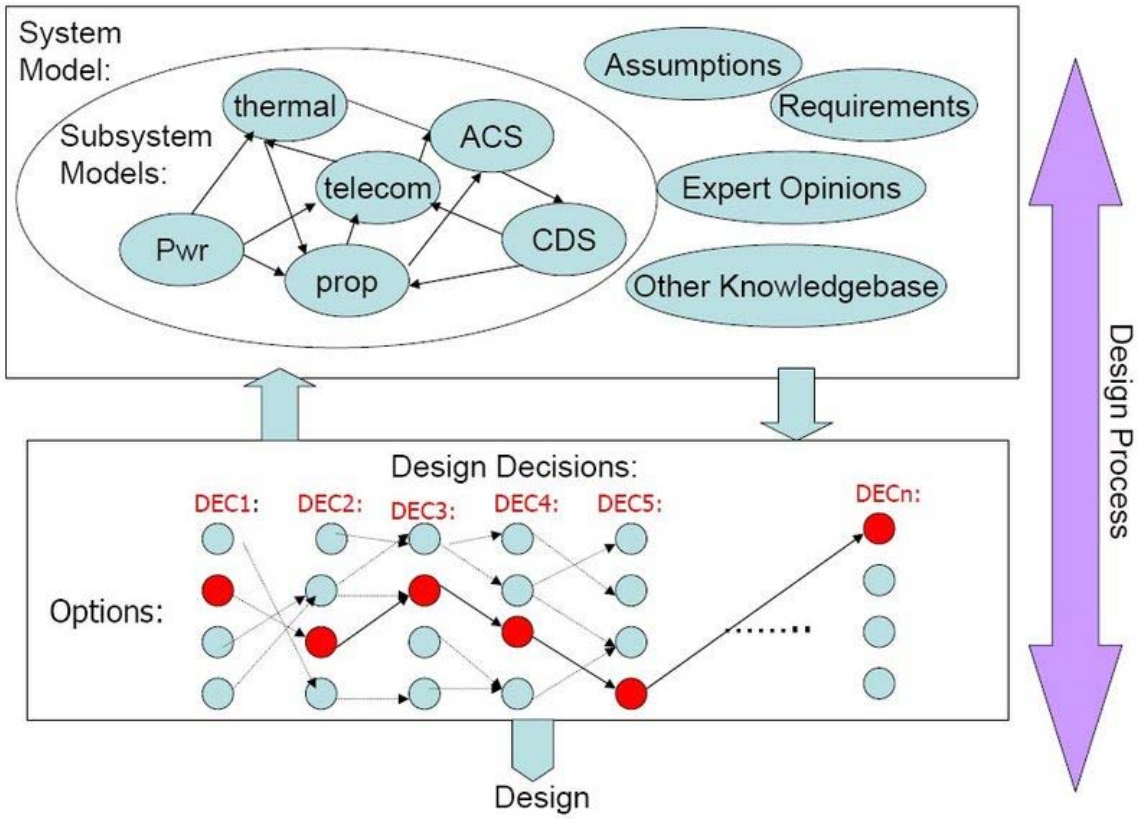
Modeling experts have written sharp criticisms of UML, including Bertrand Meyer's "UML: The Positive Spin", and Brian Henderson-Sellers and Cesar Gonzalez-Perez in "Uses and Abuses of the Stereotype Mechanism in UML 1.x and 2.0".

## *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

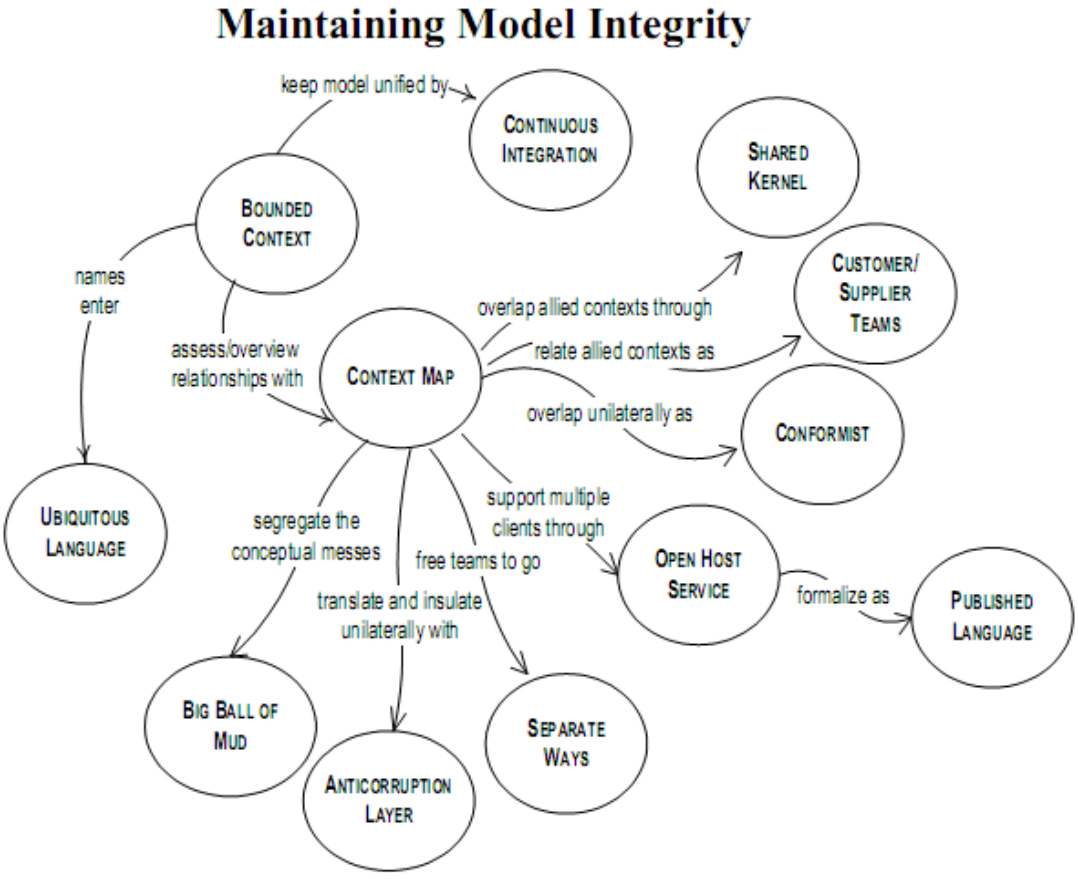
- You have access to domain experts
- You have an iterative process

## *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 aggregate 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

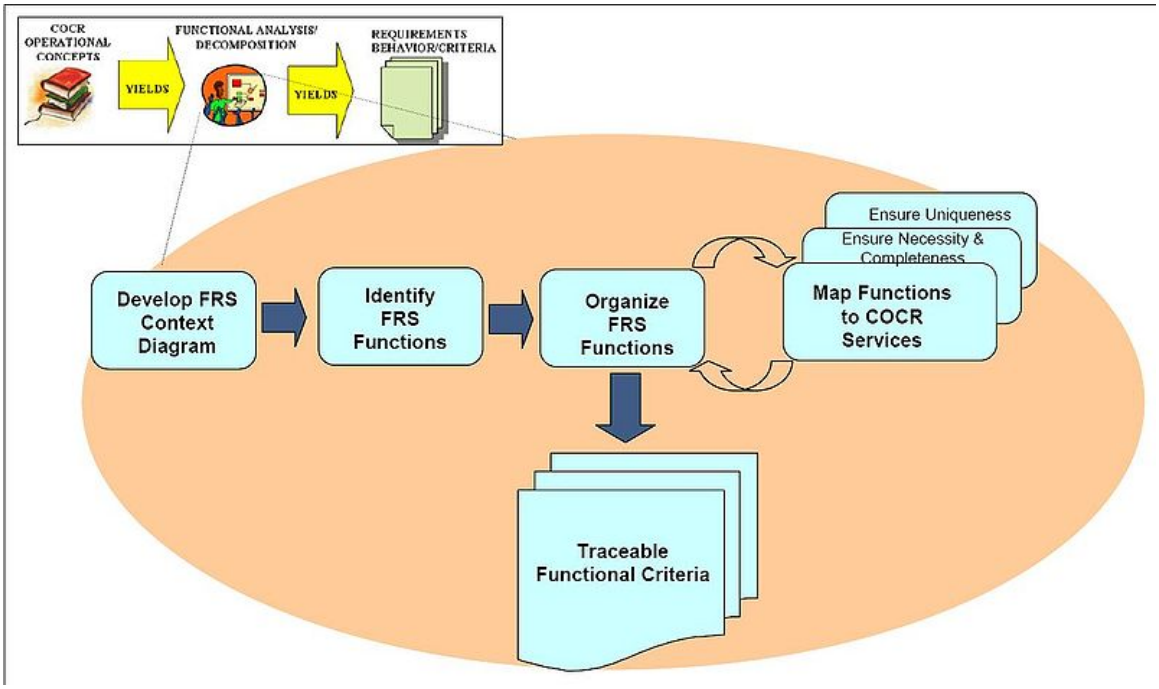
identified through the development of the sequence diagrams can serve as input to the automatic generation of the global class diagram of the system.

## 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

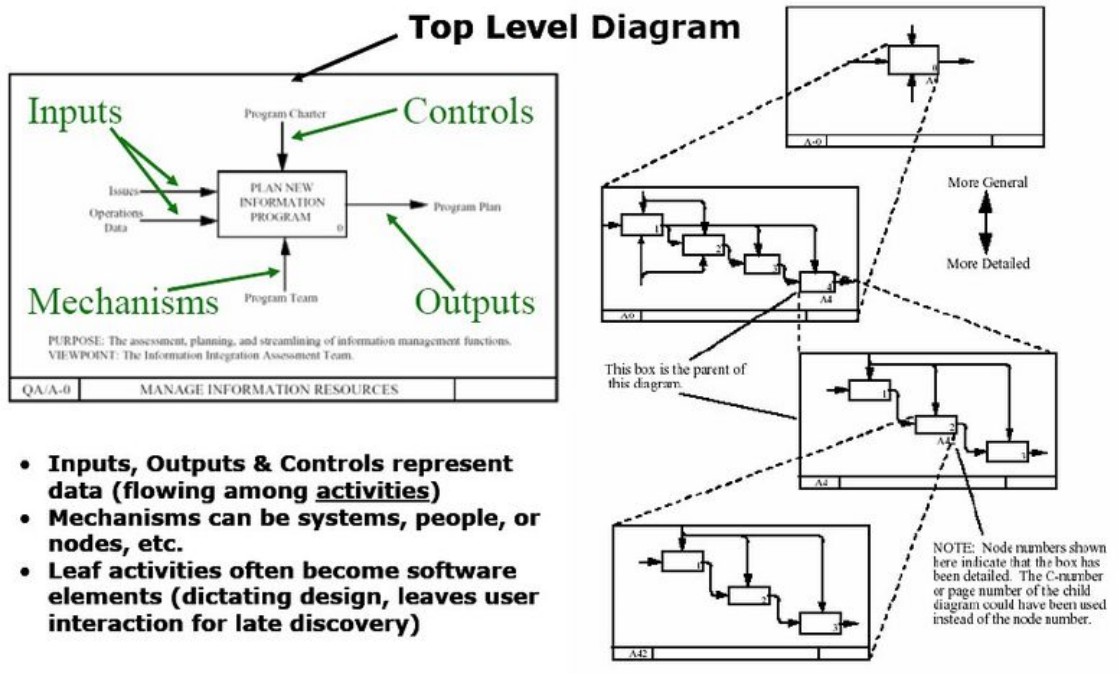



Image:Structured Analysis example.

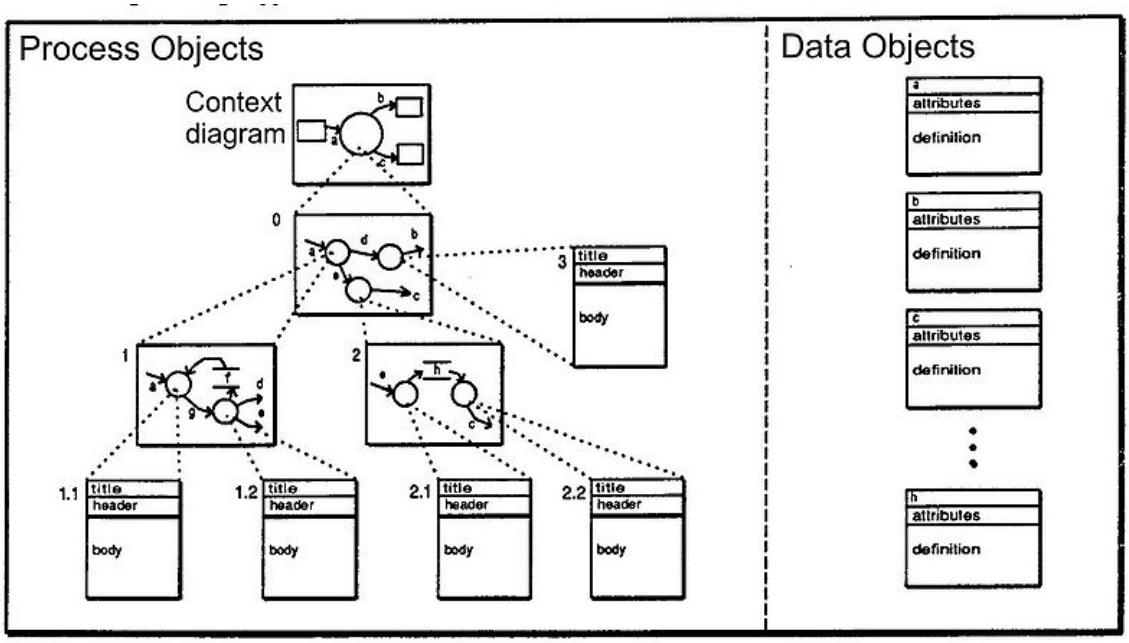
Structured analysis typically creates a hierarchy employing a single abstraction mechanism. The structured analysis method can employ IDEF (see figure), is process

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.



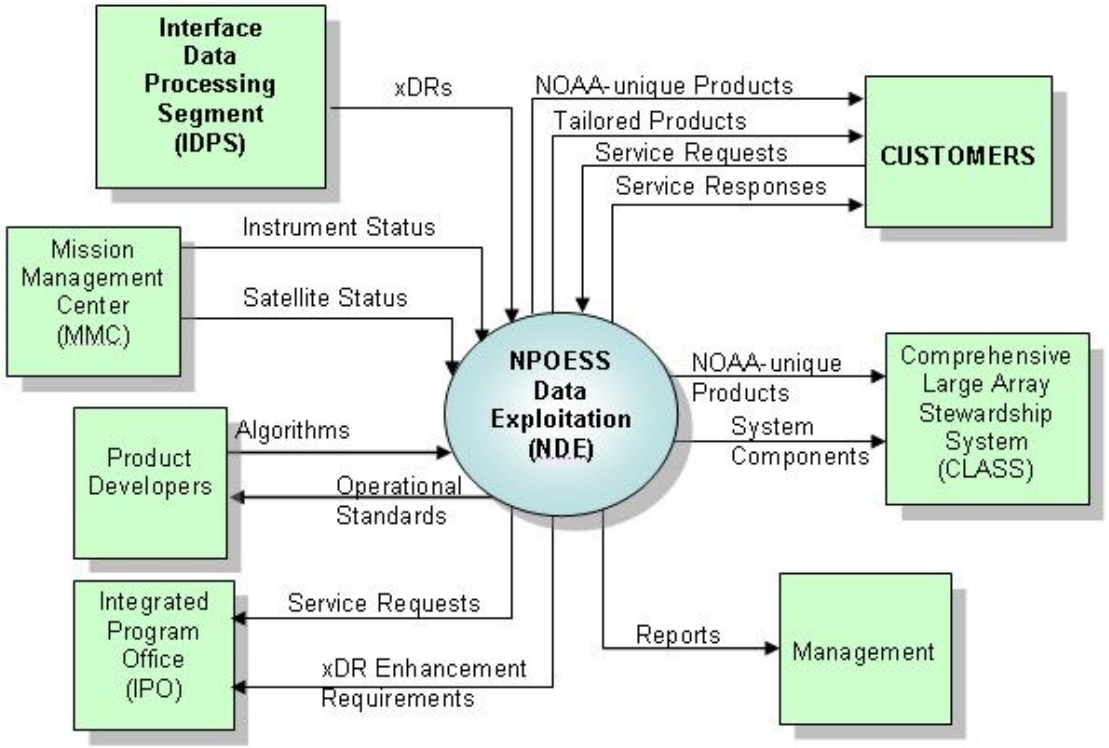
The structured analyse approach develops perspectives on both process objects and data objects.

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

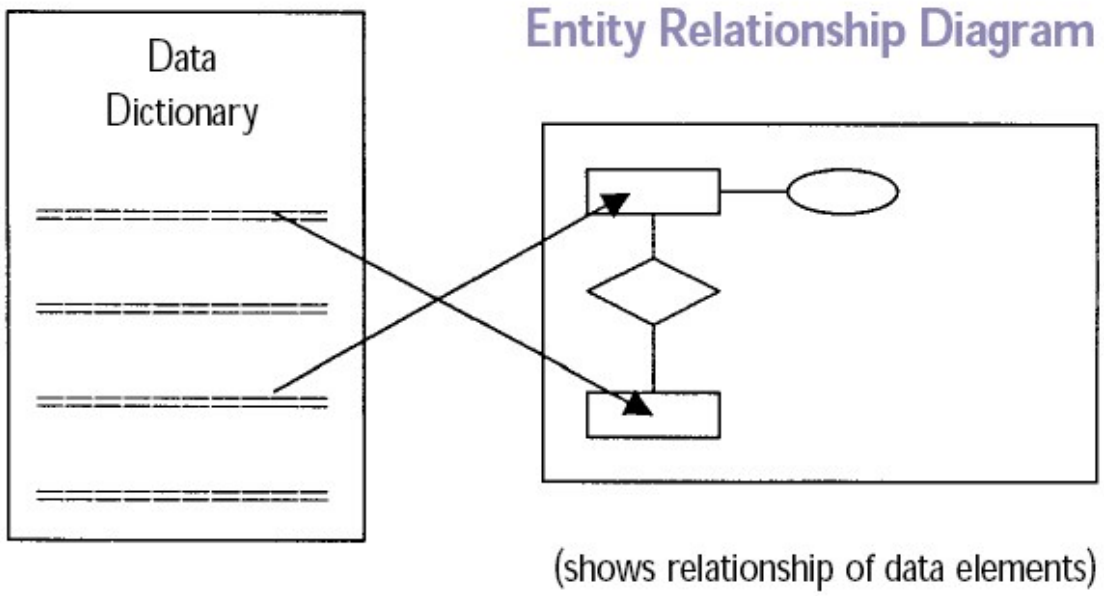


Example of a System context diagram.

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

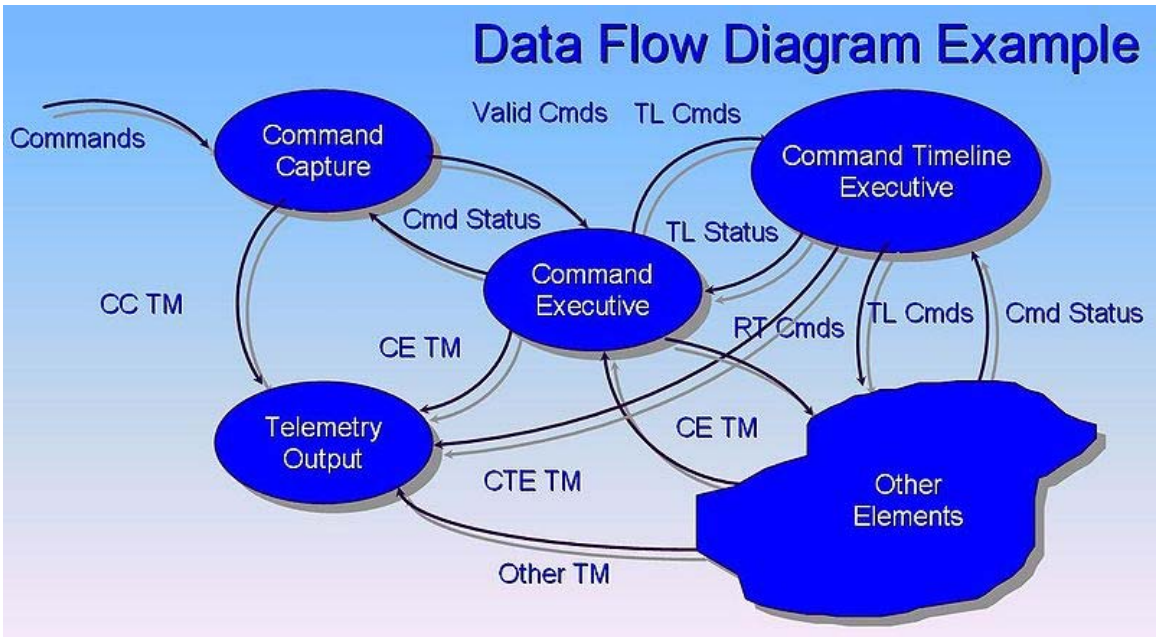



Entity relationship diagram, essential for the design of database tables, extracts, and metadata.

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.

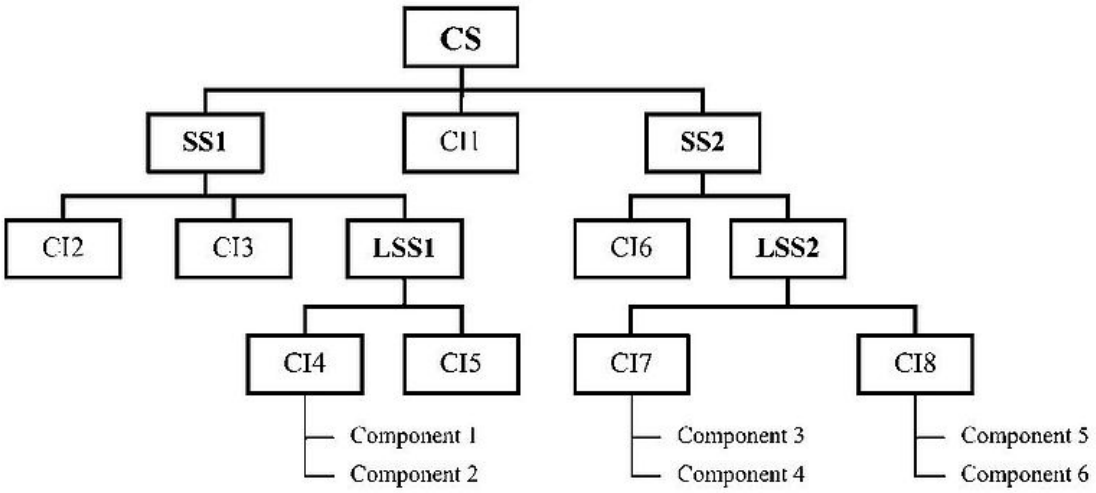
A Data Flow Diagram (DFD) is a graphical representation of the "flow" of data through an information system. It differs from the system flowchart as it shows the flow of data through processes instead of hardware. Data flow diagrams were invented by Larry Constantine, developer of structured design,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

# Top-Down and Bottom-Up Design

**Top-down** and **bottom-up** are strategies of information processing and knowledge ordering, mostly involving software, but also other humanistic and scientific theories. In practice, they can be seen as a style of thinking and teaching. In many cases *top-down* is used as a synonym of *analysis* or *decomposition*, and *bottom-up* of *synthesis*.

A **top-down** approach (also known as step-wise design) is essentially the breaking down of a system to gain insight into its compositional sub-systems. In a top-down approach an overview of the system is formulated, specifying but not detailing any first-level subsystems. Each subsystem is then refined in yet greater detail, sometimes in many additional subsystem levels, until the entire specification is reduced to base elements. A top-down model is often specified with the assistance of "black boxes", these make it easier to manipulate. However, black boxes may fail to elucidate elementary mechanisms or be detailed enough to realistically validate the model.

A **bottom-up** approach is the piecing together of systems to give rise to grander systems, thus making the original systems sub-systems of the emergent system. In a bottom-up approach the individual base elements of the system are first specified in great detail. These elements are then linked together to form larger subsystems, which then in turn are linked, sometimes in many levels, until a complete top-level system is formed. This strategy often resembles a "seed" model, whereby the beginnings are small but eventually grow in complexity and completeness. However, "organic strategies" may result in a tangle of elements and subsystems, developed in isolation and subject to local optimization as opposed to meeting a global purpose.

## *Computer science*

### Software development

In the software development process, the **top-down** and **bottom-up** approaches play a key role.

Top-down approaches emphasize planning and a complete understanding of the system. It is inherent that no coding can begin until a sufficient level of detail has been reached in the design of at least some part of the system. The Top-Down Approach is done by attaching the stubs in place of the module. This, however, delays testing of the ultimate functional units of a system until significant design is complete. Bottom-up emphasizes coding and early testing, which can begin as soon as the first module has been specified. This approach, however, runs the risk that modules may be coded without having a clear idea of how they link to other parts of the system, and that such linking may not be as easy as first thought. Re-usability of code is one of the main benefits of the bottom-up approach.

Top-down design was promoted in the 1970s by IBM researcher Harlan Mills and Niklaus Wirth. Mills developed structured programming concepts for practical use and tested them in a 1969 project to automate the *New York Times* morgue index. The engineering and management success of this project led to the spread of the top-down approach through IBM and the rest of the computer industry. Among other achievements, Niklaus Wirth, the developer of Pascal programming language, wrote the influential paper *Program Development by Stepwise Refinement*. Since Niklaus Wirth went on to develop languages such as Modula and Oberon (where one could define a module before knowing about the entire program specification), one can infer that top down programming was not strictly what he promoted. Top-down methods were favored in software engineering until the late 1980s, and object-oriented programming assisted in demonstrating the idea that both aspects of top-down and bottom-up programming could be utilized.

Modern software design approaches usually combine both top-down and bottom-up approaches. Although an understanding of the complete system is usually considered necessary for good design, leading theoretically to a top-down approach, most software projects attempt to make use of existing code to some degree. Pre-existing modules give designs a bottom-up flavour. Some design approaches also use an approach where a partially-functional system is designed and coded to completion, and this system is then expanded to fulfill all the requirements for the project

## Programming

## Top-down approach

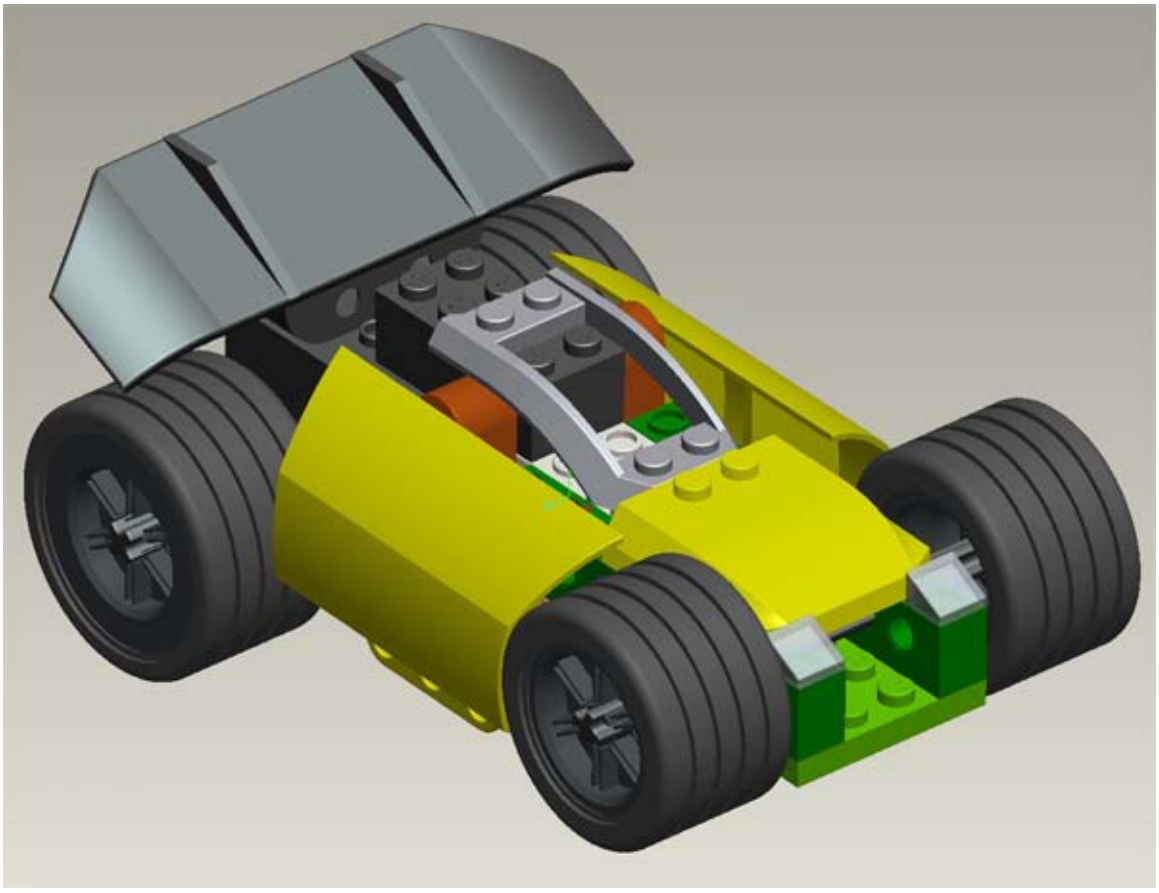
Top-down is a programming style, the mainstay of traditional procedural languages, in which design begins by specifying complex pieces and then dividing them into successively smaller pieces.

The technique for writing a program using top-down methods is to write a main procedure that names all the major functions it will need. Later, the programming team looks at the requirements of each of those functions and the process is repeated. These compartmentalized sub-routines eventually will perform actions so simple they can be

easily and concisely coded. When all the various sub-routines have been coded the program is ready for testing.

By defining how the application comes together at a high level, lower level work can be self-contained. By defining how the lower level abstractions are expected to integrate into higher level ones, interfaces become clearly defined.

## Bottom-up approach



Lego Racer Pro/ENGINEER Parts is a good example of bottom-up design because the parts are first created and then assembled without regard to how the parts will work in the assembly.

In a bottom-up approach the individual base elements of the system are first specified in great detail. These elements are then linked together to form larger subsystems, which then in turn are linked, sometimes in many levels, until a complete top-level system is formed. This strategy often resembles a "seed" model, whereby the beginnings are small, but eventually grow in complexity and completeness.

Object-oriented programming (OOP) is a programming paradigm that uses "objects" to design applications and computer programs.

In Mechanical Engineering with software programs such as Pro/ENGINEER, Solidworks, and Autodesk Inventor users can design products as pieces not part of the whole and later add those pieces together to form assemblies like building LEGO. Engineers call this piece part design.

This bottom-up approach has one weakness. Good intuition is necessary to decide the functionality that is to be provided by the module. If a system is to be built from existing system, this approach is more suitable as it starts from some existing modules.

Pro/ENGINEER (as well as other commercial Computer Aided Design (CAD) programs) does however hold the possibility to do Top-Down design by the use of so-called *skeletons*. They are generic structures that hold information on the overall layout of the product. Parts can inherit interfaces and parameters from this generic structure. Like parts, skeletons can be put into a hierarchy. Thus, it is possible to build the overall layout of a product before the parts are designed.

## Parsing

Parsing is the process of analyzing an input sequence (such as that read from a file or a keyboard) in order to determine its grammatical structure. This method is used in the analysis of both natural languages and computer languages, as in a compiler.

Bottom-up parsing is a strategy for analyzing unknown data relationships that attempts to identify the most fundamental units first, and then to infer higher-order structures from them. Top-down parsers, on the other hand, hypothesize general parse tree structures and then consider whether the known fundamental structures are compatible with the hypothesis.

## *Nanotechnology*

**Top-down** and **bottom-up** are two approaches for the manufacture of products. These terms were first applied to the field of nanotechnology by the Foresight Institute in 1989 in order to distinguish between molecular manufacturing (to mass-produce large atomically precise objects) and conventional manufacturing (which can mass-produce large objects that are not atomically precise). Bottom-up approaches seek to have smaller (usually molecular) components built up into more complex assemblies, while top-down approaches seek to create nanoscale devices by using larger, externally-controlled ones to direct their assembly.

The top-down approach often uses the traditional workshop or microfabrication methods where externally-controlled tools are used to cut, mill, and shape materials into the desired shape and order. Micropatterning techniques, such as photolithography and inkjet printing belong to this category.

Bottom-up approaches, in contrast, use the chemical properties of single molecules to cause single-molecule components to (a) self-organize or self-assemble into some useful

conformation, or (b) rely on positional assembly. These approaches utilize the concepts of molecular self-assembly and/or molecular recognition.

Such bottom-up approaches should, broadly speaking, be able to produce devices in parallel and much cheaper than top-down methods, but could potentially be overwhelmed as the size and complexity of the desired assembly increases.

### *Neuroscience and psychology*


An example of top down processing: Even though the second letter in each word is ambiguous, top down processing allows for easy disambiguation based on the context.

These terms are also employed in neuroscience, cognitive neuroscience and cognitive psychology to discuss the flow of information in processing. Typically sensory input is considered "down", and higher cognitive processes, which have more information from other sources, are considered "up". A **bottom-up** process is characterized by an absence of higher level direction in sensory processing, whereas a **top-down** process is characterized by a high level of direction of sensory processing by more cognition, such as goals or targets.

According to Psychology notes written by Dr. Charles Ramskov, a Psychology professor at De Anza College, Rock, Neiser, and Gregory claim that top-down approach involves perception that is active and constructive process. Additionally, it is an approach not directly given by stimulus input, but is the result of stimulus, internal hypotheses, and expectation interactions. According to Theoretical Synthesis, "when a stimulus is presented short and clarity is uncertain that gives a vague stimulus, perception becomes a top-down approach."

Conversely, Psychology defines bottom-up processing as an approach wherein there is a progression from the individual elements to the whole. According to Ramskov, one proponent of bottom-up approach, Gibson, claims that it is a process that includes visual perception that needs information available from proximal stimulus produced by the distal stimulus. Theoretical Synthesis also claims that bottom-up processing occurs "when a stimulus is presented long and clearly enough."

Cognitively speaking, certain cognitive processes, such as fast reactions or quick visual identification, are considered bottom-up processes because they rely primarily on sensory information, whereas processes such as motor control and directed attention are considered top-down because they are goal directed.

Neurologically speaking, some areas of the brain, such as area V1 mostly have bottom-up connections. Other areas, such as the fusiform gyrus have inputs from higher brain areas and are considered to have top-down influence.

The study of visual attention provides an example. If your attention is drawn to a flower in a field, it may be because the color or shape of the flower are visually salient. The information that caused you to attend to the flower came to you in a bottom-up fashion — your attention was not contingent upon knowledge of the flower; the outside stimulus was sufficient on its own.

Contrast this situation with one in which you are looking for a flower. You have a representation of what you are looking for. When you see the object you are looking for, it is salient. This is an example of the use of top-down information.

In cognitive terms, two thinking approaches are distinguished. "Top down" (or "big chunk") is stereotypically the visionary, or the person who sees the larger picture and overview. Such people focus on the big picture and from that derive the details to support it. "Bottom up" (or "small chunk") cognition is akin to focusing on the detail primarily, rather than the landscape. The expression "seeing the wood for the trees" references the two styles of cognition.

## *Management and organization*

In management and organizational arenas, the terms "top down" and "bottom up" are used to indicate how decisions are made.

A "top down" approach is one where an executive, decision maker, or other person or body makes a decision. This approach is disseminated under their authority to lower levels in the hierarchy, who are, to a greater or lesser extent, bound by them. For example, a structure in which decisions either are approved by a manager, or approved by his or her authorized representatives based on the manager's prior guidelines, is top-down management.

A "bottom up" approach is one that works from the grassroots — from a large number of people working together, causing a decision to arise from their joint involvement. A decision by a number of activists, students, or victims of some incident to take action is a "bottom-up" decision.

Positive aspects of top-down approaches include their efficiency and superb overview of higher levels. Also, external effects can be internalized. On the negative side, if reforms are perceived to be imposed ‘from above’, it can be difficult for lower levels to accept them (e.g. Bresser Pereira, Maravall, and Przeworski 1993). Evidence suggests this to be true regardless of the content of reforms (e.g. Dubois 2002). A bottom-up approach allows for more experimentation and a better feeling for what is needed at the bottom.

### State organization

Both approaches can be found in the organization of states, this involving political decisions.

In bottom-up organized organizations, e.g. ministries and their subordinate entities, decisions are prepared by experts in their fields, which define, out of their expertise, the policy they deem necessary. If they cannot agree, even on a compromise, they *escalate* the problem to the next higher hierarchy level, where a decision would be sought. Finally, the highest common principal might have to take the decision. Information is in the debt of the inferior to the superior, which means that the inferior owes information to the superior. In the effect, as soon as inferiors agree, the head of the organization only provides his or her “face″ for the decision which their inferiors have agreed upon.

Among several countries, the German political system provides one of the purest forms of a bottom-up approach. The German Federal Act on the Public Service provides that any inferior has to consult and support any superiors, that he or she – only – has to follow “general guidelines" of the superiors, and that he or she would have to be fully responsible for any own act in office, and would have to follow a specific, formal complaint procedure if in doubt of the legality of an order. Frequently, German politicians had to leave office on the allegation that they took wrong decisions because of their resistance to inferior experts' opinions (this commonly being called to be “beratungsresistent", or resistant to consultation, in German). The historical foundation of this approach lies with the fact that, in the 19th century, many politicians used to be noblemen without appropriate education, who more and more became forced to rely on consultation of educated experts, which (in particular after the Prussian reforms of Stein and Hardenberg) enjoyed the status of financially and personally independent, indismissable, and neutral experts as *Beamte* (public servants under public law).

A similar approach can be found in British police laws, where entitlements of police constables are vested in the constable in person and not in the police as an administrative agency, this leading to the single constable being fully responsible for his or her own acts in office, in particular their legality. The experience of two dictatorships in the country and, after the end of such regimes, emerging calls for the legal responsibility of the “aidees of the aidees" (*Helfershelfer*) of such regimes also furnished calls for the principle of personal responsibility of any expert for any decision made, this leading to a strengthening of the bottom-up approach, which requires maximum responsibility of the superiors.

In the opposite, the French administration is based on a top-down approach, where regular public servants enjoy no other task than simply to execute decisions made by their superiors. As those superiors also require consultation, this consultation is provided by members of a *cabinet*, which is distinctive from the regular ministry staff in terms of staff and organization. Those members who are not members of the *cabinet* are not entitled to make any suggestions or to take any decisions of political dimension.

The advantage of the bottom-up approach is the level of expertise provided, combined with the motivating experience of any member of the administration to be responsible and finally the independent “engine" of progress in that field of personal responsibility. A disadvantage is the lack of democratic control and transparency, this leading, from a democratic viewpoint, to the deferment of actual power of policy-making to faceless, if

even unknown, public servants. Even the fact that certain politicians might “provide their face" to the actual decisions of their inferiors might not mitigate this effect, but rather strong parliamentary rights of control and influence in legislative procedures (as they do exist in the example of Germany).

The advantage of the top-bottom principle is that political and administrative responsibilities are clearly distinguished from each other, and that responsibility for political failures can be clearly identified with the relevant office holder. Disadvantages are that the system triggers demotivation of inferiors, who know that their ideas to innovative approaches might not be welcome just because of their position, and that the decision-makers cannot make use of the full range of expertise which their inferiors will have collected.

Administrations in *dictatorships* traditionally work according to a strict top-down approach. As civil servants below the level of the political leadership are discouraged from making suggestions, they use to suffer from the lack of expertise which could be provided by the inferiors, which regularly leads to a breakdown of the system after an few decades. Modern communist states, which the People's Republic of China forms an example of, therefore prefer to define a framework of permissible, or even encouraged, criticism and self-determination by inferiors, which would not affect the major state doctrine, but allows the use of professional and expertise-driven knowledge and the use of it for the decision-making persons in office.

### Public health

Both top-down and bottom-up approaches exist in public health. There are many examples of top-down programs, often run by governments or large inter-governmental organizations (IGOs); many of these are disease-specific or issue-specific, such as HIV control or Smallpox Eradication. Examples of bottom-up programs include many small NGOs set up to improve local access to healthcare. However, a lot of programs seek to combine both approaches; for instance, guinea worm eradication, a single-disease international program currently run by the Carter Center has involved the training of many local volunteers, boosting bottom-up capacity, as have international programs for hygiene, sanitation, and access to primary health-care.

## *Architectural*

Often, the École des Beaux-Arts school of design is said to have primarily promoted top-down design because it taught that an architectural design should begin with a parti, a basic plan drawing of the overall project.

By contrast, the Bauhaus focused on bottom-up design. This method manifested itself in the study of translating small-scale organizational systems to a larger, more architectural scale (as with the woodpanel carving and furniture design).

## *Ecological*

In ecology, top down control refers to when a top predator controls the structure or population dynamics of the ecosystem. The classic example is of kelp forest ecosystems. In such ecosystems, sea otters are a keystone predator. They prey on urchins which in turn eat kelp. When otters are removed, urchin populations grow and reduce the kelp forest creating urchin barrens. In other words, such ecosystems are not controlled by productivity of the kelp but rather a top predator.

Bottom up control in ecosystems refers to ecosystems in which the nutrient supply and productivity and type of primary producers (plants and phytoplankton) control the ecosystem structure. An example would be how plankton populations are controlled by the availability of nutrients. Plankton populations tend to be higher and more complex in areas where upwelling brings nutrients to the surface.

There are many different examples of these concepts. It is not uncommon for populations to be influenced by both types of control.

Chapter 12

# 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.

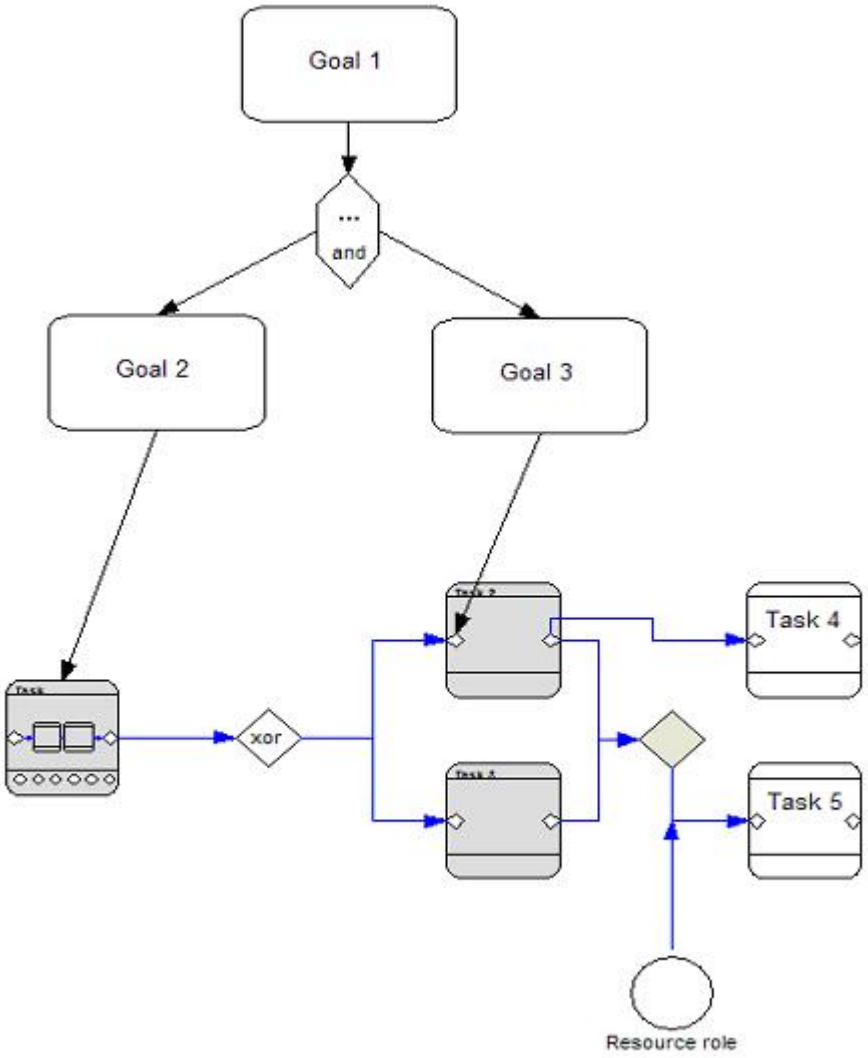
## *Current Implementation*

### 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 13

# Extended Enterprise Modeling Language




Example of EEML Goal modeling and process modeling.

**Extended Enterprise Modeling Language** (EEML) in software engineering is a modelling language used for Enterprise modelling across a number of layers.

## *Overview*

Extended Enterprise Modeling Language (EEML) is modelling language, which combines structural modeling, business process modeling, goal modeling with goal hierarchies, and resource modeling. It is used in practice to bridge the type of goal modeling used in common requirements engineering to other modeling approaches. According to Johannesson and Söderström (2008) "the process logic in EEML is mainly expressed through nested structures of tasks and decision points. The sequencing of tasks is expressed by the flow relation between decision points. Each task has an input port and the output port being decision points for modeling process logic".

EEML is intended to be a simple language, which makes it easy to update models. In addition to capturing the various tasks(can consist of several sub-tasks) and their interdependencies, models show which roles perform each task, and the tools, services and information they apply.

## *History*

Extended Enterprise Modeling Language (EEML) is from the late 1990s, developed in the EU project EXTERNAL as extension of the Action Port Model (APM) by S. Carlsen (1998). The EXTERNAL project aimed to "facilitate inter-organisational cooperation in knowledge intensive industries. It is the hypotheses of the project that interactive process models form a suitable framework for tools and methodologies for dynamically networked organisations. In the project EEML (Extended Enterprise Modelling Language) was first constructed as a common metamodel, designed to enable syntactic and semantic interoperability".

It has been further developed in the EU projects Unified Enterprise Modelling Language (UEML) from 2002 to 2003 and the ongoing ATHENA project. The objectives of UEML Working group has been to "define, to validate and to disseminate a set of core language constructs to support a Unified Language for Enterprise Modelling, named UEML, to serve as a basis for interoperability within a smart organisation or a network of enterprises".

## *EEML Topics*

### Modeling domains

The EEML-language is divided into 4 sub-languages, with well-defined links across these languages:

- Process modeling
- Data modeling

- Resource modeling
- Goal modeling

Process modeling in EEML, according to Krogstie (2006) "supports the modeling of process logic which is mainly expressed through nested structures of tasks and decision points. The sequencing of the tasks is expressed by the flow relation between decision points. Each task has minimum an input port and an output port being decision points for modeling process logic, Resource roles are used to connect resources of various kinds (persons, organizations, information, material objects, software tools and manual tools) to the tasks. In addition, data modeling (using UML class diagrams), goal modeling and competency modeling (skill requirements and skills possessed) can be integrated with the process models".

## EEML Layers

EEML has four layers of interest

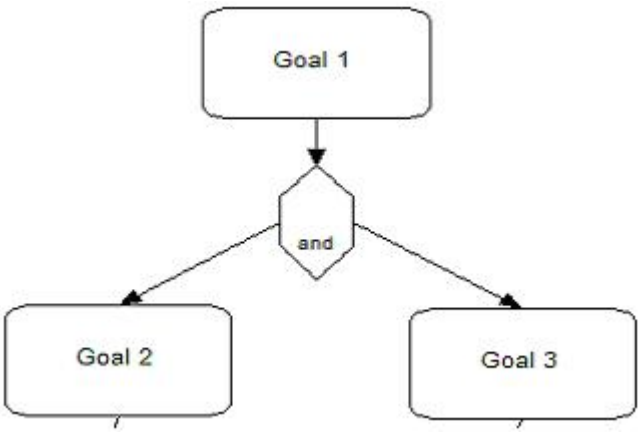
- **Generic Task Type**: This layer identifies the constituent tasks of generic, repetitive processes and the logical dependencies between these tasks.
- S**pecific Task Type**: At this layer, we deal with process modelling in another scale, which is more linked to the concretisation, decomposition and specialisation phases. Here process models are expanded and elaborated to facilitate business solutions. From an integration viewpoint, this layer aims at uncovering more efficiently the dependencies between the sub-activities, with regards for the resources required for actual performance.
- **Manage Task Instances**: The purpose of this layer consists in providing constraints but also useful resources (in the form of process templates) to the planning and performance of an enterprise process. The performance of organizational, information, and tool resources in their environment are highlighted through concrete resources allocation management.
- **Perform Task Instances**: Here is covered the actual execution of tasks with regards to issues of empowerment and decentralization. At this layer, resources are utilized or consumed in an exclusive or shared manner.

These tasks are tied together through another layer called **Manage Task Knowledge** which allows to achieve a global interaction through the different layers by performing a real consistency between them. According to EEML 2005 Guide, this Manage Task Knowledge can be defined as the collection of processes necessary for innovation, dissemination, and exploitation of knowledge in a co-operating ensemble where interact knowledge seekers and knowledge sources by the mean of a shared knowledge base.

## Goal Modelling

Goal Modelling is one of the four EEML modeling domains age. A goal expresses the wanted (or unwanted) state of affairs (either current or future) in a certain context. Example of the goal model is depicted below. It shows goals and relationships between

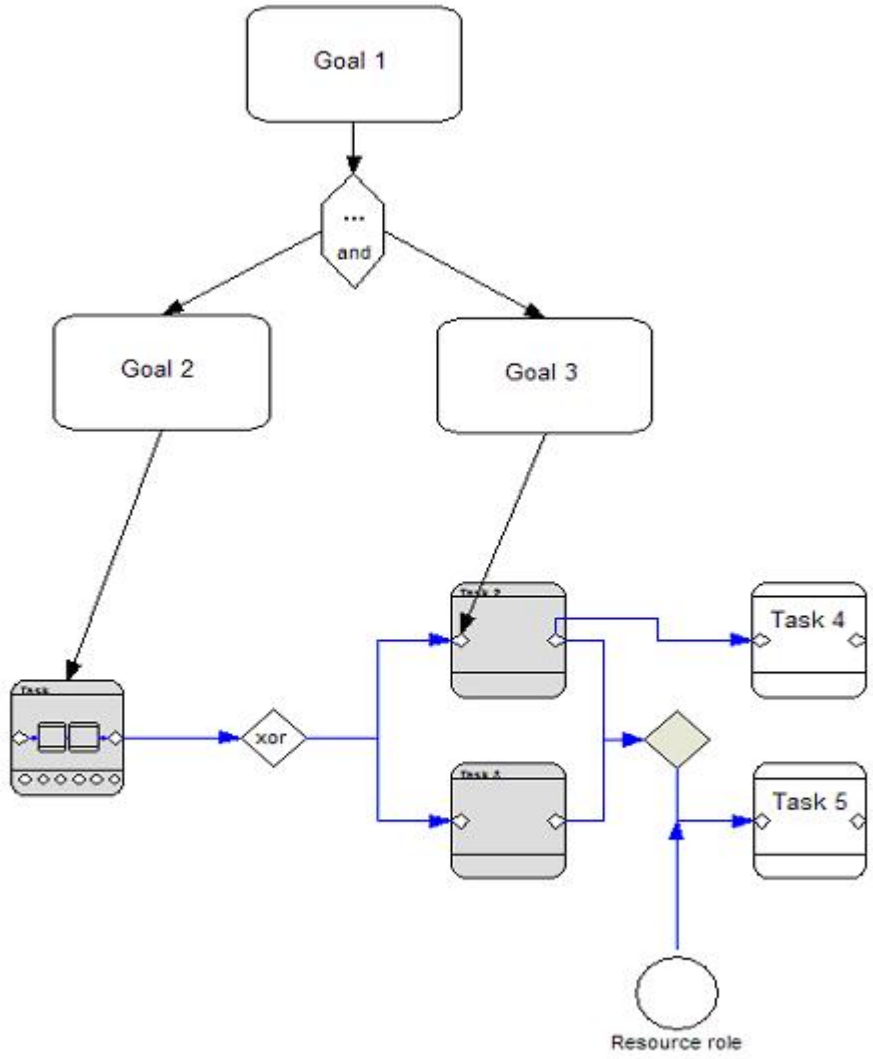
them. It is possible to model advanced goal-relationships in EEML by using goal connectors. A goal connector is used when one need to link several goals.




Goal modeling in EEML

| Relation type | Related type | Explanation |
|---|---|---|
| Goaltogoal | Goal | Indicate relationships between goals. A goal connector is used when needing to link several goals |
| Is source of | person\|organisation (Origin) | The person or organisation that is the source of the goal |
| Applies to | task\|meeting\|role\|resource (target) | The task, meeting, role or resource which the goal is relevant for |
| Is precondition for | inport (target) | A rule to describe more precisely the condition for starting a task |
| Is postcondition for | outport (target) | |
| Is decisionrule of | decisionpoint (target) | A rule to guide the performance of a decision |
| Is actionrule for | task(target) | A rule for automation of task |

Connecting relationships




Goal modeling and process modeling

In goal modeling to fulfil Goal1, one must achieve to other goals: both Goal2 and Goal3 (goal-connector with “and” as the logical relation going out). If Goal2 and Goal3 are two different ways of achieving Goal1, then it should be “xor” logical relationship. It can be an opposite situation when both Goal2 and Goal3 need to be fulfilled and to achieve them one must fulfil Goal1. In this case Goal2 and Goal3 are linked to goal connector and this goal connector has a link to Goal1 with ”and”-logical relationship.

The table indicate different types of connecting relationships in EEML goal modeling. Goal model can also be interlinked with a process model.

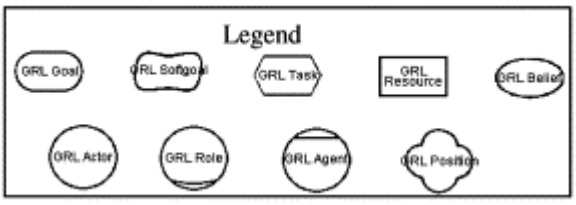
## Goal modelling principles

Within requirements engineering (RE), the notion of goal has increasingly been used. Goals generally describe objectives which a system should achieve through cooperation

of actors in the intended software and in the environment. Goals are central in some RE frameworks, and can play a supporting role in others. Goal-oriented techniques may particularly be useful in early-phase RE. Early-phase requirements consider e.g. how the intended system meets organizational goals, why the system is needed and how the stakeholders' interests may be addressed.

- Expresses the relationships between systems and their environments : Earlier, requirements engineering focused only on what the system is supposed to do. Over the past years, there has been a more or less mutual understanding, that it is also very important to understand and characterize the interaction between the intended system and its environment. Relationships between systems and their environments are often expressed as goal-based relationships. The motivation for this is "partly today's more dynamic business and organizational environments, where systems are increasingly used to fundamentally change business processes rather than to automate long-established practices". Goals can also be useful when modelling contexts.
- Clarifies requirements : Specifying goals leads to asking "why", "how" and "how else". Requirements of the stakeholders are often revealed in this process. The stakeholders may seem to be more likely to become aware of potential alternatives for fulfilling their goals, and thereby less likely to over-specify their requirements. Requirements from clients and stakeholders may often be unclear, especially the non-functional ones. A goal-oriented approach allows the requirements to be refined and clarified through an incremental process, by analyzing requirements in terms of goal decomposition.
- Deals with conflicts : Goals may provide a useful way of dealing with conflicts, such as tradeoffs between costs performance, flexibility, etc., and divergent interests of the stakeholders. Goals can deal with conflicts because meeting of one goal can interfere with the meeting of others. Different opinions on how to meet a goal has led to different ways of handling conflicts.
- Decides requirements completeness : Requirements can be considered complete if they fulfil explicit goals in the requirement model.
- Connects requirements to design : Goals can be used in order to connect the requirements to the design. For some, goals are an important mechanism in this matter. (The Non-Functional Requirements (NFR) framework uses goals to guide the design process.)

## Goal-oriented Requirements Language




GRL Notation

Goal-oriented Requirements Language (GRL) is a language that is designed to support goal-oriented modeling and reasoning about requirements, especially the non-functional requirements  It allows to express conflict between goals and helps to make decisions that resolve conflicts. There are three main categories of concepts in GRL: intentional elements, intentional relationships and actors. They are called for intentional because they are used in models that primarily concerned with answering "why" question of requirements (for ex. why certain choices for behavior or structure were made, what alternatives exist and what is the reason for choosing of certain alternative.

## *Goal and process oriented modeling*

We can describe process model as models that comprise a set of activities and an activity can be decomposed into sub-activities. These activities have relationship amongst themselves. A goal describes the expected state of operation in a business enterprise and it can be linked to whole process model or to a process model fragment with each level activity in a process model can be considered as a goal.

Goals are related in a hierarchical format where you find some goals are dependent on other sub goals for them to be complete which means all the sub goals must be achieved for the main goal to be achieved. There is other goals where only one of the goals need to be fulfilled for the main goal to be achieved. In goal modeling, there is use of deontic operator which falls in between the context and achieved state. Goals apply to tasks, milestones, resource roles and resources as well and can be considered as action rule for at task. EEML rules were also possible to although the goal modeling requires much more consultation in finding the connections between rules on the different levels. Goal-oriented analysis focuses on the description and evaluation of alternatives and their relationship to the organizational objectives.

## *Resource modeling*

Resources have specific roles during the execution of various processes in an organisation. The following icons represent the various resources required in modeling. The relations of these resources can be of different types:

a. Is Filled By - -this is the assignment relation between roles and resources. It has a cardinality of one-to-many relationship.

b. Is Candidate For – candidate indicates the possible filling of the role by a resource.

c. Has Member – this is a kind of relations between organization and person by denoting that a certain person has membership in the organization. Has a cardinality of many-to-many relation.

d. Provide Support To – support pattern between resources and roles.

e. Communicates With – Communication pattern between resources and roles.

f. Has Supervision Over – shows which role resource supervises another role or resource.

g. Is Rating Of – describes the relation between skill and a person or organization.

h. Is required By – this is the primary skill required for this role

i. Has Access to – creating of models with the access rights.

## *Benefits of using EEML*

From a general point of view, EEML can be used like any other modeling languages in numerous cases. However we can highlight the virtual enterprise example, which can be considered as a direct field of application for EEML with regard to Extended Enterprise planning, operation, and management.

- Knowledge sharing: Create and maintain a shared understanding of the scope and purpose of the enterprise, as well as viewpoints on how to fulfil the purpose.
- Dynamically networked organisations: Make knowledge as available as possible within the organization.
- Heterogeneous infrastructures: Achieve a relevant knowledge sharing process through heterogeneous infrastructures.
- Process knowledge management: Integrate the different business processes levels of abstraction.
- Motivation: creates enthusiasm and commitment among members of an organization to follow up on the various actions that are necessary to restructure the enterprise.

EEML can help organisations meet these challenges by modeling all the manufacturing and logistics processes in the extended enterprise. This model allows capturing a rich set of relationships between the organization, people, processes and resources of the virtual enterprise. It also aims at making people understand, communicate, develop and cultivate solutions to business problems

According to J. krogstie (2008), Enterprise Models can be created to serve various purposes which include:

1. Human sense making and communication -the main purpose of enterprise modeling is to make sense of the real world aspects of an enterprise in order to facilitate communicate with parties involved.
2. Computer assisted analysis - the main purpose of enterprise modeling is to gain knowledge about the enterprise through simulation and computation of various parameters.
3. Model deployment and activation - the main purpose of enterprise modeling is to integrate the model in an enterprise-wide information system and enabling on-line information retrieval and direct work process guidance.

EEML enables Extended Enterprises to build up their operation based on standard processes through allowing modeling of all actors, processes and tasks in the Extended Enterprise and thereby have clear description of the Extended Enterprise. Finally, models developed will be used to measure and evaluate the Extended Enterprise.