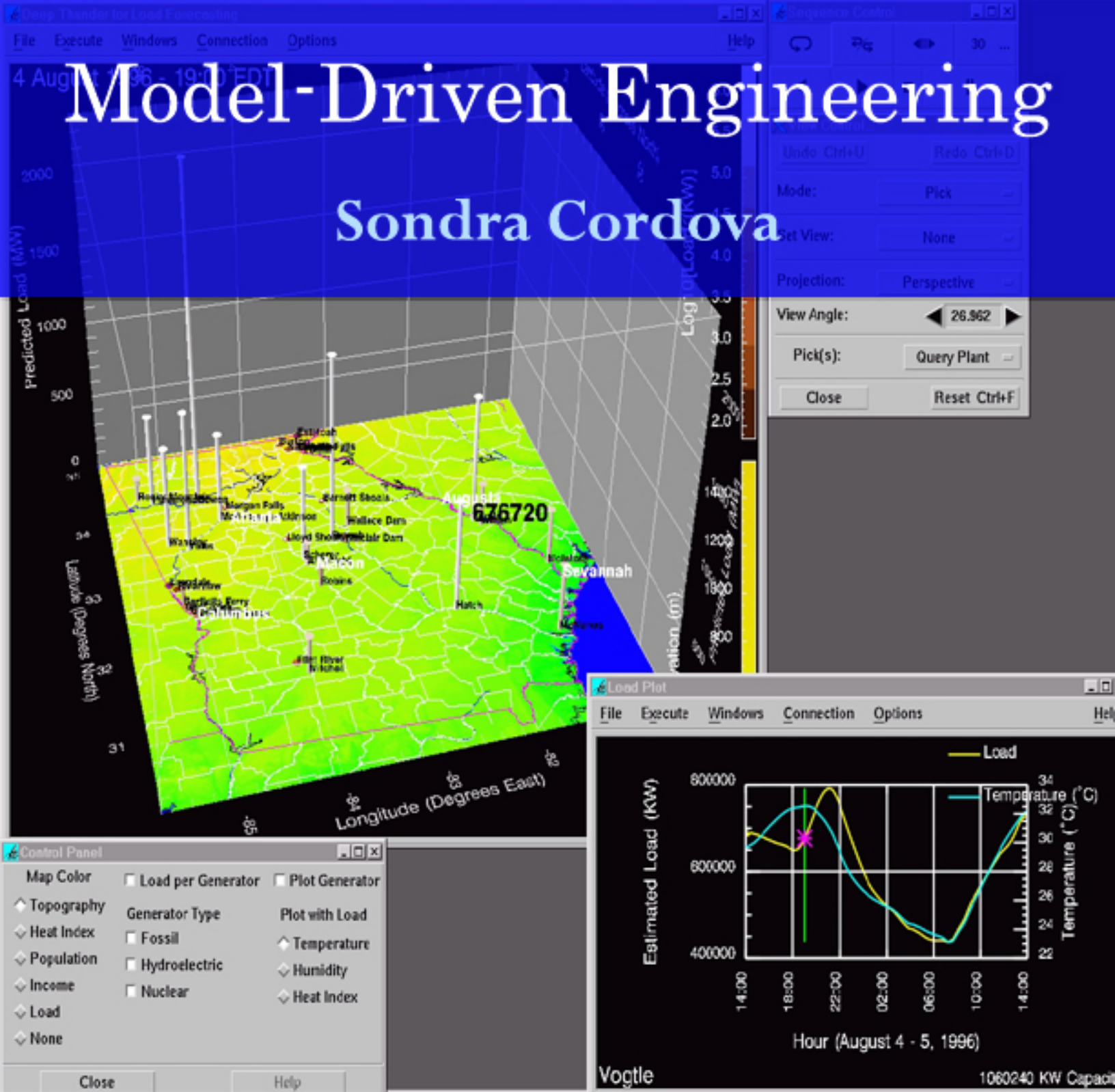
Model-Driven Engineering

Sondra Cordova



First Edition, 2012

ISBN 978-81-323-4187-1

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

Table of Contents

Introduction

Model-driven engineering (MDE) is a software development methodology which focuses on creating and exploiting domain models (that is, abstract representations of the knowledge and activities that govern a particular application domain), rather than on the computing (or algorithmic) concepts. The MDE approach is meant to increase productivity by maximizing compatibility between systems (via reuse of standardized models), simplifying the process of design (via models of recurring design patterns in the application domain), and promoting communication between individuals and teams working on the system (via a standardization of the terminology and the *best practices* used in the application domain).

A modeling paradigm for MDE is considered effective, if its models make sense from the point of view of a user that is familiar with the domain, and if they can serve as a basis for implementing systems. The models are developed through extensive communication among product managers, designers, developers and users of the application domain. As the models approach completion, they enable the development of software and systems.

Some of the better known MDE initiatives are:

- the Object Management Group (OMG) initiative Model-Driven Architecture (MDA), which is a registered trademark of OMG.
- the Eclipse ecosystem of programming and modelling tools.

History of MDE

The first tools to support MDE were the Computer-Aided Software Engineering (CASE) tools developed in the 1980s. Companies like Integrated Development Environments (IDE - StP), Higher Order Software (now Hamilton Technologies, Inc., HTI), Cadre Technologies, Bachman Information Systems, and Logicworks (BP-Win and ER-Win) were pioneers in the field. Except for HTI's 001AXES Universal Systems Language (USL) and its associated automation (001), CASE had the same problem that current MDA/MDE tools have today: the model gets out of sync with the application (see below). The government got involved in the modeling definitions creating the IDEF specifications. With several variations of the modeling definitions they were eventually joined creating the Unified Modeling Language (UML). Rational Rose, the dominant

product for UML implementation, was done by Rational Corporation (Booch) which in 2002 was acquired by IBM.

MDE as used in software engineering

As it pertains to software development, model-driven engineering refers to a range of development approaches that are based on the use of software modeling as a primary form of expression. Sometimes models are constructed to a certain level of detail, and then code is written by hand in a separate step. Sometimes complete models are built including executable actions. Code can be generated from the models, ranging from system skeletons to complete, deployable products. With the introduction of the Unified Modeling Language (UML), MDE has become very popular today with a wide body of practitioners and supporting tools. More advanced types of MDE have expanded to permit industry standards which allow for consistent application and results. The continued evolution of MDE has added an increased focus on architecture and automation.

MDE technologies with a greater focus on architecture and corresponding automation yield higher levels of abstraction in software development. This abstraction promotes simpler models with a greater focus on problem space. Combined with executable semantics this elevates the total level of automation possible. The Object Management Group (OMG) has developed a set of standards called model-driven architecture (MDA), building a foundation for this advanced architecture-focused approach.

According to Douglas C. Schmidt, model-driven engineering technologies offer a promising approach to address the inability of third-generation languages to alleviate the complexity of platforms and express domain concepts effectively.

MDE and Rapid Applications Development

Very Rapid Applications Development uses Object Programing to separate that the customer demands from software. It is Model Driven Engineering applied to RAD EDI. One stage of programing is deleted which is the development of the software from the analysis. Some passive files are read from the VRAD engine to create the software. The passive files are the analysis.

Some enterprises used Very Rapid Applications Development to assimilate quickly the business of little enterprises. But most of them did not used MDE to include them. They just created another EDI.

Chapter 1

Model-Driven Architecture

Model-driven architecture (MDA) is a software design approach for the development of software systems. It provides a set of guidelines for the structuring of specifications, which are expressed as models. Model-driven architecture is a kind of domain engineering, and supports model-driven engineering of software systems. It was launched by the Object Management Group (OMG) in 2001.

Overview

The Model-Driven Architecture approach defines system functionality using a platform-independent model (PIM) using an appropriate domain-specific language (DSL).

Then, given a platform definition model (PDM) corresponding to CORBA, .NET, the Web, etc., the PIM is translated to one or more platform-specific models (PSMs) that computers can run. This requires mappings and transformations and should be modeled too.

The PSM may use different Domain Specific Languages (DSLs), or a General Purpose Language (GPL) like Java, C#, PHP, Python, etc.. Automated tools generally perform this translation.

The OMG organization provides rough specifications rather than implementations, often as answers to Requests for Proposals (RFPs). Implementations come from private companies or open source groups.

MDA principles can also apply to other areas such as business process modeling (BPM) where the PIM is translated to either automated or manual processes.

Related standards

The MDA model is related to multiple standards, including the Unified Modeling Language (UML), the Meta-Object Facility (MOF), XML Metadata Interchange (XMI), Enterprise Distributed Object Computing (EDOC), the Software Process Engineering Metamodel (SPEM), and the Common Warehouse Metamodel (CWM). Note that the

term “architecture” in Model-driven architecture does not refer to the architecture of the system being modeled, but rather to the architecture of the various standards and model forms that serve as the technology basis for MDA.

Executable UML is another specific approach to implement MDA

Trademark

The Object Management Group holds trademarks on MDA, as well as several similar terms including Model Driven Application Development, Model Based Application Development, Model Based Programming, and others. The main acronym that has not yet been deposited by OMG until now is Model-driven engineering (MDE). As a consequence, the research community uses MDE to refer to general model engineering ideas, without committing to strict OMG standards.

Model-driven architecture topics

MDA approach

OMG focuses Model-driven architecture on forward engineering, i.e. producing code from abstract, human-elaborated modelling diagrams (e.g. class diagrams). OMG's ADTF (Analysis and Design Task Force) group leads this effort. With some humour, the group chose ADM (MDA backwards) to name the study of reverse engineering. ADM decodes to Architecture-Driven Modernization. The objective of ADM is to produce standards for model-based reverse engineering of legacy systems . Knowledge Discovery Metamodel (KDM) is the furthest along of these efforts, and describes information systems in terms of various assets (programs, specifications, data, test files, database schemas, etc.).

One of the main aims of the MDA is to separate design from architecture. As the concepts and technologies used to realize designs and the concepts and technologies used to realize architectures have changed at their own pace, decoupling them allows system developers to choose from the best and most fitting in both domains. The design addresses the functional (use case) requirements while architecture provides the infrastructure through which non-functional requirements like scalability, reliability and performance are realized. MDA envisages that the platform independent model (PIM), which represents a conceptual design realizing the functional requirements, will survive changes in realization technologies and software architectures.

Of particular importance to model-driven architecture is the notion of model transformation. A specific standard language for model transformation has been defined by OMG called QVT.

MDA tools

The OMG organization provides rough specifications rather than implementations, often as answers to Requests for Proposals (RFPs). The OMG documents the overall process in a document called the MDA Guide.

Basically, an MDA tool is a tool used to develop, interpret, compare, align, measure, verify, transform, etc. models or metamodels. In the following section "model" is interpreted as meaning any kind of model (e.g. a UML model) or metamodel (e.g. the CWM metamodel). In any MDA approach we have essentially two kinds of models: *initial models* are created manually by human agents while *derived models* are created automatically by programs. For example an analyst may create a UML initial model from its observation of some loose business situation while a Java model may be automatically derived from this UML model by a Model transformation operation.

An MDA tool may be one or more of the following types:

- Creation Tool: A tool used to elicit initial models and/or edit derived models.
- Analysis Tool: A tool used to check models for completeness, inconsistencies, or error and warning conditions. Also used to calculate metrics for the model.
- Transformation Tool: A tool used to transform models into other models or into code and documentation.
- Composition Tool: A tool used to compose (i.e. to merge according to a given composition semantics) several source models, preferably conforming to the same metamodel.
- Test Tool: A tool used to "test" models as described in Model-based testing.
- Simulation Tool: A tool used to simulate the execution of a system represented by a given model. This is related to the subject of model execution.
- Metadata Management Tool: A tool intended to handle the general relations between different models, including the metadata on each model (e.g. author, date of creation or modification, method of creation (which tool? which transformation? etc.)) and the mutual relations between these models (i.e. one metamodel is a version of another one, one model has been derived from another one by a transformation, etc.)
- Reverse Engineering Tool: A tool intended to transform particular legacy or information artifact portfolios into full-fledged models.

Some tools perform more than one of the functions listed above. For example, some creation tools may also have transformation and test capabilities. There are other tools that are solely for creation, solely for graphical presentation, solely for transformation, etc.

One of the characteristics of MDA tools is that they mainly take models (e.g. MOF models or metamodels) as input and generate models as output. In some cases however the parameters may be taken outside the MDA space like in model to text or text to model transformation tools.

Implementations of the OMG specifications come from private companies or open source groups. One important source of implementations for OMG specifications is the Eclipse Foundation (EF). Many implementations of OMG modeling standards may be found in the Eclipse Modeling Framework (EMF) or Graphical Modeling Framework (GMF), the Eclipse foundation is also developing other tools of various profiles as GMT. Eclipse's compliance to OMG specifications is often not strict. This is true for example for OMG's EMOF standard, which Eclipse approximates with its ECORE implementation. More examples may be found in the M2M project implementing the QVT standard or in the M2T project implementing the MOF2Text standard.

One should be careful not to confuse the *List of MDA Tools* and the List of UML tools, the former being much broader. This distinction can be made more general by distinguishing 'variable metamodel tools' and 'fixed metamodel tools'. A UML CASE tool is typically a 'fixed metamodel tool' since it has been hard-wired to work only with a given version of the UML metamodel (e.g. UML 2.1). On the contrary, other tools have internal generic capabilities allowing them to adapt to arbitrary metamodels or to a particular kind of metamodels.

Usually MDA tools focus rudimentary architecture specification, although in some cases the tools are architecture-independent (or platform independent).

Simple examples of architecture specifications include:

- Selecting one of a number of supported reference architectures like Java EE or Microsoft .NET,
- Specifying the architecture at a finer level including the choice of presentation layer technology, business logic layer technology, persistence technology and persistence mapping technology (e.g. object-relational mapper).
- Metadata: information about data.

MDA concerns

Some key concepts that underpin the MDA approach (launched in 2001) were first elucidated by the Shlaer-Mellor method during the late 1980s. Indeed a key absent technical standard of the MDA approach (that of an action language syntax for Executable UML) has been bridged by some vendors by adapting the original Shlaer-Mellor Action Language (modified for UML). However during this period the MDA approach has not gained mainstream industry acceptance; with the Gartner Group still identifying MDA as an "on the rise" technology in its 2006 "Hype Cycle", and Forrester Research declaring MDA to be "D.O.A." in 2006. Potential concerns that have been raised with the OMG MDA approach include:

- Incomplete Standards: The MDA approach is underpinned by a variety of technical standards, some of which are yet to be specified (e.g. an action semantic language for xtUML), or are yet to be implemented in a standard manner (e.g. a QVT transformation engine or a PIM with a virtual execution environment).

- Vendor Lock-in: Although MDA was conceived as an approach for achieving (technical) platform independence, current MDA vendors have been reluctant to engineer their MDA toolsets to be interoperable. Such an outcome could result in vendor lock-in for those pursuing an MDA approach.
- Idealistic: MDA is conceived as a forward engineering approach in which models that incorporate Action Language programming are transformed into implementation artifacts (e.g. executable code, database schema) in one direction via a fully or partially automated "generation" step. This aligns with OMG's vision that MDA should allow modelling of a problem domain's full complexity in UML (and related standards) with subsequent transformation to a complete (executable) application. This approach does, however, imply that changes to implementation artifacts (e.g. database schema tuning) are not supported . This constitutes a problem in situations where such post-transformation "adapting" of implementation artifacts is seen to be necessary. Evidence that the full MDA approach may be too idealistic for some real world deployments has been seen in the rise of so-called "pragmatic MDA". Pragmatic MDA blends the literal standards from OMG's MDA with more traditional model driven mechanisms such as round-trip engineering that provides support for adapting implementation artifacts.
- Specialised Skillsets: Practitioners of MDA based software engineering are (as with other toolsets) required to have a high level of expertise in their field. Current expert MDA practitioners (often referred to as Modeller/Architects) are scarce relative to the availability of traditional developers.
- OMG Track Record: The OMG consortium who sponsor the MDA approach (and own the MDA trademark) also introduced and sponsored the CORBA standard which itself failed to materialise as a widely utilised standard.
- Uncertain Value Proposition (UVP): As discussed, the vision of MDA allows for the specification of a system as an abstract model, which may be realized as a concrete implementation (program) for a particular computing platform (e.g. .NET). Thus an application that has been successfully developed via a pure MDA approach could theoretically be ported to a newer release .NET platform (or even a Java platform) in a deterministic manner – although significant questions remain as to real-world practicalities during translation (such as user interface implementation). Whether this capability represents a significant value proposition remains a question for particular adopters. Regardless, adopters of MDA who are seeking value via an "alternative to programming" should be very careful when assessing this approach. The complexity of any given problem domain will always remain, and the programming of business logic needs to be undertaken in MDA as with any other approach. The difference with MDA is that the programming language used (e.g. xtUML) is more abstract (than, say, Java or C#) and exists interwoven with traditional UML artifacts (e.g. class diagrams). Whether programming in a language that is more abstract than mainstream 3GL languages will result in systems of better quality, cheaper cost or faster delivery, is a question that has yet to be adequately answered.

Conferences

Among the various conferences on this topic we may mention ECMDA, the European Conference on MDA and also MoDELS, former firmed as <<UML>> conference series (till 2004), the Italian Forum on MDA in collaboration with the OMG. There are also several conferences and workshops (at OOPSLA, ECOOP mainly) focusing on more specific aspects of MDA like model transformation, model composition, and generation.

Code generation controversy

Code generation means that the user abstractly models solutions, which are connoted by some model data, and then an automated tool derives from the models parts or all of the source code for the software system. In some tools, the user can provide a skeleton of the program source code, in the form of a source code template where predefined tokens are then replaced with program source code parts during the code generation process.

An often cited criticism is that the UML diagrams just lack the detail which is needed to contain the same information as is covered with the program source. Some developers even claim that "the Code *is* the design" .

Chapter 2

Software Development Methodology

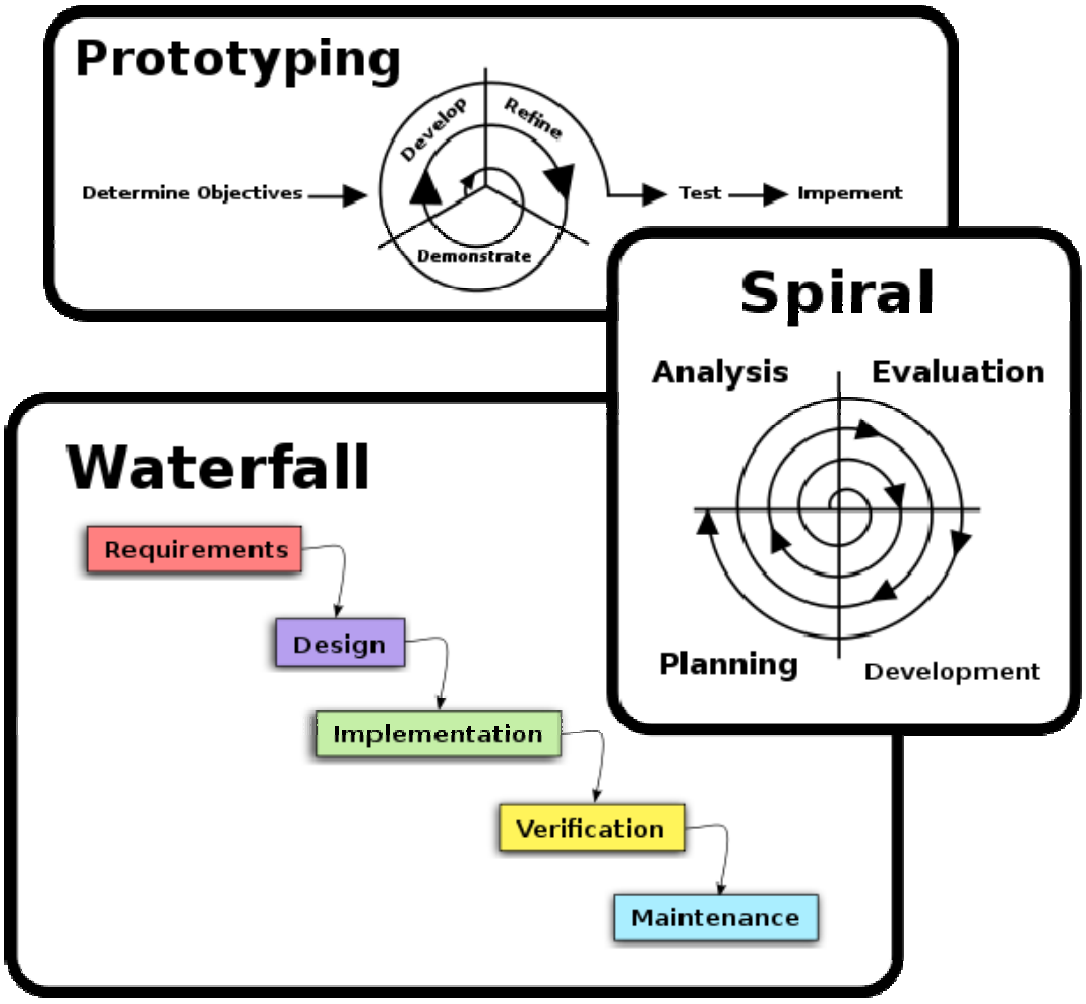
A **software development methodology** or **system development methodology** in software engineering is a framework that is used to structure, plan, and control the process of developing an information system.

History

The software development methodology framework didn't emerge until the 1960s. According to Elliott (2004) the systems development life cycle (SDLC) can be considered to be the oldest formalized methodology framework for building information systems. The main idea of the SDLC has been "to pursue the development of information systems in a very deliberate, structured and methodical way, requiring each stage of the life cycle from inception of the idea to delivery of the final system, to be carried out in rigidly and sequentially". within the context of the framework being applied. The main target of this methodology framework in the 1960s was "to develop large scale functional business systems in an age of large scale business conglomerates. Information systems activities revolved around heavy data processing and number crunching routines".

As a noun

As a noun, a software development methodology is a framework that is used to structure, plan, and control the process of developing an information system - this includes the pre-definition of specific deliverables and artifacts that are created and completed by a project team to develop or maintain an application.



The three basic approaches applied to software development methodology frameworks.

A wide variety of such frameworks have evolved over the years, each with its own recognized strengths and weaknesses. One software development methodology framework is not necessarily suitable for use by all projects. Each of the available methodology frameworks are best suited to specific kinds of projects, based on various technical, organizational, project and team considerations.

These software development frameworks are often bound to some kind of organization, which further develops, supports the use, and promotes the methodology framework. The methodology framework is often defined in some kind of formal documentation. Specific software development methodology frameworks (noun) include

- Rational Unified Process (RUP, IBM) since 1998.
- Agile Unified Process (AUP) since 2005 by Scott Ambler

As a verb

As a verb, the software development methodology is an approach used by organizations and project teams to apply the software development methodology framework (noun). Specific software development methodologies (verb) include:

1970s

- Structured programming since 1969
- Cap Gemini SDM, originally from PANDATA, the first English translation was published in 1974. SDM stands for System Development Methodology

1980s

- Structured Systems Analysis and Design Methodology (SSADM) from 1980 onwards
- Information Requirement Analysis/Soft systems methodology

1990s

- Object-oriented programming (OOP) has been developed since the early 1960s, and developed as a dominant programming approach during the mid-1990s
- Rapid application development (RAD) since 1991
- Scrum, since the late 1990s
- Team software process developed by Watts Humphrey at the SEI
- Extreme Programming since 1999

Verb approaches

Every software development methodology framework acts as a basis for applying specific approaches to develop and maintain software. Several software development approaches have been used since the origin of information technology. These are:

- Waterfall: a linear framework
- Prototyping: an iterative framework
- Incremental: a combined linear-iterative framework
- Spiral: a combined linear-iterative framework
- Rapid application development (RAD): an iterative framework
- Extreme Programming

Waterfall development

The Waterfall model is a sequential development approach, in which development is seen as flowing steadily downwards (like a waterfall) through the phases of requirements analysis, design, implementation, testing (validation), integration, and maintenance. The

first formal description of the method is often cited as an article published by Winston W. Royce in 1970 although Royce did not use the term "waterfall" in this article.

The basic principles are:

- Project is divided into sequential phases, with some overlap and splashback acceptable between phases.
- Emphasis is on planning, time schedules, target dates, budgets and implementation of an entire system at one time.
- Tight control is maintained over the life of the project via extensive written documentation, formal reviews, and approval/signoff by the user and information technology management occurring at the end of most phases before beginning the next phase.

Prototyping

Software prototyping, is the development approach of activities during software development, the creation of prototypes, i.e., incomplete versions of the software program being developed.

The basic principles are:

- Not a standalone, complete development methodology, but rather an approach to handling selected parts of a larger, more traditional development methodology (i.e. incremental, spiral, or rapid application development (RAD)).
- Attempts to reduce inherent project risk by breaking a project into smaller segments and providing more ease-of-change during the development process.
- User is involved throughout the development process, which increases the likelihood of user acceptance of the final implementation.
- Small-scale mock-ups of the system are developed following an iterative modification process until the prototype evolves to meet the users' requirements.
- While most prototypes are developed with the expectation that they will be discarded, it is possible in some cases to evolve from prototype to working system.
- A basic understanding of the fundamental business problem is necessary to avoid solving the wrong problem.

Incremental development

Various methods are acceptable for combining linear and iterative systems development methodologies, with the primary objective of each being to reduce inherent project risk by breaking a project into smaller segments and providing more ease-of-change during the development process.

The basic principles are:

- A series of mini-Waterfalls are performed, where all phases of the Waterfall are completed for a small part of a system, before proceeding to the next increment, or
- Overall requirements are defined before proceeding to evolutionary, mini-Waterfall development of individual increments of a system, or
- The initial software concept, requirements analysis, and design of architecture and system core are defined via Waterfall, followed by iterative Prototyping, which culminates in installing the final prototype, a working system.

Spiral development

The spiral model.

The spiral model is a software development process combining elements of both design and prototyping-in-stages, in an effort to combine advantages of top-down and bottom-up concepts. It is a meta-model, a model that can be used by other models.

The basic principles are:

- Focus is on risk assessment and on minimizing project risk by breaking a project into smaller segments and providing more ease-of-change during the development process, as well as providing the opportunity to evaluate risks and weigh consideration of project continuation throughout the life cycle.
- "Each cycle involves a progression through the same sequence of steps, for each part of the product and for each of its levels of elaboration, from an overall concept-of-operation document down to the coding of each individual program."
- Each trip around the spiral traverses four basic quadrants: (1) determine objectives, alternatives, and constraints of the iteration; (2) evaluate alternatives; Identify and resolve risks; (3) develop and verify deliverables from the iteration; and (4) plan the next iteration.

- Begin each cycle with an identification of stakeholders and their win conditions, and end each cycle with review and commitment.

Rapid application development

Rapid application development (RAD) is a software development methodology, which involves iterative development and the construction of prototypes. Rapid application development is a term originally used to describe a software development process introduced by James Martin in 1991.

The basic principles are:

- Key objective is for fast development and delivery of a high quality system at a relatively low investment cost.
- Attempts to reduce inherent project risk by breaking a project into smaller segments and providing more ease-of-change during the development process.
- Aims to produce high quality systems quickly, primarily via iterative Prototyping (at any stage of development), active user involvement, and computerized development tools. These tools may include Graphical User Interface (GUI) builders, Computer Aided Software Engineering (CASE) tools, Database Management Systems (DBMS), fourth-generation programming languages, code generators, and object-oriented techniques.
- Key emphasis is on fulfilling the business need, while technological or engineering excellence is of lesser importance.
- Project control involves prioritizing development and defining delivery deadlines or “timeboxes”. If the project starts to slip, emphasis is on reducing requirements to fit the timebox, not in increasing the deadline.
- Generally includes joint application design (JAD), where users are intensely involved in system design, via consensus building in either structured workshops, or electronically facilitated interaction.
- Active user involvement is imperative.
- Iteratively produces production software, as opposed to a throwaway prototype.
- Produces documentation necessary to facilitate future development and maintenance.
- Standard systems analysis and design methods can be fitted into this framework.

Other practices

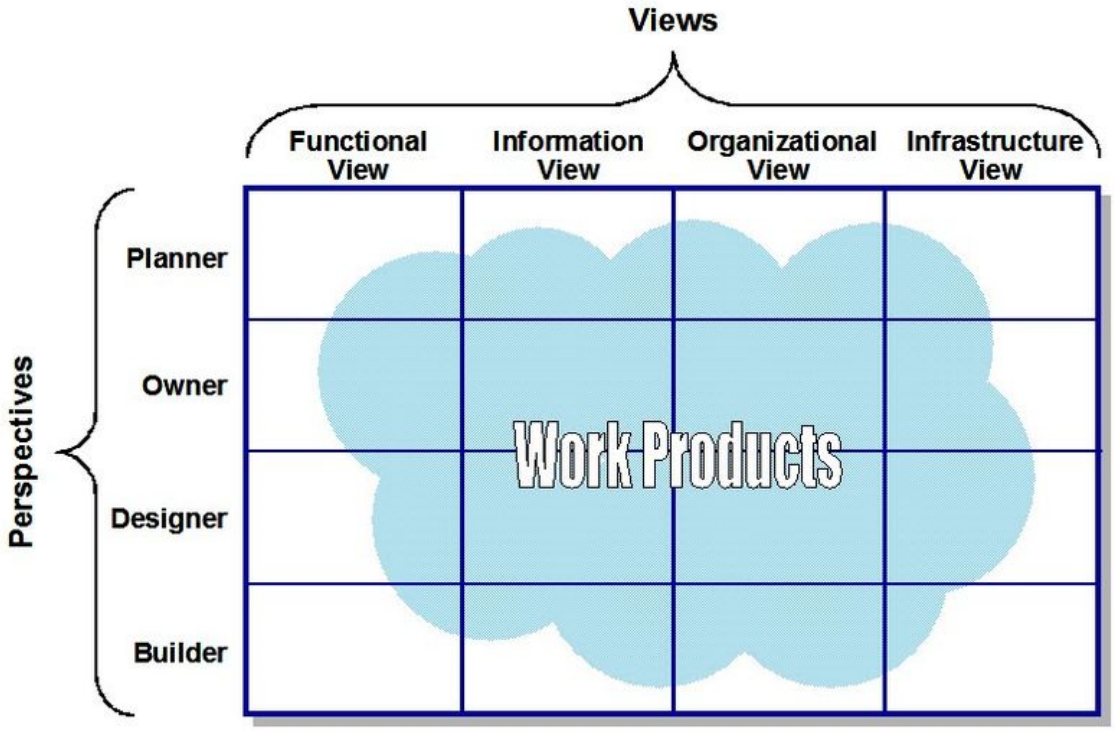
Other methodology practices include:

- Object-oriented development methodologies, such as Grady Booch's object-oriented design (OOD), also known as object-oriented analysis and design (OOAD). The Booch model includes six diagrams: class, object, state transition, interaction, module, and process.
- Top-down programming: evolved in the 1970s by IBM researcher Harlan Mills (and Niklaus Wirth) in developed structured programming.

- Unified Process (UP) is an iterative software development methodology framework, based on Unified Modeling Language (UML). UP organizes the development of software into four phases, each consisting of one or more executable iterations of the software at that stage of development: inception, elaboration, construction, and guidelines. Many tools and products exist to facilitate UP implementation. One of the more popular versions of UP is the Rational Unified Process (RUP).
- Agile software development refers to a group of software development methodologies based on iterative development, where requirements and solutions evolve via collaboration between self-organizing cross-functional teams. The term was coined in the year 2001 when the Agile Manifesto was formulated.

Subtopics

View model



The TEAF Matrix of Views and Perspectives.

A view model is framework which provides the viewpoints on the system and its environment, to be used in the software development process. It is a graphical representation of the underlying semantics of a view.

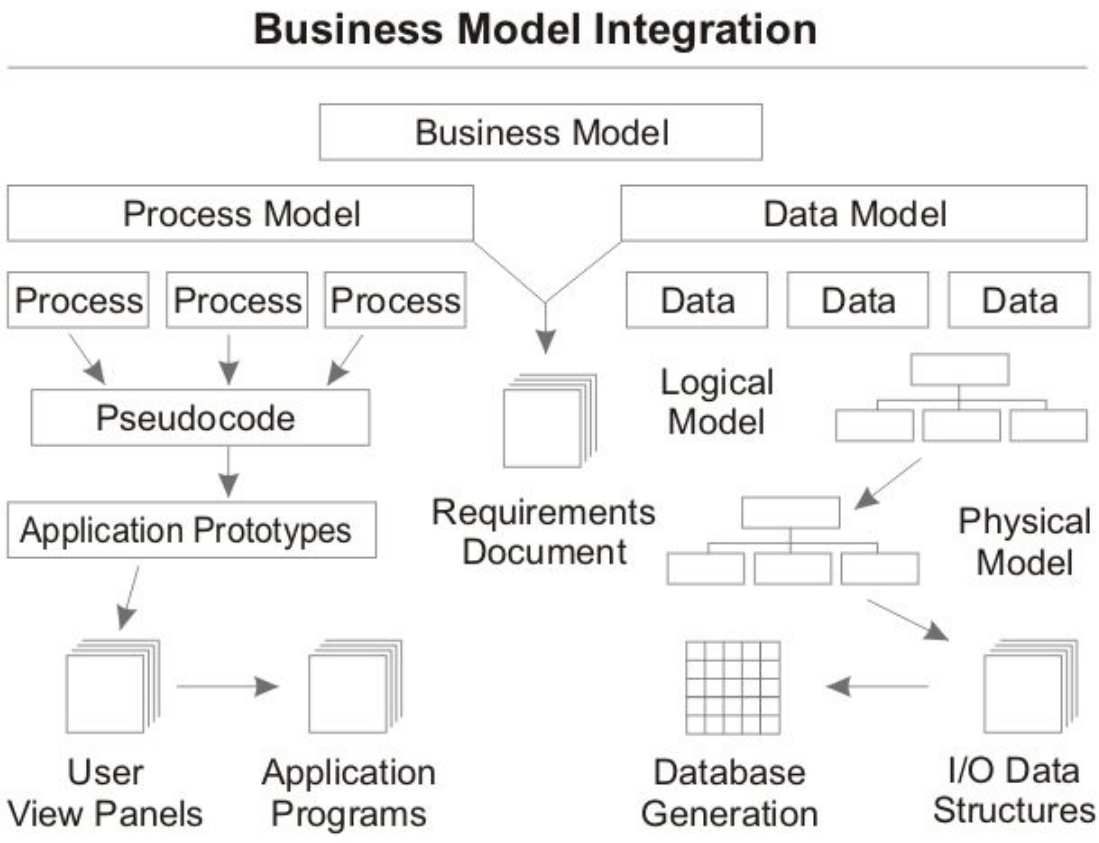
The purpose of viewpoints and views is to enable human engineers to comprehend very complex systems, and to organize the elements of the problem and the solution around

domains of expertise. In the engineering of physically intensive systems, viewpoints often correspond to capabilities and responsibilities within the engineering organization.

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

Business process and data modelling

Graphical representation of the current state of information provides a very effective means for presenting information to both users and system developers.



example of the interaction between business process and data models.

- A business model illustrates the functions associated with the business process being modeled and the organizations that perform these functions. By depicting activities and information flows, a foundation is created to visualize, define, understand, and validate the nature of a process.
- A data model provides the details of information to be stored, and is of primary use when the final product is the generation of computer software code for an application or the preparation of a functional specification to aid a computer software make-or-buy decision.

Usually, a model is created after conducting an interview, referred to as business analysis. The interview consists of a facilitator asking a series of questions designed to extract required information that describes a process. The interviewer is called a facilitator to emphasize that it is the participants who provide the information. The facilitator should have some knowledge of the process of interest, but this is not as important as having a structured methodology by which the questions are asked of the process expert. The methodology is important because usually a team of facilitators is collecting information across the facility and the results of the information from all the interviewers must fit together once completed.

The models are developed as defining either the current state of the process, in which case the final product is called the "as-is" snapshot model, or a collection of ideas of what the process should contain, resulting in a "what-can-be" model. Generation of process and data models can be used to determine if the existing processes and information systems are sound and only need minor modifications or enhancements, or if re-engineering is required as a corrective action. The creation of business models is more than a way to view or automate your information process. Analysis can be used to fundamentally reshape the way your business or organization conducts its operations.

Computer-aided software engineering

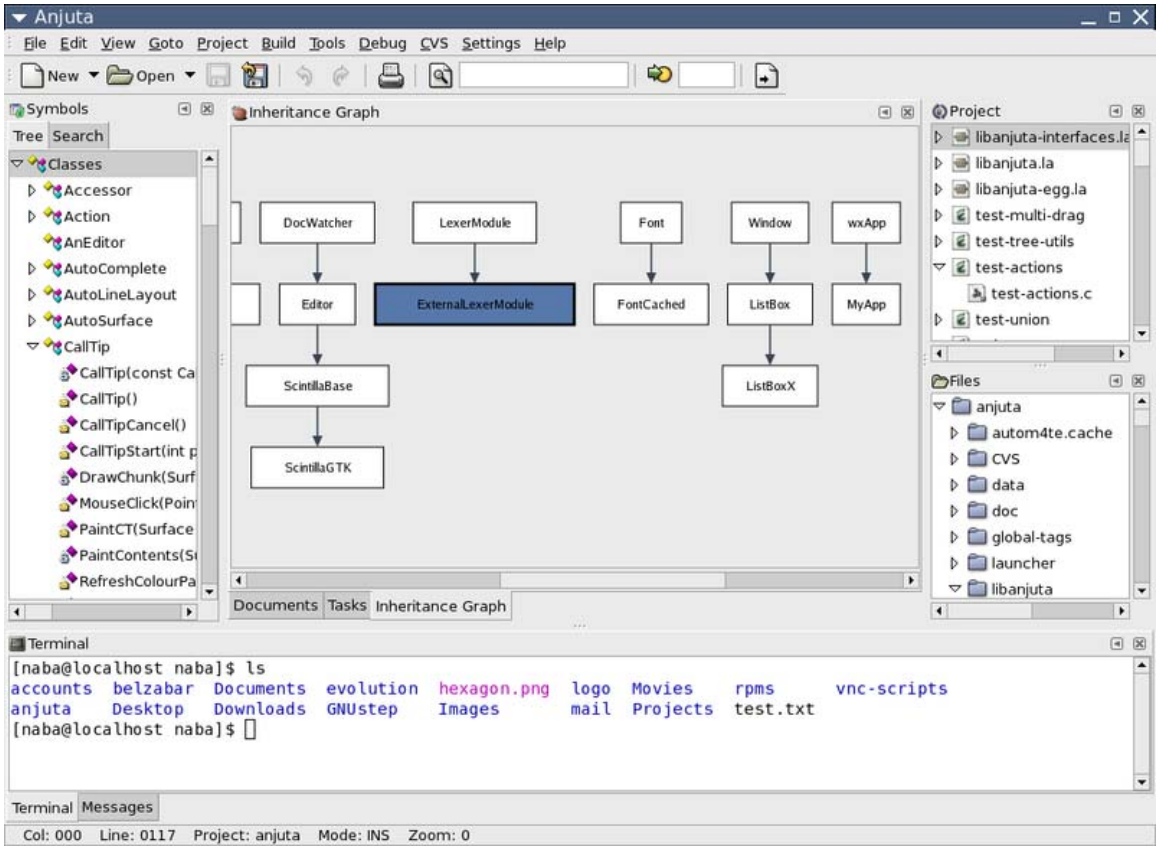
Computer-aided software engineering (CASE), in the field software engineering is the scientific application of a set of tools and methods to a software which results in high-quality, defect-free, and maintainable software products. It also refers to methods for the development of information systems together with automated tools that can be used in the software development process. The term "computer-aided software engineering" (CASE) can refer to the software used for the automated development of systems software, i.e., computer code. The CASE functions include analysis, design, and programming. CASE tools automate methods for designing, documenting, and producing structured computer code in the desired programming language.

Two key ideas of Computer-aided Software System Engineering (CASE) are:

- Foster computer assistance in software development and or software maintenance processes, and
- An engineering approach to software development and or maintenance.

Typical CASE tools exist for configuration management, data modeling, model transformation, refactoring, source code generation, and Unified Modeling Language.

Integrated development environment



Anjuta, a C and C++ IDE for the GNOME environment

An integrated development environment (IDE) also known as *integrated design environment* or *integrated debugging environment* is a software application that provides comprehensive facilities to computer programmers for software development. An IDE normally consists of a:

- source code editor,
- compiler and/or interpreter,
- build automation tools, and
- debugger (usually).

IDEs are designed to maximize programmer productivity by providing tight-knit components with similar user interfaces. Typically an IDE is dedicated to a specific programming language, so as to provide a feature set which most closely matches the programming paradigms of the language.

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. Graphical modeling languages use a diagram techniques with named symbols that represent concepts and lines that connect the symbols and that represent relationships and various other graphical annotation to represent constraints. Textual modeling languages typically use standardised keywords accompanied by parameters to make computer-interpretable expressions.

Example of graphical modelling languages in the field of software engineering are:

- Business Process Modeling Notation (BPMN, and the XML form BPML) 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 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.
- 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.
- Specification and Description Language(SDL) is a specification language targeted at the unambiguous specification and description of the behaviour of reactive and distributed systems.
- Unified Modeling Language (UML) is a general-purpose modeling language that is an industry standard for specifying software-intensive systems. UML 2.0, the current version, supports thirteen different diagram techniques, and has widespread tool support.

Not all modeling languages are executable, and for those that are, using them doesn't necessarily mean that programmers are no longer needed. On the contrary, executable modeling languages are intended to amplify the productivity of skilled programmers, so that they can address more difficult problems, such as parallel computing and distributed systems.

Programming paradigm

A programming paradigm is a fundamental style of computer programming, in contrast to a software engineering methodology, which is a style of solving specific software

engineering problems. Paradigms differ in the concepts and abstractions used to represent the elements of a program (such as objects, functions, variables, constraints...) and the steps that compose a computation (assignation, evaluation, continuations, data flows...).

A programming language can support multiple paradigms. For example programs written in C++ or Object Pascal can be purely procedural, or purely object-oriented, or contain elements of both paradigms. Software designers and programmers decide how to use those paradigm elements. In object-oriented programming, programmers can think of a program as a collection of interacting objects, while in functional programming a program can be thought of as a sequence of stateless function evaluations. When programming computers or systems with many processors, process-oriented programming allows programmers to think about applications as sets of concurrent processes acting upon logically shared data structures.

Just as different groups in software engineering advocate different *methodologies*, different programming languages advocate different *programming paradigms*. Some languages are designed to support one paradigm (Smalltalk supports object-oriented programming, Haskell supports functional programming), while other programming languages support multiple paradigms (such as Object Pascal, C++, C#, Visual Basic, Common Lisp, Scheme, Python, Ruby, and Oz).

Many programming paradigms are as well known for what methods they *forbid* as for what they enable. For instance, pure functional programming forbids using side-effects; structured programming forbids using goto statements. Partly for this reason, new paradigms are often regarded as doctrinaire or overly rigid by those accustomed to earlier styles. Avoiding certain methods can make it easier to prove theorems about a program's correctness, or simply to understand its behavior.

Software framework

A software framework is a re-usable design for a software system or subsystem. A software framework may include support programs, code libraries, a scripting language, or other software to help develop and *glue together* the different components of a software project. Various parts of the framework may be exposed via an API.

Software development process

A software development process is a framework imposed on the development of a software product. Synonyms include software life cycle and *software process*. There are several models for such processes, each describing approaches to a variety of tasks or activities that take place during the process.

A largely growing body of software development organizations implement process methodologies. Many of them are in the defense industry, which in the U.S. requires a rating based on 'process models' to obtain contracts. The international standard describing the method to select, implement and monitor the life cycle for software is ISO 12207.

A decades-long goal has been to find repeatable, predictable processes that improve productivity and quality. Some try to systematize or formalize the seemingly unruly task of writing software. Others apply project management methods to writing software. Without project management, software projects can easily be delivered late or over budget. With large numbers of software projects not meeting their expectations in terms of functionality, cost, or delivery schedule, effective project management appears to be lacking.

Chapter 3

Model-Based Design and Model-Based Testing

Model-based design

Model-Based Design (MBD) is a mathematical and visual method of addressing problems associated with designing complex control, signal processing and communication systems. It is used in many motion control, industrial equipment, aerospace, and automotive applications. Model-based design is a methodology applied in designing embedded software.

MBD provides an efficient approach for establishing a common framework for communication throughout the design process while supporting the development cycle ("V" diagram). In Model-based design of control systems, development is manifested in these four steps: 1) modeling a plant, 2) analyzing and synthesizing a controller for the plant, 3) simulating the plant and controller, and 4) integrating all these phases by deploying the controller. The model-based design paradigm is significantly different from traditional design methodology. Rather than using complex structures and extensive software code, designers can use MBD to define models with advanced functional characteristics using continuous-time and discrete-time building blocks. These built models used with simulation tools can lead to rapid prototyping, software testing, and verification. Not only is the testing and verification process enhanced, but also, in some cases, hardware-in-the-loop simulation can be used with the new design paradigm to perform testing of dynamic effects on the system more quickly and much more efficiently than with traditional design methodology.

The main steps in MBD approach are:

1. Plant modeling. Plant modeling can be data-driven or first principles based. Data-driven plant modeling uses techniques such as System identification. With system identification, the plant model is identified by acquiring and processing raw data from a real-world system and choosing a mathematical algorithm with which to identify a mathematical model. Various kinds of analysis and simulations can be performed using the identified model before it is used to design a model-based controller. First principles based modeling is based on creating a block diagram

model that implements known differential-algebraic equations governing plant dynamics. A type of first principles based modeling is physical modeling, where a model is created by connectings blocks that represent physical elements that the actual plant consists of.

2. Controller analysis and synthesis. The mathematical model conceived in step 1 is used to identify dynamic characteristics of the plant model. A controller can be then be synthesized based on these characteristics.
3. Offline simulation and real-time simulation. The time response of the dynamic system to complex, time-varying inputs is investigated. This is done by simulating a simple LTI or a non-linear model of the plant with the controller. Simulation allows specification, requirements, and modeling errors to be found immediately, rather than later in the design effort. Real-time simulation can be done by automatically generating code for the controller developed in step 3. This code can be deployed to a special real-time protoyping computer that can run the code and control the operation of the plant. If plant prototype is not available, or testing on the prototype is dangerous or expensive, code can be automatically generated from the plant model. This code can be deployed to the special real-time computer that can be connected to the target processor with running controller code. This way, controller can be tested in real-time against a real-time plant model.
4. Deployment. Ideally this is done via automatic code generation from the controller developed in step 3. It is unlikely that the controller will work on the actual system as well as it did in simulation, so an iterative debugging process is done by analyzing results on the actual target and updating the controller model. Model based design tools allow all these iterative steps to be performed in a unified visual environment.

Some of the notable advantages MBD offers in comparison to the traditional approach are:

- MBD provides a common design environment, which facilitates general communication, data analysis, and system verification between development groups.
- Engineers can locate and correct errors early in system design, when the time and financial impact of system modification are minimized.
- Design reuse, for upgrades and for derivative systems with expanded capabilities, is facilitated

History

The dawn of the electrical age brought many innovative and advanced control systems. As early as the 1920s two aspects of engineering, control theory and control systems, converged to make large-scale integrated systems possible. In those early days controls systems were commonly used in the industrial environment. Large process facilities started using process controllers for regulating continuous variables such as temperature, pressure, and flow rate. Electrical relays built into ladder-like networks were one of the first discrete control devices to automate an entire manufacturing process.

Control systems gained momentum, primarily in the automotive and aerospace sectors. In the 1950s and 1960s the push to Space generated interest in embedded control systems. Engineers constructed control systems such as engine control units and flight simulators, that could be part of the end product. By the end of the twentieth century, embedded control systems were ubiquitous, as even White goods such as washing machines and air-conditions contained complex and advanced control algorithms, making them a much more "intelligent".

In the year 1969, the first computer-based controllers were introduced, These early programmable logic controllers (PLC), mimicked the operations of already available discrete control technologies that used the out-dated relay ladders. The advent of PC technology brought a drastic shift in the process and discrete control market. An off-the-shelf desktop loaded with adequate hardware and software can run an entire process unit, and execute complex and established PID algorithms or work as a Distributed Control System (DCS).

Challenges

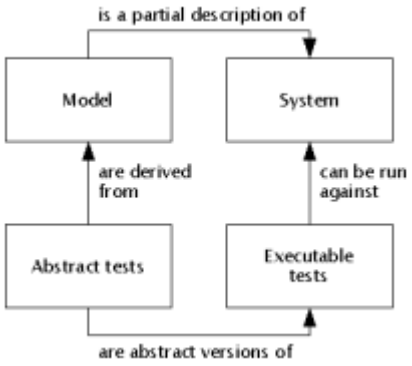
Modeling and simulation tools have long been in use, but traditional text-based tools are inadequate for the complex nature of modern control systems. Because of the limitations of graphical tools, design engineers previously relied heavily on text-based programming and mathematical models. However, developing these models was difficult, time-consuming, and highly prone to error. In addition, debugging text-based programs was a tedious process, requiring much trial and error before a final fault-free model could be created, especially since mathematical models undergo unseen changes during the translation through the various design stages.

These challenges are overcome by the use of graphical modeling tools, used today in all aspects of design. These tools provide a very generic and unified graphical modeling environment, they reduce the complexity of model designs by breaking them into hierarchies of individual design blocks. Designers can thus achieve multiple levels of model fidelity by simply substituting one block element with another. Graphical models are also the best way to document engineers’ ideas. It helps engineers to conceptualize the entire system and simplifies the process of transporting the model from one stage to another in the design process. Boeing's simulator EASY5 was among the first modeling tools to be provided with a graphical user interface. This was followed by many other tools.

When developing embedded control systems, designers are squeezed by two trends —- shrinking development cycles and growing design intricacy. The divide-and-conquer strategy for developing these complex systems means coordinating the resources of people with expertise in a wide range of disciplines. The traditional, text-based approach of embedded system design is not efficient enough to handle such advanced, complex systems.

Model-based testing

Model-based testing is the application of Model based design for designing and executing the necessary artifacts to perform software testing. This is achieved by having a model that describes all aspects of the testing data, mainly the test cases and the test execution environment. Usually, the testing model is derived in whole or in part from a model that describes some (usually functional) aspects of the system under test (SUT).



General model-based testing setting

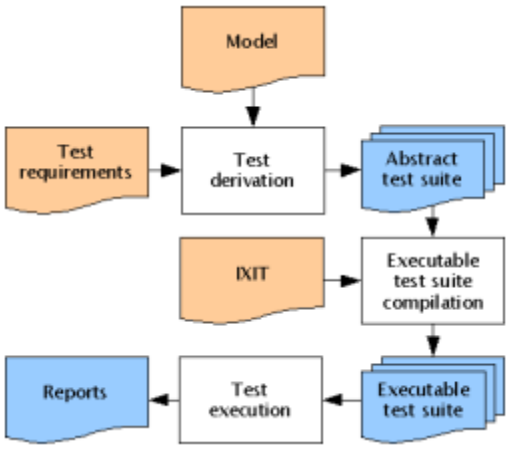
The model describing the SUT is usually an abstract, partial presentation of the system under test's desired behavior. The test cases derived from this model are functional tests on the same level of abstraction as the model. These test cases are collectively known as the abstract test suite. The abstract test suite cannot be directly executed against the system under test because it is on the wrong level of abstraction. Therefore an executable test suite must be derived from the abstract test suite that can communicate with the system under test. This is done by mapping the abstract test cases to concrete test cases suitable for execution. In some model-based testing tools, the model contains enough information to generate an executable test suite from it. In the case of online testing (see below), the abstract test suite exists only as a concept but not as an explicit artifact.

There are many different ways to "derive" tests from a model. Because testing is usually experimental and based on heuristics, there is no one best way to do this. It is common to consolidate all test derivation related design decisions into a package that is often known as "test requirements", "test purpose" or even "use case". This package can contain e.g. information about the part of the model that should be the focus for testing, or about the conditions where it is correct to stop testing (test stopping criteria).

Because test suites are derived from models and not from source code, model-based testing is usually seen as one form of black-box testing. In some aspects, this is not completely accurate. Model-based testing can be combined with source-code level test coverage measurement, and functional models can be based on existing source code in the first place.

Model-based testing for complex software systems is still an evolving field.

Models



An example of a model-based testing workflow (offline test case generation). IXIT refers to "implementation extra information" and denotes here the total package of information that is needed when the abstract test suite is converted into an executable one. Typically, it includes information about test harness, data mappings and SUT configuration.

Especially in Model Driven Engineering or in OMG's model-driven architecture the model is built before or parallel to the development process of the system under test. The model can also be constructed from the completed system. Recently the model is created mostly manually, but there are also attempts to create the model automatically, for instance out of the source code. One important way to create new models is by model transformation, using languages like ATL, a QVT-like Domain Specific Language.

Model-based testing inherits the complexity of the domain or, more particularly, of the related domain models.

Deploying model-based testing

There are various known ways to deploy model-based testing, which include **online testing**, **offline generation of executable tests**, and **offline generation of manually deployable tests**.

Online testing means that a model-based testing tool connects “directly” to a system under test and tests it dynamically.

Offline generation of executable tests means that a model-based testing tool generates test cases as a computer-readable asset that can be later deployed automatically. This asset can be, for instance, a collection of Python classes that embodies the generated testing logic.

Offline generation of manually deployable tests means that a model-based testing tool generates test cases as a human-readable asset that can be later deployed manually. This asset can be, for instance, a PDF document in English that describes the generated test steps.

Deriving tests algorithmically

The effectiveness of model-based testing is primarily due to the potential for automation it offers. If the model is machine-readable and formal to the extent that it has a well-defined behavioral interpretation, test cases can in principle be derived mechanically.

Often the model is translated to or interpreted as a finite state automaton or a state transition system. This automaton represents the possible configurations of the system under test. To find test cases, the automaton is searched for executable paths. A possible execution path can serve as a test case. This method works if the model is deterministic or can be transformed into a deterministic one. Valuable off-nominal test cases may be obtained by leveraging un-specified transitions in these models.

Depending on the complexity of the system under test and the corresponding model the number of paths can be very large, because of the huge amount of possible configurations of the system. For finding appropriate test cases, i.e. paths that refer to a certain requirement to proof, the search of the paths has to be guided. For test case generation, multiple techniques have been applied and are surveyed in.

Test case generation by theorem proving

Theorem proving has been originally used for automated proving of logical formulas. For model-based testing approaches the system is modeled by a set of logical expressions (predicates) specifying the system's behavior. For selecting test cases the model is partitioned into equivalence classes over the valid interpretation of the set of the logical expressions describing the system under test. Each class is representing a certain system behavior and can therefore serve as a test case. The simplest partitioning is done by the disjunctive normal form approach. The logical expressions describing the system's behavior are transformed into the disjunctive normal form.

Test case generation by constraint logic programming and symbolic execution

Constraint programming can be used to select test cases satisfying specific constraints by solving a set of constraints over a set of variables. The system is described by the means of constraints. Solving the set of constraints can be done by Boolean solvers (e.g. SAT-solvers based on the Boolean satisfiability problem) or by numerical analysis, like the Gaussian elimination. A solution found by solving the set of constraints formulas can serve as a test cases for the corresponding system.

Constraint programming can be combined with symbolic execution. In this approach a system model is executed symbolically, i.e. collecting data constraints over different control paths, and then using the constraint programming method for solving the constraints and producing test cases.

Test case generation by model checking

Model checkers can also be used for test case generation. Originally model checking was developed as a technique to check if a property of a specification is valid in a model. When used for testing, a model of the system under test, and a property to test is provided to the model checker. Within the procedure of proofing, if this property is valid in the model, the model checker detects witnesses and counterexamples. A witness is a path, where the property is satisfied, whereas a counterexample is a path in the execution of the model, where the property is violated. These paths can again be used as test cases.

Test case generation by using an event-flow model

A popular model that has recently been used extensively for testing software with a graphical user-interface (GUI) front-end is called the event-flow model that represents events and event interactions. In much the same way as a control-flow model represents all possible execution paths in a program, and a data-flow model represents all possible definitions and uses of a memory location, the event-flow model represents all possible sequences of events that can be executed on the GUI. More specifically, a GUI is decomposed into a hierarchy of modal dialogs; this hierarchy is represented as an integration tree; each modal dialog is represented as an event-flow graph that shows all possible event execution paths in the dialog; individual events are represented using their preconditions and effects. An overview of the event-flow model with associated algorithms to semi-automatically reverse engineer the model from an executing GUI software is presented in*this 2007 paper* . Because the event-flow model is not tied to a specific aspect of the GUI testing process, it may be used to perform a wide variety of testing tasks by defining specialized model-based techniques called event-space exploration strategies (ESES). These ESES use the event-flow model in a number of ways to develop an end-to-end GUI testing process, namely by checking the model, test-case generation, and test oracle creation.

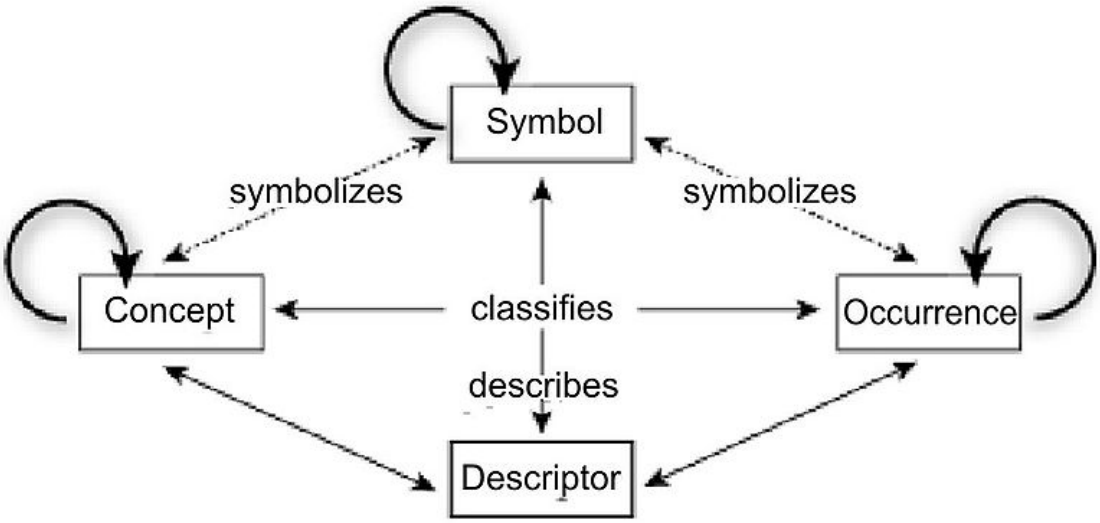
Test case generation by using a Markov chains model

Markov chains are an efficient way to handle Model-based Testing. Test model realized with Markov chains model can be understood as a usage model: we spoke of Usage/Statistical Model Based Testing. Usage models, so Markov chains, are mainly constructed by 2 artifacts : the Finite State Machine (FSM) which represents all possible usage scenario of the system and the Operational Profiles (OP) which qualify the FSM to represent how the system will statically will be used. The first (FSM) helps to know what can be or has been tested and the second (OP) helps to derive operational test cases. Usage/Statistical Model-based Testing starts from the facts that is not not possible to exhaustively test a system and that failure can appear with a very low rate.. This approach

offers a pragmatic way to statically derive test cases focused on: improving as prompt as possible the system under test reliability. The company ALL4TEC provides an implementation of this approach with the tool MaTeLo (Markov Test Logic). MaTeLo allows to model the test with Markov chains, derive executables test cases w.r.t the usage testing approach, and assess the system under test reliability with the help of the so called Test Campaign Analysis module.

Chapter 4

Metamodeling



Example of a Geologic map information meta-model, with four types of meta-objects, and their self-references.

Metamodeling, or *meta-modeling* in software engineering and systems engineering among other disciplines, is the analysis, construction and development of the frames, rules, constraints, models and theories applicable and useful for modeling a predefined class of problems. As its name implies, this concept applies the notions of meta- and modeling.

Overview

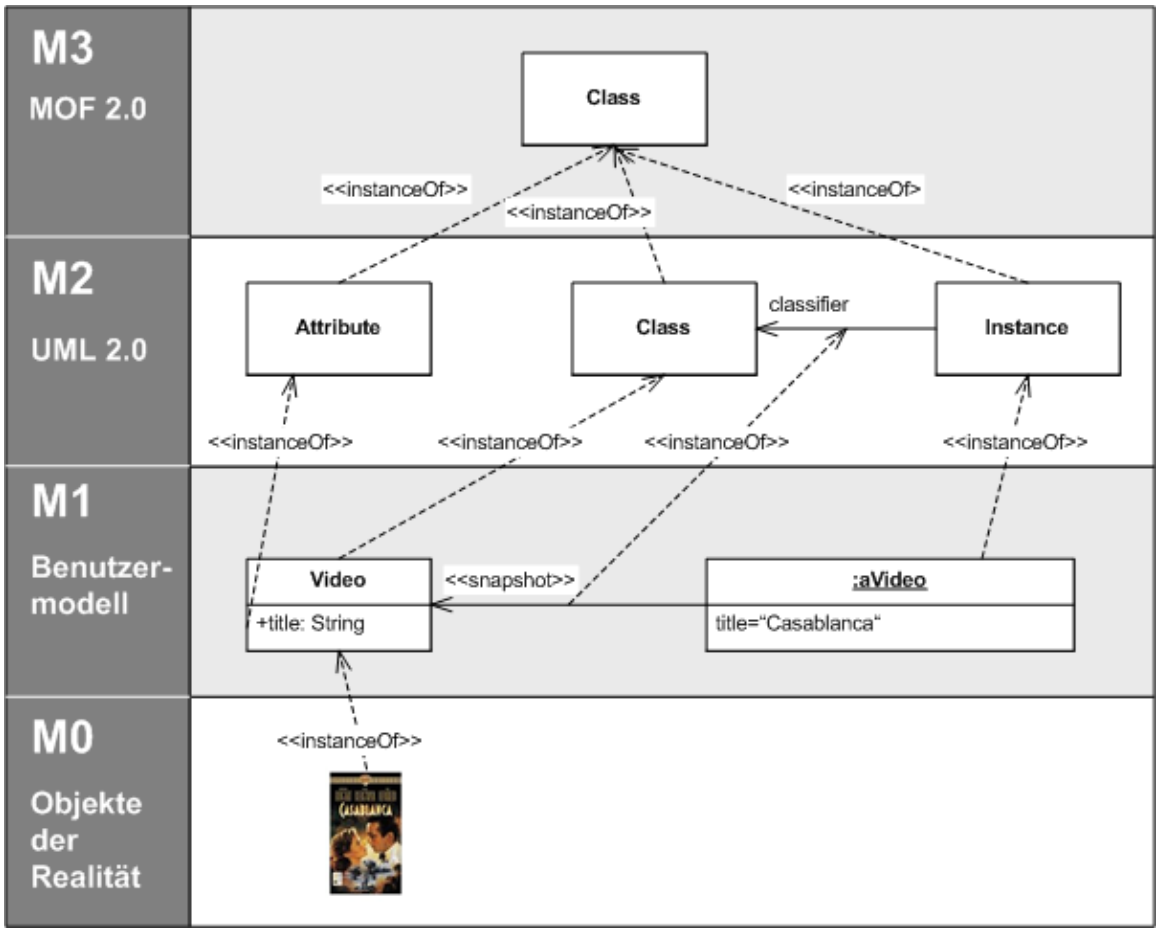
"Metamodeling" is the construction of a collection of "concepts" (things, terms, etc.) within a certain domain. A model is an abstraction of phenomena in the real world; a metamodel is yet another abstraction, highlighting properties of the model itself. A model conforms to its metamodel in the way that a computer program conforms to the grammar of the programming language in which it is written.

Common uses for metamodels are:

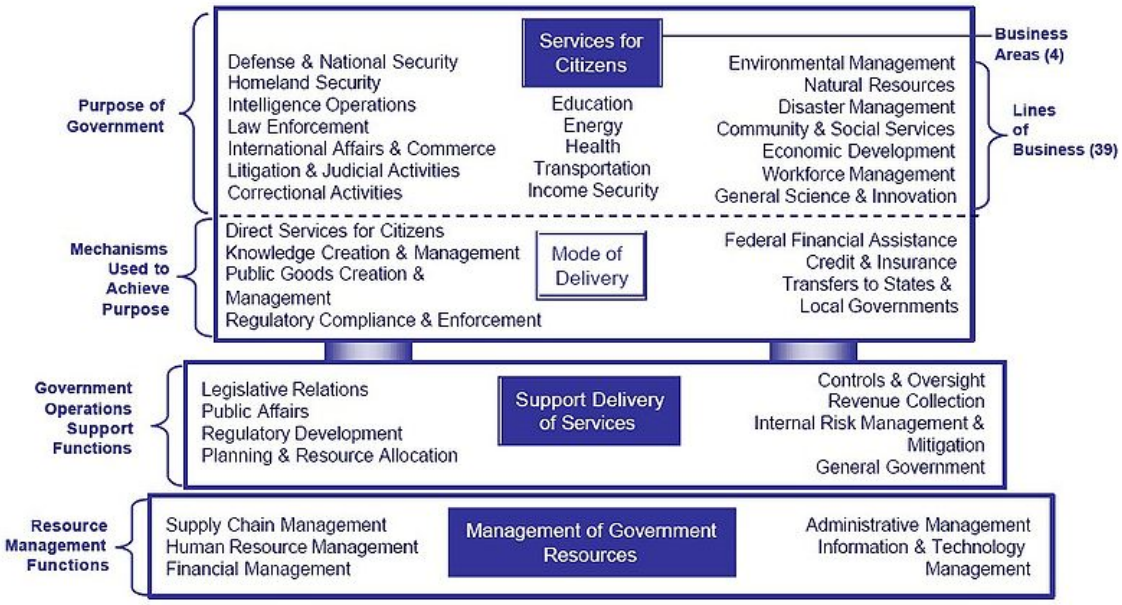
- As a schema for semantic data that needs to be exchanged or stored
- As a language that supports a particular method or process
- As a language to express additional semantics of existing information

Because of the "meta" character of metamodeling, both the praxis and theory of metamodels are of relevance to metascience, metaphilosophy, metatheories and systemics, and meta-consciousness. The concept can be useful in mathematics, and has practical applications in computer science and computer engineering/software engineering, which are the main focus of this article.

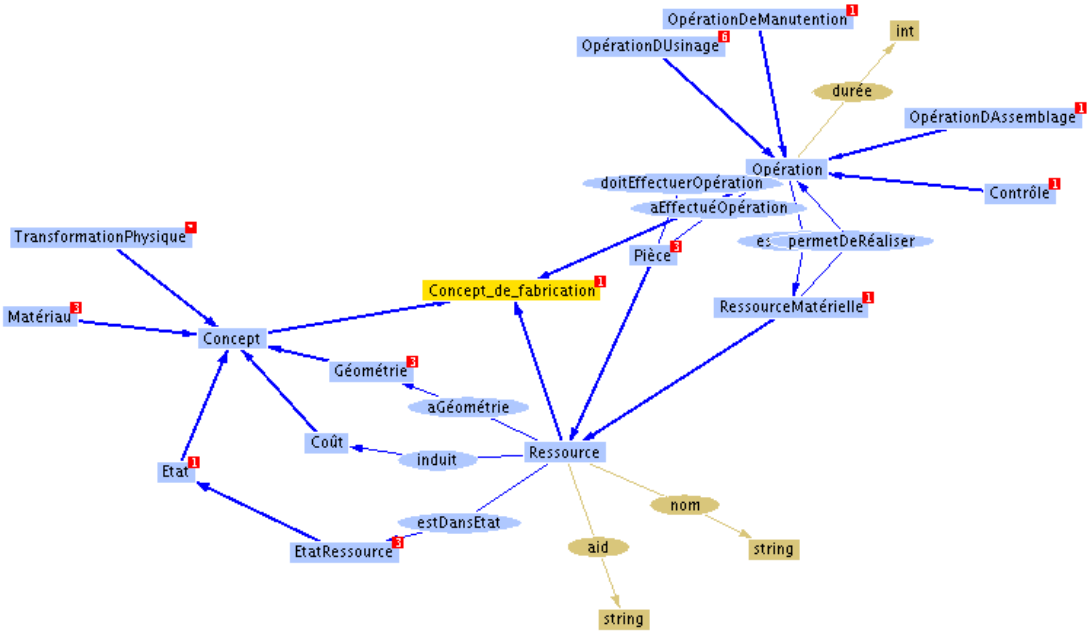
Metamodeling topics



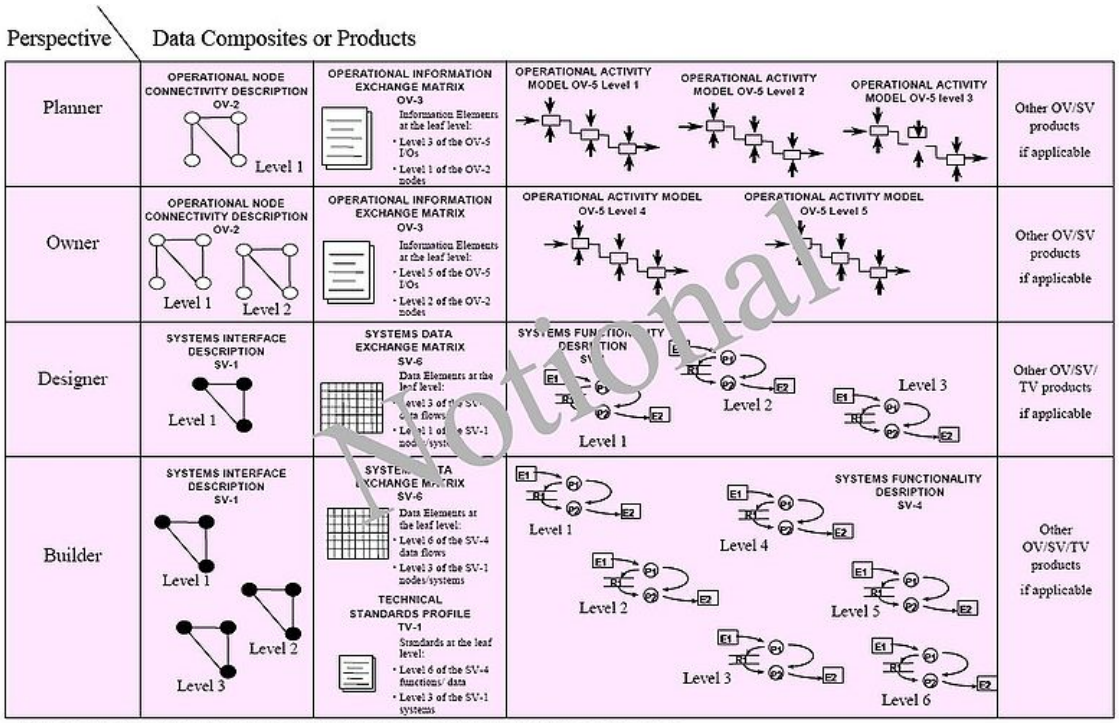
Meta-Object Facility Illustration.



An US FEA Business reference model.



Example of an ontology.



A DoDAF metamodel.

Definition

In software engineering, the use of models is more and more recommended. This should be contrasted with the classical code-based development techniques. A model always conforms to a unique metamodel. One of the currently most active branch of Model Driven Engineering is the approach named model-driven architecture proposed by OMG. This approach is based on the utilization of a language to write metamodels called the Meta Object Facility or MOF. Typical metamodels proposed by OMG are UML, SysML, SPEM or CWM. ISO has also published the standard metamodel ISO/IEC 24744. All the languages presented below could be defined as MOF metamodels.

Metadata modeling

Metadata modeling is a type of metamodeling used in software engineering and systems engineering for the analysis and construction of models applicable and useful to some predefined class of problems.

Model transformations

One important move in Model Driven Engineering is the systematic use of Model Transformation Languages. The OMG has proposed a standard for this called QVT for Queries/Views/Transformations. QVT is based on the Meta-Object Facility or MOF.

Among many other Model Transformation Languages (MTLs), some examples of implementations of this standard are AndroMDA, VIATRA, Tefkat, MT, ManyDesigns Portofino.

Relationship to ontologies

Meta-models are closely related to ontologies. Both are often used to describe and analyze the relations between concepts

- Ontologies: express something meaningful within a specified universe or domain of discourse by utilizing a grammar for using vocabulary. The grammar specifies what it means to be a well-formed statement, assertion, query, etc. (formal constraints) on how terms in the ontology's controlled vocabulary can be used together.
- Meta-modeling: can be considered as an *explicit* description (constructs and rules) of how a domain-specific model is built. In particular, this comprises a formalized specification of the domain-specific notations. Typically, metamodels are – and always should follow - a strict rule set. . "A valid metamodel is an ontology, but not all ontologies are modeled *explicitly* as metamodels".

Types of meta-models

For software engineering, several *types* of models (and their corresponding modeling activities) can be distinguished:

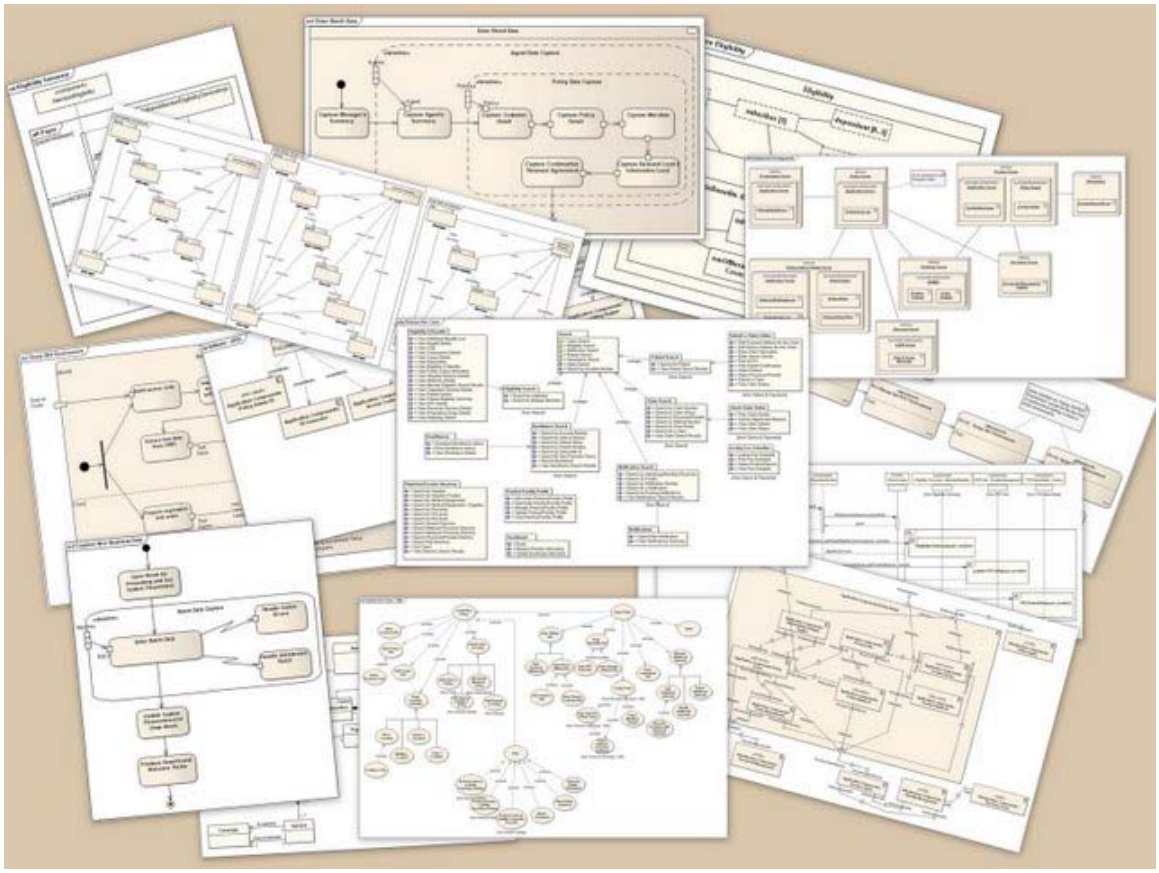
- Metadata modeling (MetaData Model)
- Meta-Process Modeling (MetaProcess Model)
- Executable Meta-Modeling (combining both of the above and much more, as in the general purpose tool Kermeta)
- Model Transformation Language (see below)

Zoos of metamodels

A library of similar meta-models has been called a Zoo of meta-models. There are several types of meta-model zoos. Some are expressed in ECore. Others are written in MOF 1.4 - XMI 1.2. The metamodels expressed in UML-XMI1.2 may be uploaded in Poseidon for UML, a UML CASE tool.

Chapter 5

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

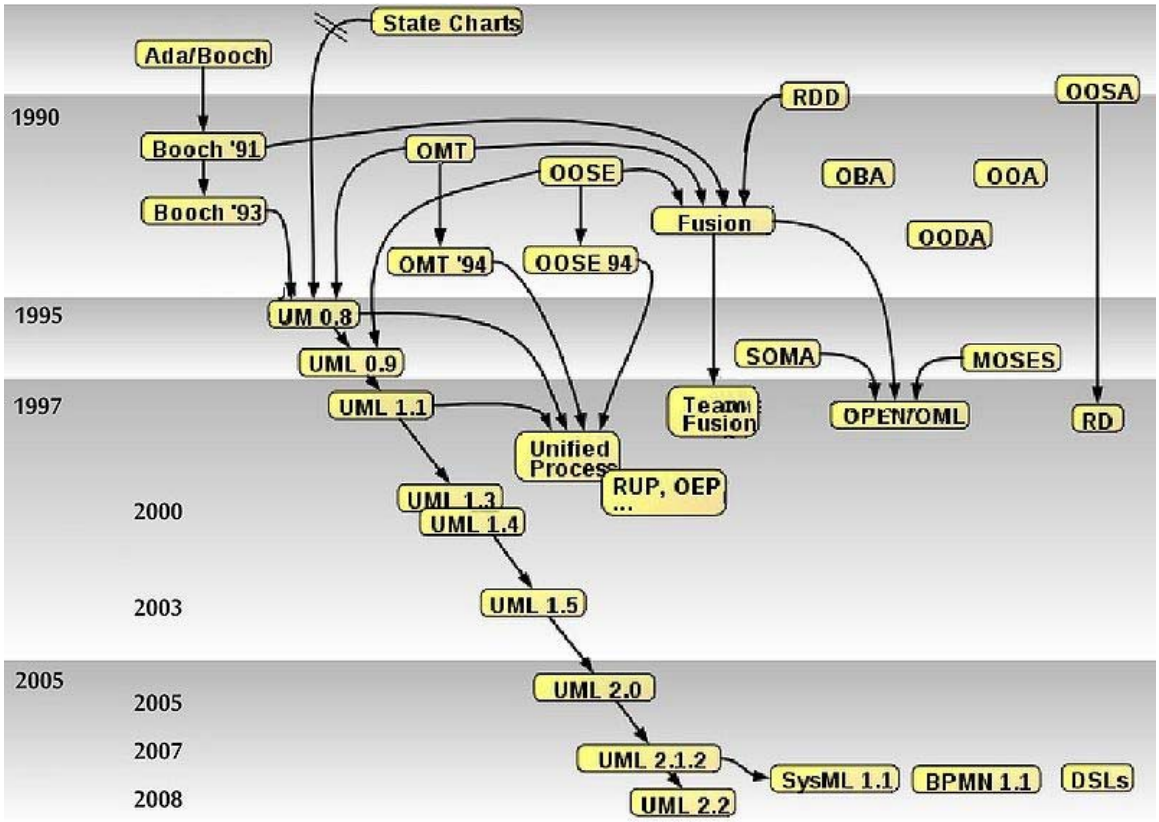
The Unified Modeling Language (UML) is used to specify, visualize, modify, construct and document the artifacts of an object-oriented software-intensive system under development. UML offers a standard way to visualize a system's architectural blueprints, including elements such as:

- 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:

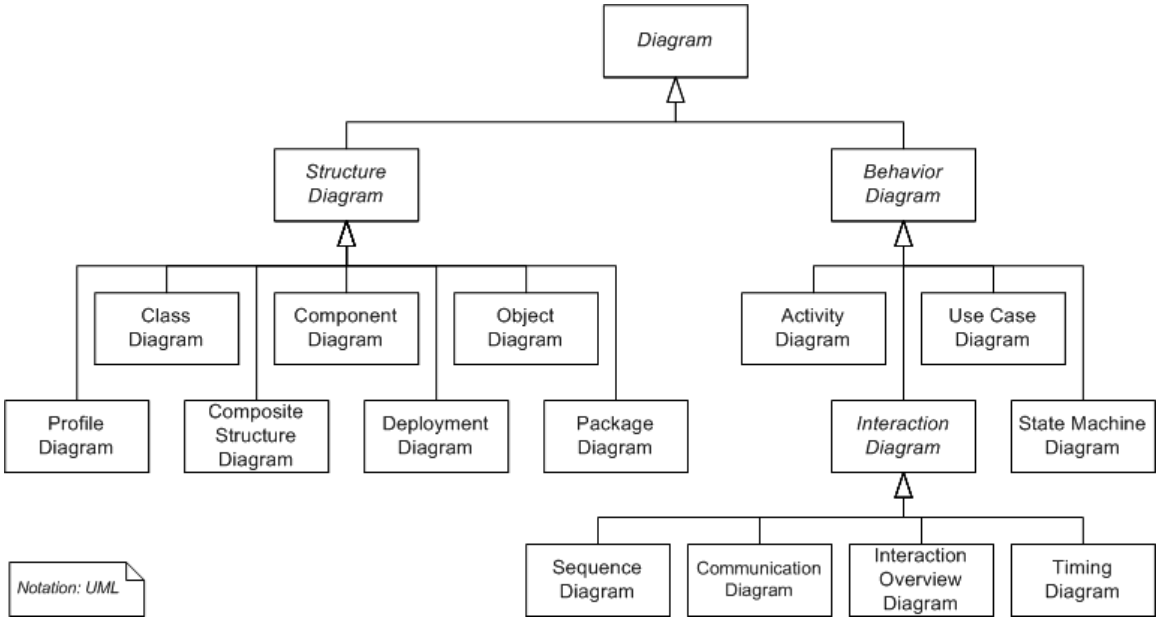
- Static (or *structural*) view: emphasizes the static structure of the system using objects, attributes, operations and relationships. The structural view includes class diagrams and composite structure diagrams.
- Dynamic (or *behavioral*) view: emphasizes the dynamic behavior of the system by showing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

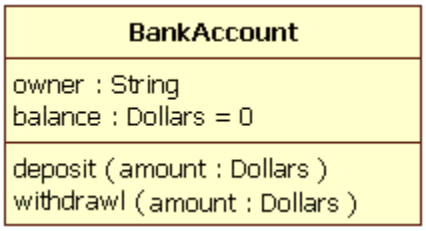
In keeping with the tradition of engineering drawings, a comment or note explaining usage, constraint, or intent is allowed in a UML diagram.

Structure diagrams

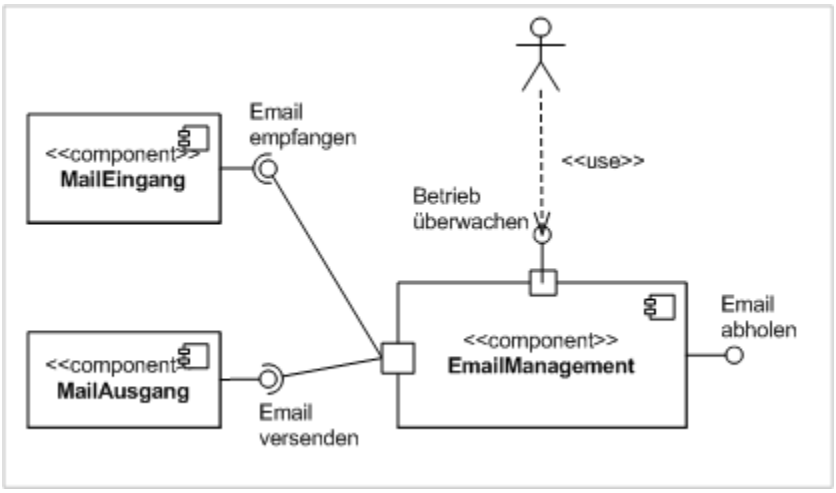
Structure diagrams emphasize the things that must be present in the system being modeled. Since structure diagrams represent the structure, they are used extensively in documenting the software architecture of software systems.

- 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.

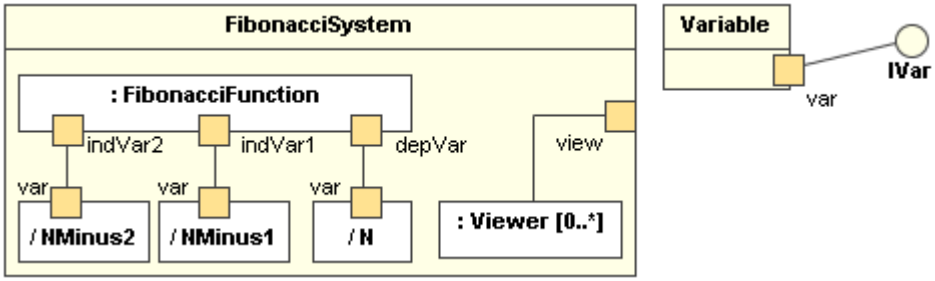
- Profile diagram: operates at the metamodel level to show stereotypes as classes with the <<stereotype>> stereotype, and profiles as packages with the <<profile>> stereotype. The extension relation (solid line with closed, filled arrowhead) indicates what metamodel element a given stereotype is extending.



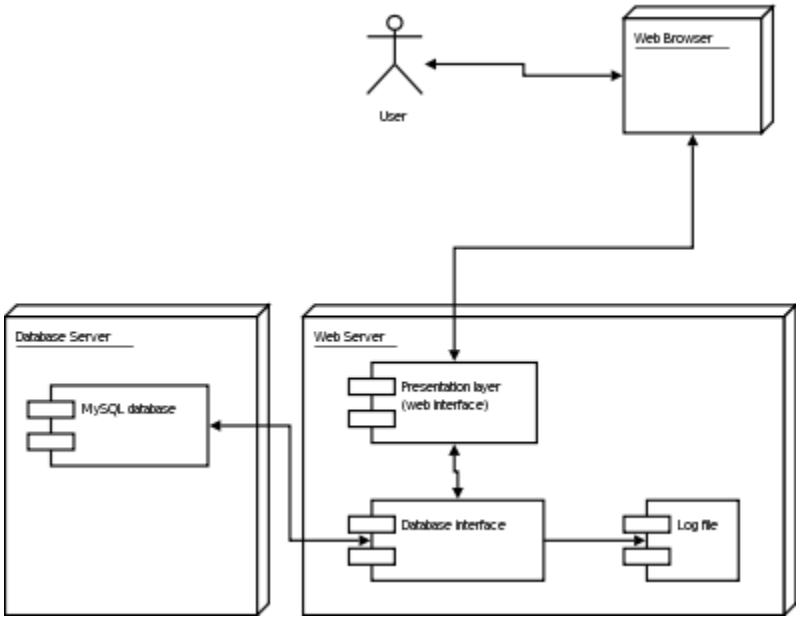
Class diagram



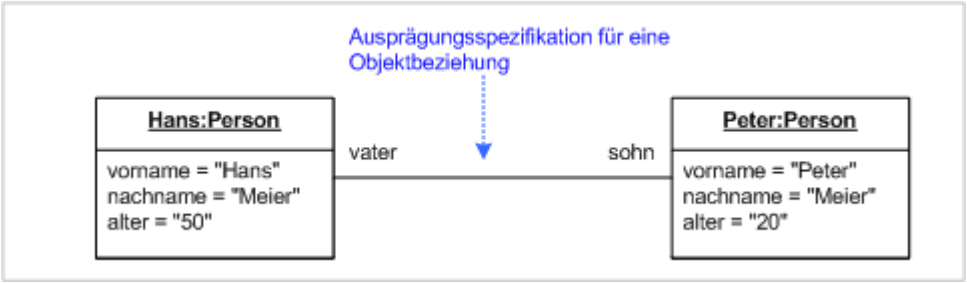
Component diagram



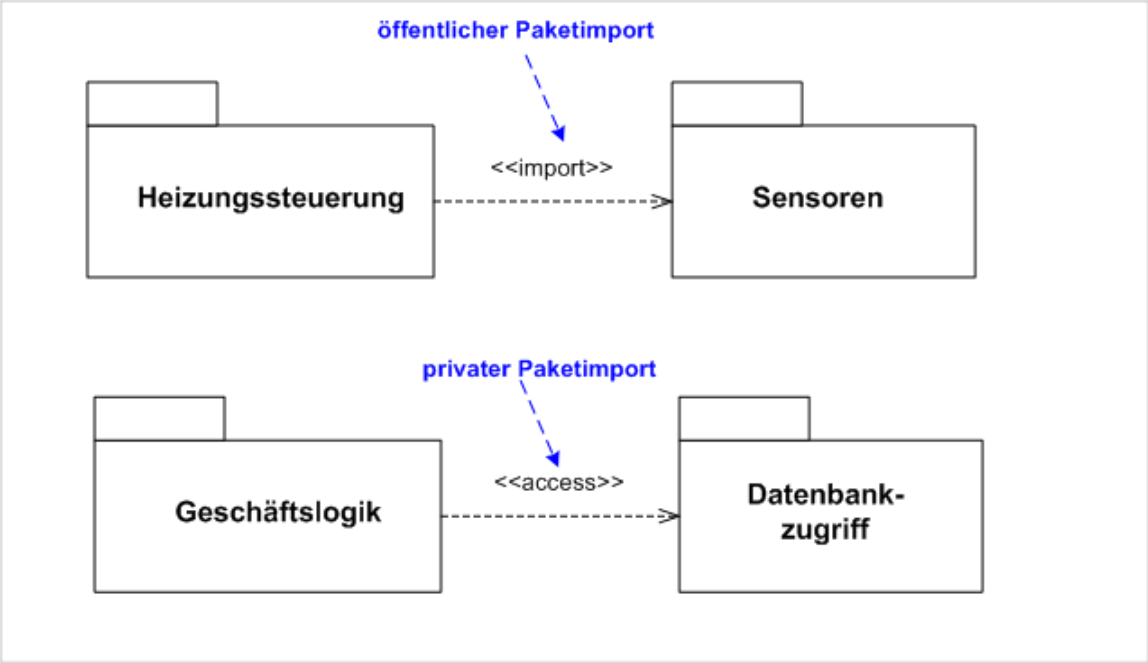
Composite structure diagrams



Deployment diagram



Object diagram

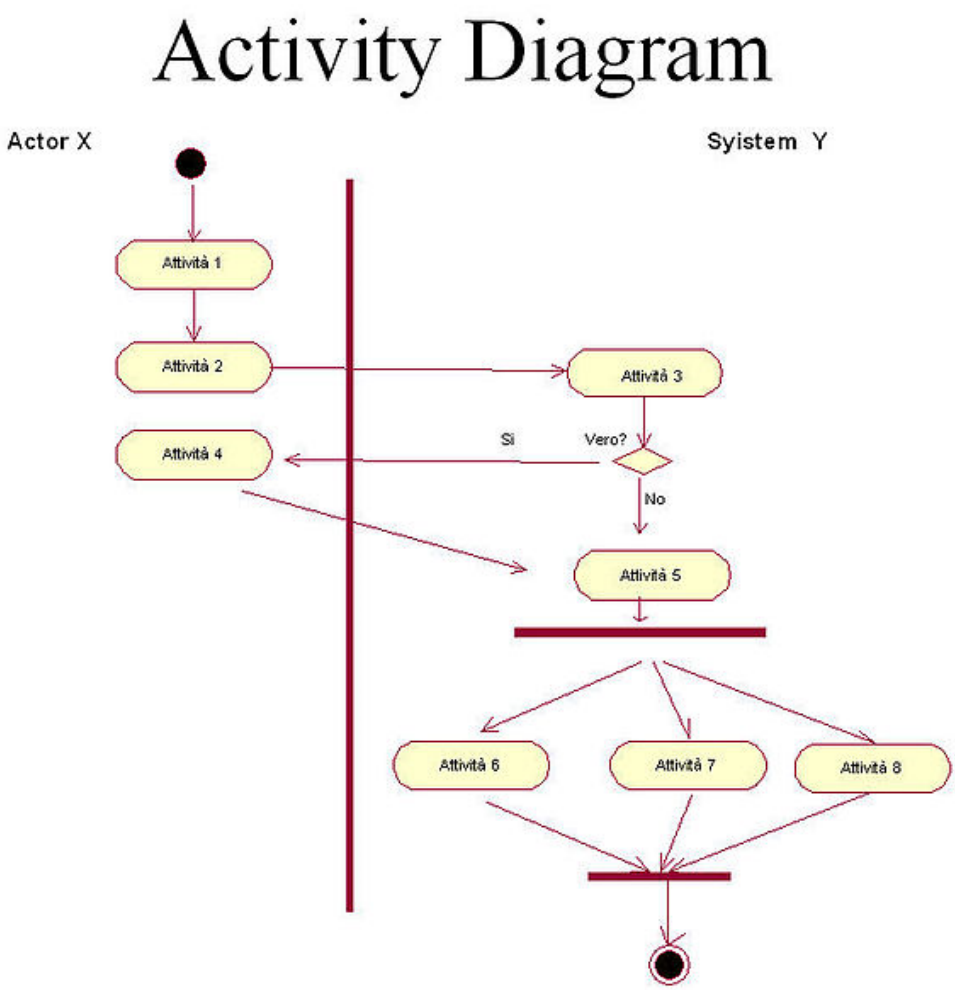


Package diagram

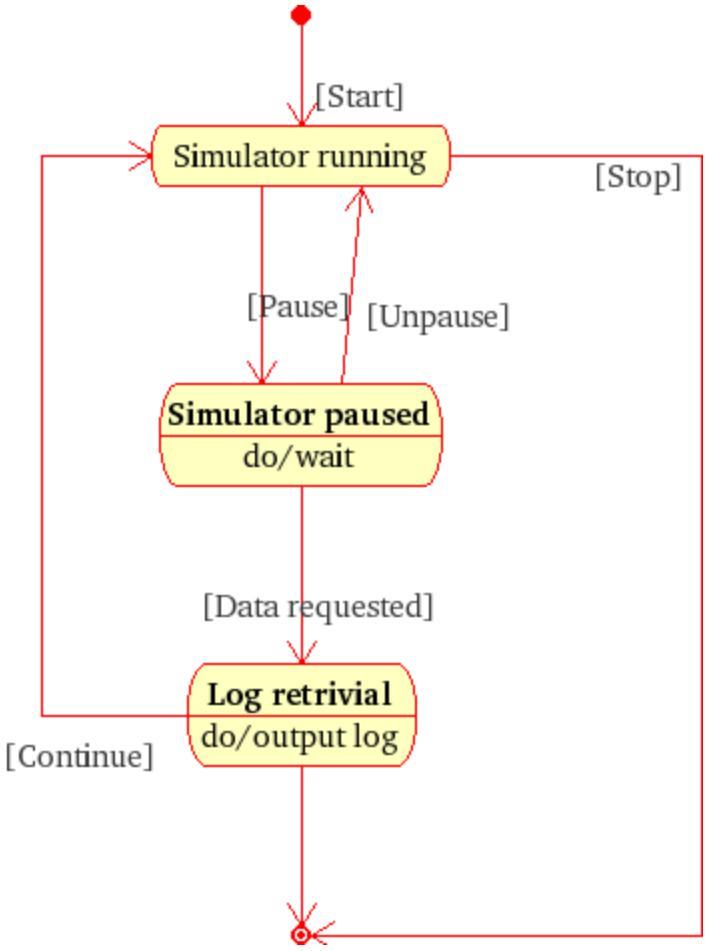
Behaviour diagrams

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.

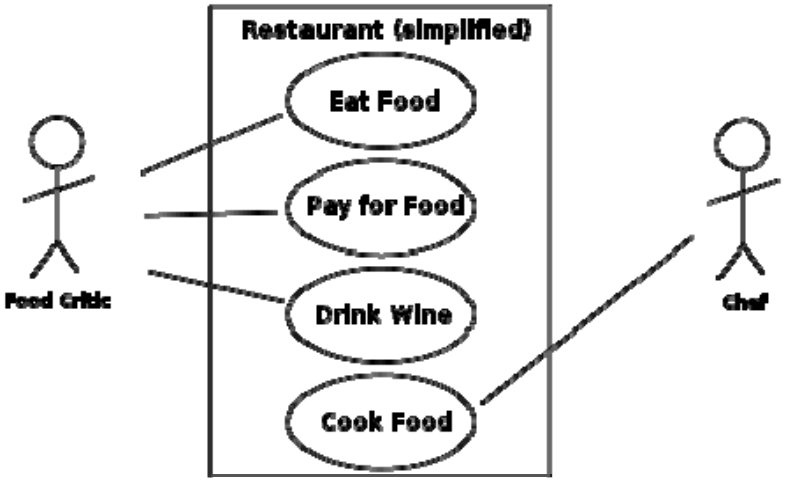
- Activity diagram: describes the business and operational step-by-step workflows of components in a system. An activity diagram shows the overall flow of control.
- UML state machine diagram: describes the states and state transitions of the system.
- Use case diagram: describes the functionality provided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

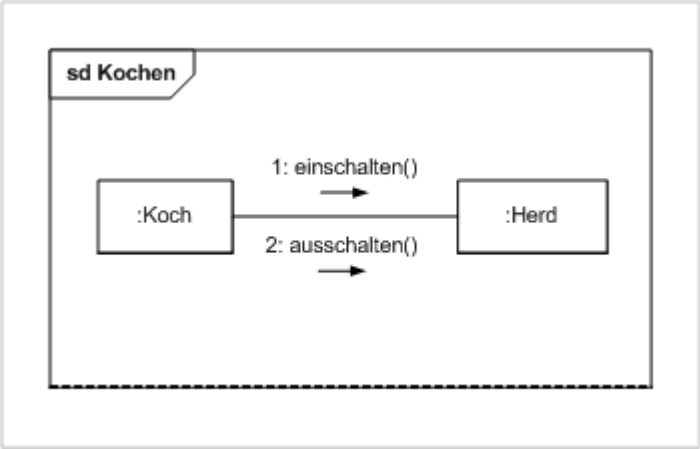


Use case diagram

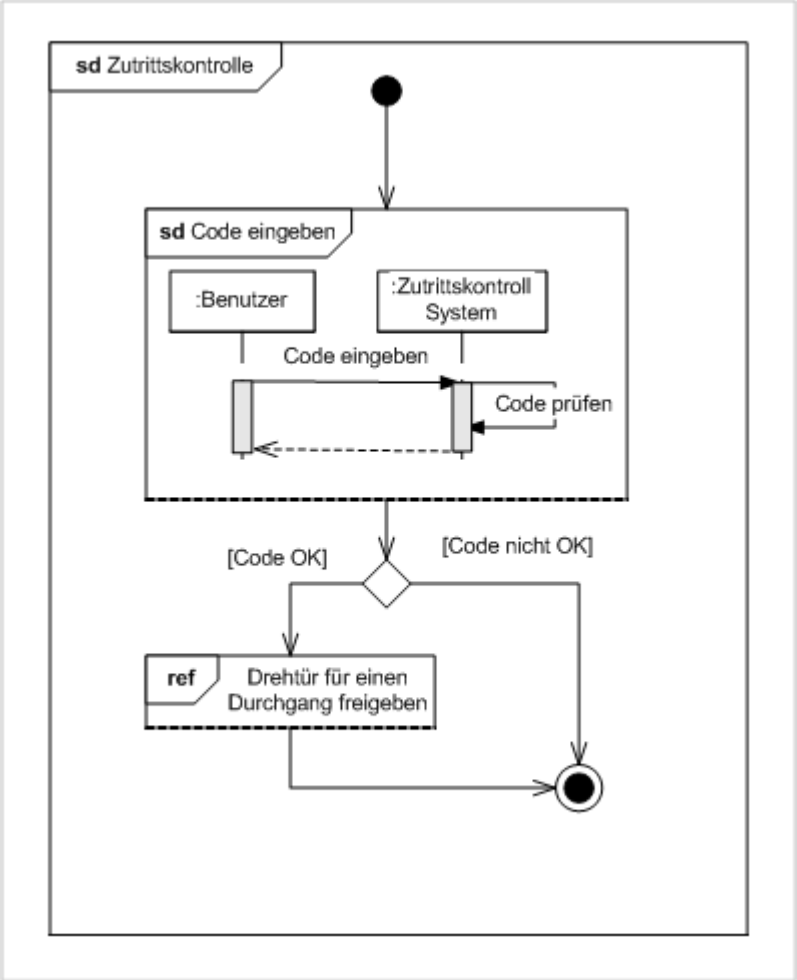
Interaction diagrams

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

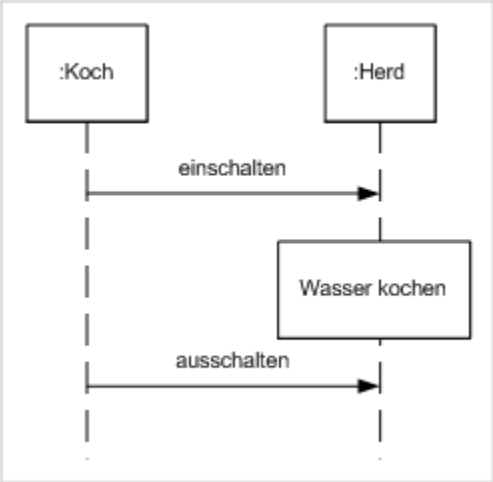
- Communication diagram: shows the interactions between objects or parts in terms of sequenced messages. They represent a combination of information taken from Class, Sequence, and Use Case Diagrams describing both the static structure and dynamic behavior of a system.
- Interaction overview diagram: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

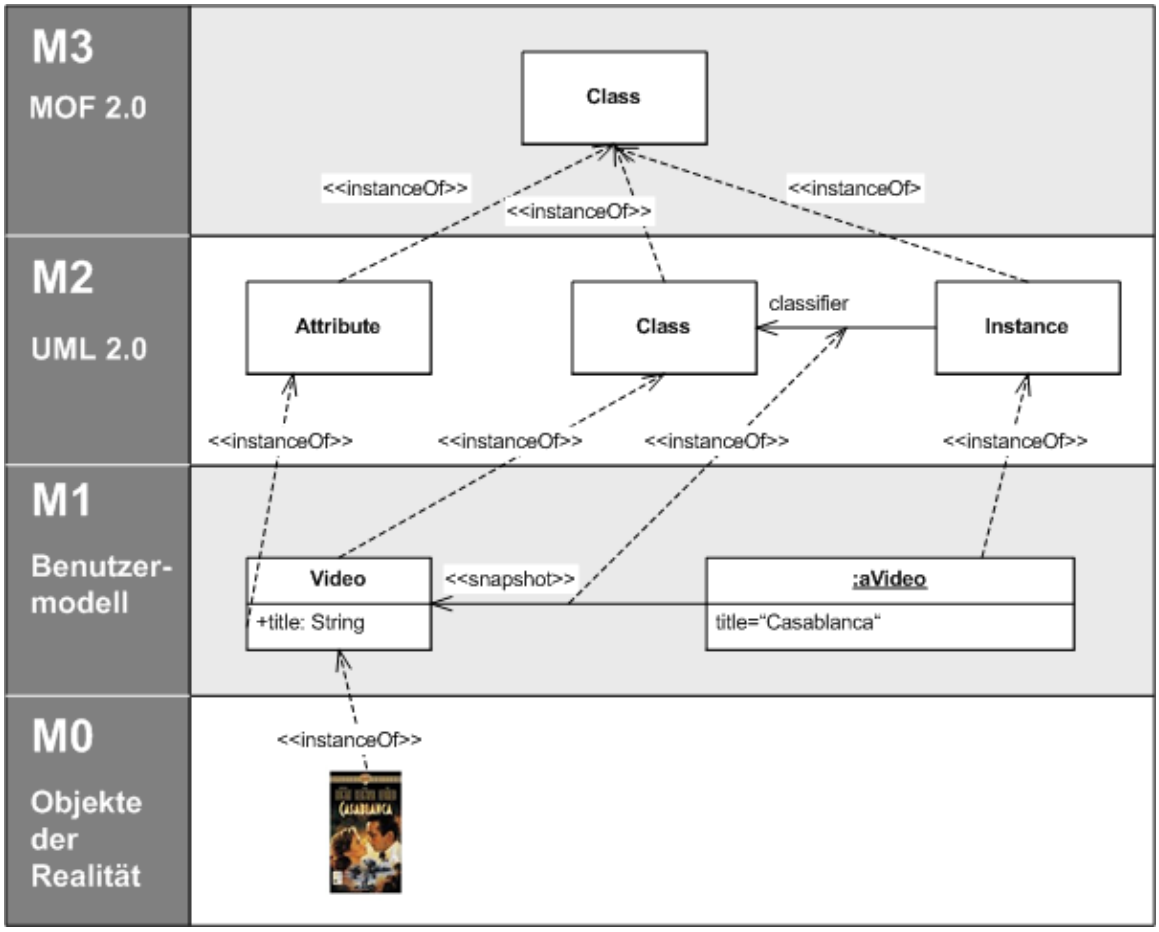


Illustration of the Meta-Object Facility.

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 here 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 6

Metadata Modeling

Metadata modeling is a type of metamodeling used in software engineering and systems engineering for the analysis and construction of models applicable and useful some predefined class of problems.

Meta-modeling is the analysis, construction and development of the frames, rules, constraints, models and theories applicable and useful for the modeling in a predefined class of problems.

The meta-data side of the diagram consists of a concept diagram. This is basically an adjusted class diagram as described in Booch, Rumbaugh and Jacobson (1999). Important notions are concept, generalization, association, multiplicity and aggregation.

Metadatamodeling Concepts

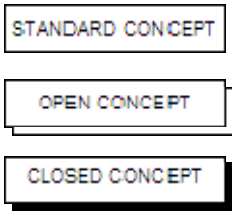


Fig.1 STANDARD, OPEN and CLOSED CONCEPTS

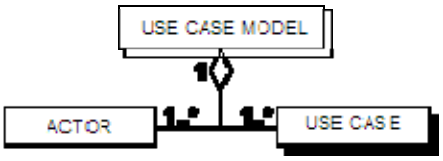


Fig.2 Example of STANDARD, OPEN and CLOSED CONCEPTS

First of all, a concept is a simple version of a UML class. The class definition is adopted to define a concept, namely: a set of objects that share the same attributes, operations, relations, and semantics.

The following concept types are specified:

- STANDARD CONCEPT: a concept that contains no further (sub) concepts. A standard concept is visualized with a rectangle.
- COMPLEX CONCEPT: a concept that consists of a collection of (sub) concepts. Complex concepts are divided into:
- OPEN CONCEPT: a complex concept whose (sub) concepts are expanded. An open concept is visualized with two white rectangles above each other.
- CLOSED CONCEPT: a complex concept whose (sub) concepts are not expanded since it is not relevant in the specific context. A closed concept is visualized by a white rectangle above a black rectangle.

In Figure 1 the three concept types that are used in the modeling technique are illustrated. Concepts are always capitalized, not only in the diagram, but also when referring to them outside the diagram.

In Figure 2 all three concept types are exemplified. Part of the process-data diagram of the requirements workflow in the Unified Process is illustrated. The USE CASE MODEL is an open concept and consists of one or more ACTORS and one or more USE CASES. ACTOR is a standard concept, it contains no further sub-concepts. USE CASE, however, is a closed concept. A USE CASE consists of a description, a flow of events, conditions, special requirements, etc. Because in this case it is unnecessary to reveal that information, the USE CASE is illustrated with a closed concept.

Generalization

Fig.3 Generalization

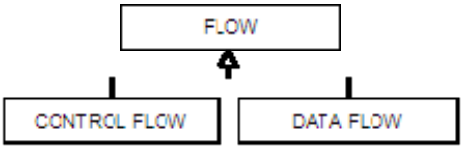


Fig. Example generalization

Generalization is a way to express a relationship between a general concept and a more specific concept. Also, if necessary, one can indicate whether the groups of concepts that are identified are overlapping or disjoint, complete or incomplete. Generalization is visualized by a solid arrow with an open arrowhead, pointing to the parent, as is illustrated in Figure 3.

In Figure 4 generalization is exemplified by showing the relationships between the different concepts described in the preceding paragraph. STANDARD CONCEPT and COMPLEX CONCEPT are both a specific kind of CONCEPT. Subsequently, a COMPLEX CONCEPT can be specified into an OPEN CONCEPT and a CLOSED CONCEPT.

Association

Fig.5 Association

An association is a structural relationship that specifies how concepts are connected to another. It can connect two concepts (binary association) or more than two concepts (n-ary association). An association is represented with an undirected solid line. To give a meaning to the association, a name and name direction can be provided. The name is in the form of an active verb and the name direction is represented by a triangle that points in the direction one needs to read. Association with a name and name direction is visualized in Figure 5.

In Figure 6 (removed) an example of association is illustrated. The example is a fragment of the process-data diagram of the requirements analysis in the Unified Process. Because both concepts are not expanded any further, although several sub concepts exist, the concepts are illustrated as closed concepts. The figure reads as “SURVEY DESCRIPTION describes USE CASE MODEL”.

Multiplicity

Fig.7 Multiplicity

Fig.8 Example multiplicity

Except name and name direction, an association can have more characteristics. With multiplicity one can state how many objects of a certain concept can be connected across an instance of an association. Multiplicity is visualized by using the following expressions: (1) for exactly one, (0..1) for one or zero, (0..*) for zero or more, (1..*) for one or more, or for example (5) for an exact number. In Figure 7 association with multiplicity is visualized.

An example of multiplicity is represented in Figure 8. It is the same example as in Figure 6, only the multiplicity values are added. The figure reads as ‘exactly one SURVEY DESCRIPTION describes exactly one USE CASE MODEL’. This implies that a SURVEY DESCRIPTION cannot describe zero or more than one USE CASE MODEL and a USE CASE MODEL cannot be described by zero or more than one SURVEY DESCRIPTIONS.

Aggregation

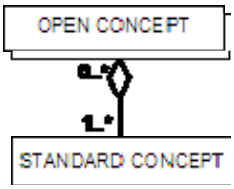


Fig.9 Aggregation

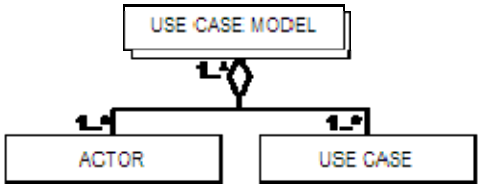


Fig.10 Example aggregation

A special type of association is aggregation. Aggregation represents the relation between a concept (as a whole) containing other concepts (as parts). It can also be described as a 'has-a' relationship. In Figure 3-9 an aggregation relationship between OPEN CONCEPT and STANDARD CONCEPT is illustrated. An OPEN CONCEPT consists of one or more STANDARD CONCEPTS and a STANDARD CONCEPT is part of one OPEN CONCEPT.

In Figure 10 aggregation is exemplified by a fragment of the requirements capture workflow in UML-Based Web Engineering. A USE CASE MODEL consists of one or more ACTORS and USE CASES.

Properties

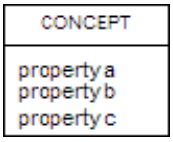


Fig.11 Aggregation

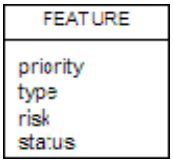


Fig.12 Example aggregation

Sometimes the needs exist to assign properties to concepts. Properties are written in lower case, under the concept name, as is illustrated in Figure 11.

In Figure 12 an example of a concept with properties is visualized. The concept FEATURE has four properties, respectively: priority, type, risk and status.

In Table 1 a list presented Each CONCEPT requires a proper definition which is preferably copied from a standard glossary. All CONCEPT names in the text are with capital characters.

Chapter 7

Domain-Specific Multimodeling

Domain-specific multimodeling is a software development paradigm where each view is made explicit as a separate domain-specific language (DSL).

Successful development of a modern enterprise system requires the convergence of multiple views. Business analysts, domain experts, interaction designers, database experts, and developers with different kinds of expertise all take part in the process of building such a system. Their different work products must be managed, aligned, and integrated to produce a running system. Every participant of the development process has a particular language tailored to solve problems specific to its view on the system. The challenge of integrating these different views and avoiding the potential cacophony of multiple different languages is *the coordination problem*.

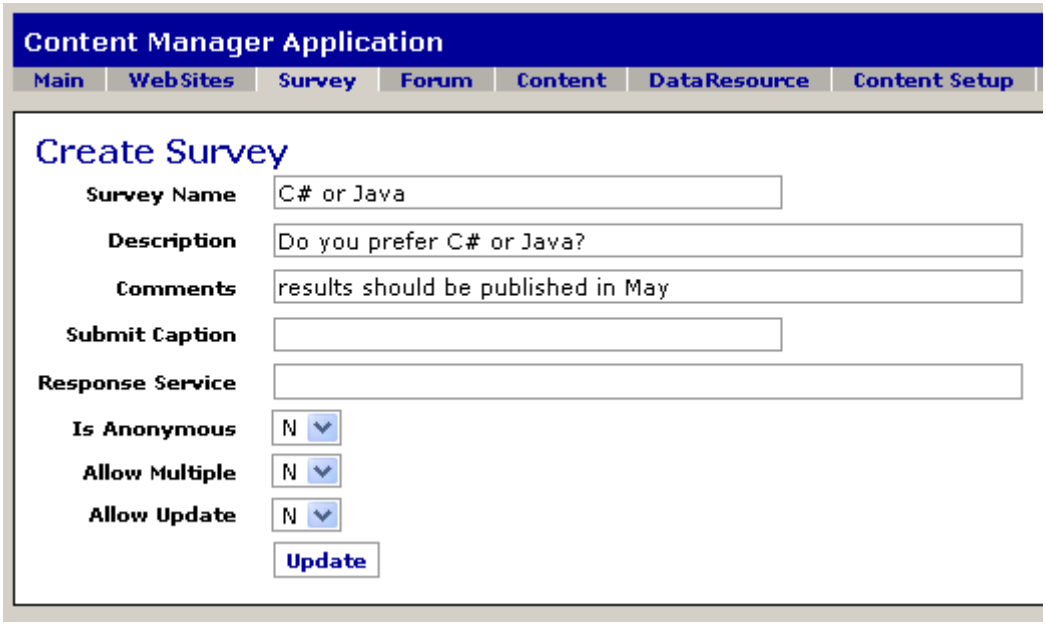
Domain-specific multimodeling is promising when compared to more traditional development paradigms such as single-language programming and general-purpose modeling. To reap the benefits of this new paradigm, we must solve the coordination problem. This problem is also known as the fragmentation problem in the context of Global Model Management.

One proposal to solve this problem is *the coordination method*. This is a three-step method to overcome the obstacles of integrating different views and coordinating multiple languages. The method prescribes how to (1) identify and (2) specify the references across language boundaries, that is the *overlaps* between different languages. Finally, the method offers concrete proposals on how to (3) apply this knowledge in actual development in the form of consistency, navigation, and guidance.

Motivating example

Enterprise systems based on multiple domain-specific languages are abundant. Languages with a metamodel defined in the Extensible Markup Language (XML) enjoy particularly widespread adoption. To illustrate development with multiple languages, we will draw an example from a case study: The Apache Open For Business (OFBiz) system. Briefly stated, OFBiz is an enterprise resource planning system that includes standard components such as inventory, accounting, e-commerce etc. These components

are implemented by a mixture of XML-based languages and regular Java code. As an example, let us focus on the content management component, particularly a use case in which the administrative user creates an online web survey as shown in the screenshot below. We will refer to this example as the *create survey* example.



The figure shows a screenshot of the administrative interface of the content management application in a running OFBiz instance. To create a survey, the user fills out the fields of the input form and hits the *update* button. This creates a new survey which can be edited and later published on a frontend website in OFBiz. Behind the scenes, this use case involves several artifacts written in different languages. In this example, let us focus on only three of these languages: the Entity, the Service, and the Form DSL.

These three languages correspond roughly to the structural, the behavioural, and the user interface concern in OFBiz. The Entity DSL is used to describe the underlying data model and hence the way the created survey will be saved. The Service DSL is used to describe the interface of the service that is invoked when the user hits the *update* button. Finally, the Form DSL is used to describe the visual appearance of the form. Although the three languages are tailored for different things, they can not be separated entirely. The user interface invokes a certain application logic and this application logic manipulates the data of the application. This is an example of *non-orthogonal concerns*. The languages overlap because the concerns that they represent cannot be separated entirely. Let us examine these three languages in a bottom-up manner and point out their overlaps.

Entity DSL

The Entity DSL defines the structure of data in OFBiz. The listing below shows the definition of the Survey entity which is the business object that represents the concept of a survey. The code in the Listing is self-explanatory: An entity called Survey is defined

with 10 fields. Each field has a name and a type. The field surveyId is used as the primary key. This definition is loaded by a central component in OFBiz called the *entity engine*. The entity engine instantiates a corresponding business object. The purpose of the entity engine is to manage transactional properties of all business objects and interact with various persistence mechanisms such as Java Database Connectivity, Enterprise JavaBeans or even some legacy system.

```
<entity entity-name="Survey" ... title="Survey Entity">
  <field name="surveyId" type="id-ne"/>
  <field name="surveyName" type="name"/>
  <field name="description" type="description"/>
  <field name="comments" type="comment"/>
  <field name="submitCaption" type="short-varchar"/>
  <field name="responseService" type="long-varchar"/>
  <field name="isAnonymous" type="indicator" .../>
  <field name="allowMultiple" type="indicator" .../>
  <field name="allowUpdate" type="indicator" .../>
  <field name="acroFormContentId" type="id-ne" .../>
  <prim-key field="surveyId"/>
</entity>
```

Service DSL

The Service DSL specifies the interface of the services in OFBiz. Each service encapsulates part of the application logic of the system. The purpose of this language is to have a uniform abstraction over various implementing mechanisms. Individual services can be implemented in Java, a scripting language, or using a rule engine. The listing below shows the interface of the createSurvey service.

Apart from the name, the service element specifies the location and invocation command of the implementation for this service. The default-entity-name attribute specifies that this service refers to the Survey entity which was defined in the previous listing. This is an overlap between the two languages, specifically a so-called *soft reference*. A model in the Service DSL refers to a model in the Entity DSL. This reference is used in the two auto-attributes elements below which specify the input and output of the service in the form of typed attributes. As input, the service accepts attributes corresponding to all non-primary key (nonpk) fields of the Survey entity and these attributes are optional. As output, the service returns attributes corresponding to the primary key (pk) fields of Survey, i.e., in this case the surveyId field, and these attributes are mandatory. The purpose of the reference across languages is in this case to reduce redundancy. The attributes of the createSurvey service corresponds to the fields of the Survey entity and it is therefore only necessary to specify them once.

```
<service name="createSurvey" default-entity-name="Survey" ...
         location="org/ofbiz/content/survey/SurveyServices.xml"
         invoke="createSurvey"> ...
  <permission-service service-name="contentManagerPermission"
                      main-action="CREATE"/>
  <auto-attributes include="nonpk" mode="IN" optional="true"/>
```

```
  <auto-attributes include="pk" mode="OUT" optional="false"/>
</service>
```

Form DSL

The Form DSL is used to describe the layout and visual appearance of input forms in the user interface. The language consists of domain concepts such as Form and Field. The listing below shows the implementation of the EditSurvey form. This time the Form DSL overlaps with the Service DSL. The target attribute of the form and the alt-target elements specify that the input from the submission of this form should be directed to either the updateSurvey or createSurvey services. The auto-fields-service element specifies that the form should include a field corresponding to each of the attributes of the updateSurvey service (which are similar to the attributes of the createSurvey service). This produces a similar effect of *importing* definitions from another model as in the case of the auto-attributes elements in the previous listing. Further down, we can see that it is possible to customize the appearance of these *imported* fields such as isAnonymous. Finally, a submitButton is added with a localized title such that the user can submit his data to the referenced service.

```
<form name="EditSurvey" type="single" target="updateSurvey"
      title="" default-map-name="survey">
  <alt-target use-when="survey==null" target="createSurvey"/>
  <auto-fields-service service-name="updateSurvey"/>
  <field use-when="survey!=null" name="surveyId" ... />
  ...
  <field name="isAnonymous">
    <drop-down no-current-selected-key="N" allow-empty="false">
      <option key="Y"/><option key="N"/>
    </drop-down>
  </field>
  ...
  <field name="submitButton" title="${uiLabelMap.CommonUpdate}"
         widget-style="smallSubmit">
    <submit button-type="button"/>
  </field>
</form>
```

The *create survey* example, as described here, is implemented using models in three different languages. The complete implementation actually involves even more languages such as a Screen DSL to specify the layout of the screen where the form is placed, and a Minilang DSL which is a data-manipulation language used to implement the service. However, these three languages do illustrate the main idea of making each concern concrete. The example also shows a simple way of reducing redundancy by letting the languages overlap slightly.

Multi-level customization

Domain-specific languages, like those described above, have limited expressiveness. It is often necessary to add code snippets in a general-purpose language like Java to implement specialized functionality that is beyond the scope of the languages. This method is called *multi-level customization*. Since this method is very commonly used in setups with multiple languages, we will illustrate it by a continuation of the example. Let us call this the *build PDF* example.

Suppose we want to build a PDF file for each survey response to the online surveys that users create. Building a PDF file is outside the scope of our languages so we need to write some Java code that can invoke a third-party PDF library to perform this specialized functionality. Two artifacts are required:

First, an additional service model, as shown below, in the Service DSL that defines the interface of the concrete service such that it can be accessed on the modeling level. The service model describes the location of the implementation and what the input and output attributes are.

```
<service name="buildPdfFromSurveyResponse" engine="java"
  location="org.ofbiz.content.survey.PdfSurveyServices"
  invoke="buildPdfFromSurveyResponse">
  <attribute name="surveyResponseId" mode="IN"
    optional="false" .../>
  <attribute name="outByteWrapper" mode="OUT"
    optional="false" .../>
</service>
```

Second, we need a code snippet, as shown below, that contains the actual implementation of this service. A service can have multiple inputs and outputs so input to the Java method is a map, called context, from argument names to argument values and returns output in the form of another map, called results.

```
public static Map buildPdfFromSurveyResponse
(DispatchContext dctx , Map context) {
  String id = (String) context.get("surveyResponseId");
  Map results = new HashMap();
  try {
    ...the response is retrieved from the database...
    ...a pdf is built from the response...
    ...the pdf is serialized as a bytearray...
  ByteWrapper outByteWrapper = ...;
  results.put("outByteWrapper",outByteWrapper );
  } catch (Exception e) {}
  return results;
}
```

This multi-level customization method uses *soft references* similar to the *create survey* example. The main difference is that the reference here is between model and code rather than between model and model. The advantage, in this case, is that a third-party Java library for building PDFs can be leveraged. Another typical application is to use Java code snippets to invoke external webservices and import results in a suitable format.

Coordination problem

The example illustrates some of the advantages of using multiple languages in development. There are, however, also difficulties associated with this kind of development. These difficulties stem from the observation that the more kinds of artifacts we introduce into our process, the more coordination between developer efforts is needed. We will refer to these difficulties as the *Coordination Problem*. The Coordination Problem has a conceptual and a technical aspect. Conceptually, the main problem is to understand the different languages and their interaction. To properly design and coordinate models in multiple languages, developers must have a sufficient understanding of how languages interact. Technically, the main problem is to enforce consistency. Tools must be provided to detect inconsistencies early, i.e., at modeling time, and assist developers in resolving these inconsistencies. In the following, we will examine these two aspects in greater detail.

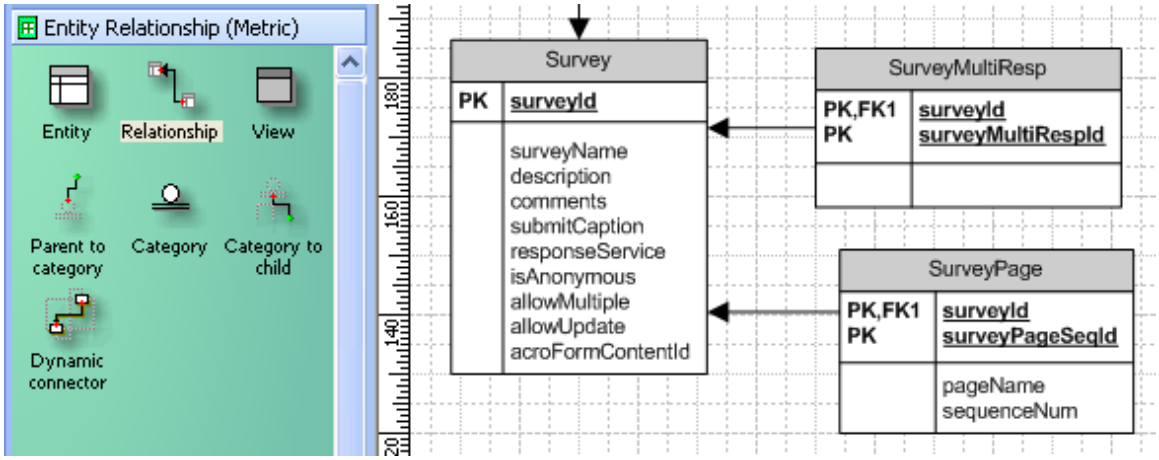
Coordination as a conceptual challenge

The first problem that developers encounter when starting on development with multiple languages is *language cacophony*. Learning the different languages and understanding their interaction is necessary to make sense of the complex composition of artifacts. The OFBiz framework for instance has seventeen different languages and more than 200 000 lines of domain-specific language code so the complexity can be quite overwhelming! There is currently no established method of characterizing different languages such that developers quickly can reach an operational understanding. Tools are important here as an *ad hoc* mechanism for learning and exploration because developers typically use tools to learn by experiments. There are especially three areas where tools for domain-specific models are helpful:

1. Understanding a language
2. Understanding language interactions
3. Understanding how to use languages

First, understanding a language can be difficult and in the case of XML-based domain-specific languages a frequent and intuitive objection is the *syntax matters* objection. This argument can be stated in the following way: “The different languages are hard to understand and only add to the confusion because their XML-based syntax is particularly verbose and unintelligible. Using a single general-purpose language like Java would be better because then developers could rely on a syntax that they already know”. While this objection is certainly important, it misses a central point. XML or a similar representation format may not be the syntax that developers actually work with. One of the advantages

of using XML-based domain-specific languages is that we can then provide domain-specific editors. The figure below shows what a hypothetical editor for the Entity DSL might look like. This editor presents the domain in a simple and visually appealing manner but may very well use the XML representation (and perhaps a layout configuration) underneath.



Just as we may complain that XML is a bad choice, we could also object that a general-purpose language like Java is a poor choice for some tasks. Furthermore, developers may feel less intimidated by the editor in figure than by code Listings in XML or Java. If we accept that *syntax matters* then the use of different languages with tailored editors becomes a reasonable strategy. The simplicity of the editor makes the language easier to understand and hence easier to use. In other words, the *syntax matters* objection may be the very reason why we explore the field of Domain-specific languages.

Second, language interactions reveal relations between languages. Developers should be able to jump between related elements in different artifacts. Ease of navigation between different software artifacts is an important criterion for tools in traditional development environments. Although we have performed no empirical studies in this area, we hypothesize that proper navigation facilities increase productivity. This claim is supported by the observation that all major development environments today offer quite sophisticated navigation facilities such as type hierarchy browser or the ability to quickly locate and jump to references to a method definition. The development environments can provide these navigation facilities because they maintain a continuously updated model of the sourcefiles in the form of an abstract syntax tree.

In a development environment with multiple languages, navigation is much more difficult. Existing environments are not geared to parsing and representing DSL models as abstract syntax trees for arbitrary and perhaps even application-specific languages such as the languages from the previous example. Furthermore without this internal representation, existing environments cannot resolve neither intra- nor inter-language references for such languages and hence cannot provide useful navigation. This means that developers must maintain a conceptual model of how the parts of their system are related. New tools with navigation facilities geared to multiple languages would on the

other hand be very helpful in understanding the relations between languages. In terms of the *create survey* example such tools should display the relations between the three languages by using the soft references as navigation points.

Third, to understand language use we must be able to distinguish correct editing operations from wrong ones in our development environment. Traditional development environments have long provided guidance during the writing of a program. Incremental compilation allows the environment to offer detailed suggestions to the developer such as how to complete a statement. More intrusive kinds of guidance also exist such as syntax-oriented editors where only input conforming to the grammar can be entered. Generic text-editors that can be parameterized with the grammar of a language have existed for a long time.

Existing editors do not take inter-language consistency relations into account when providing guidance. In the previous example, an ideal editor should for instance be able to suggest the createSurvey service as a valid value when the developer edits the target attribute in the Form definition. An environment which could reason about artifacts from different languages would also be able to help the developer identify program states where there was local but not global consistency. Such a situation can arise when a model is well-formed and hence locally consistent but at the same time violates an inter-language constraint. Guidance or intelligent assistance in the form of proposals on how to complete a model would be useful for setups with multiple languages and complex consistency constraints. Tool-suggested editing operations could make it easier for the developer to get started on the process of learning how to use the languages.

Coordination as a technical challenge

The technical aspect of the coordination problem is essentially a matter of enforcing consistency. How can we detect inconsistencies across models from multiple languages at modeling time? To fully understand the complexity of the consistency requirements of a system based on multiple languages, it is useful to refine our concept of consistency.

Consistency can be either intra- or inter-consistency. Intra-consistency concerns the consistency of elements within a single model. The requirements here are that the model must conform to its metamodel, i.e., be syntactically well-formed. In terms of the create survey example, the entity model must for instance conform to the XSD schema of the Entity DSL. This schema is the metamodel of the Entity DSL and it specifies how elements can be composed and what are, to some extent, the valid domains of attributes.

Inter-consistency is achieved when references across language boundaries can be resolved. This kind of consistency can be further subdivided into (1) model-to-model consistency and (2) model-to-code consistency. Model-to-model consistency concerns the referential integrity as well as high-level constraints of the system. In the *create survey* example, the default-entity-name attribute from the Service listing refers to the name attribute from Entity listing. If we change one of these values without updating the other, we break the reference. More high-level consistency constraints across different

models also exist as discussed later. A project can have certain patterns or conventions for naming and relating model elements. Current development environments must be tailored to specific languages with handwritten plugins or similar mechanisms in order to enforce consistency between languages such as those from the previous example.

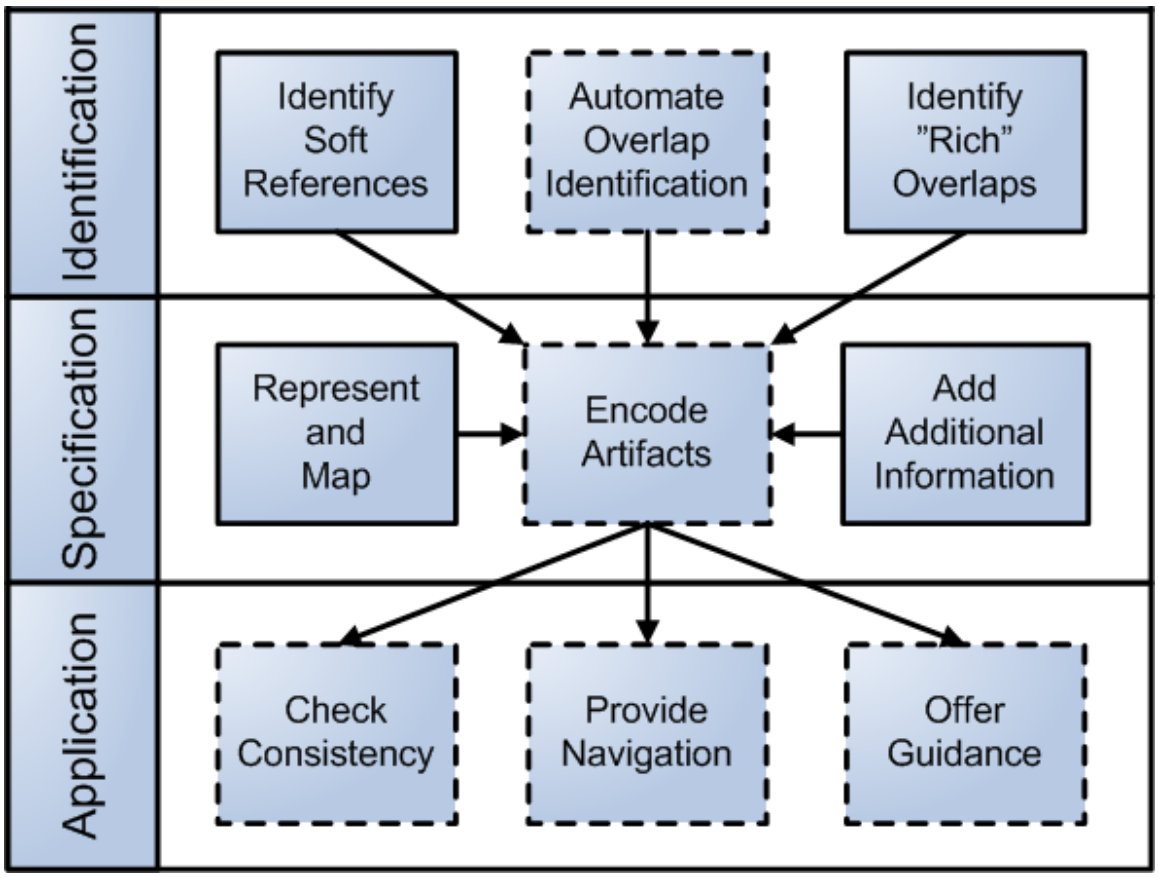
Model-to-code consistency is an essential requirement in multi-level customization. When models are supplemented with code snippets as in the *build PDF* example, it is necessary to check that models and code actually *fit*. This partly a matter of making sure that soft references between models and code are not broken, similar to referential integrity in model-to-model consistency. But it is also a matter of making sure that the code does not violate expectations set up in the model. In the *build PDF* example, the model specifies that outByteWrapper will always be part of the output, i.e., the outByteWrapper key is put in the results map. An analysis of the code shows that outByteWrapper will only be part of the output if no exceptions are thrown before line 10. In other words, some possible executions of the code will violate a specification on the modeling level. More generally, we can state that multi-level customization imposes very fine-grained constraints on the involved models and code snippets.

Solving the coordination problem

The coordination problem arises from the fact that multiple languages are used in a single system. The two previous Subsections illustrate that this problem has both a conceptual side as well as a low-level technical side. The challenges that we have described are real rather than hypothetical challenges. Specifically, we have faced these challenges in two concrete and representative case studies: an enterprise resource planning system, OFBiz, and a health care system, the District Health Information System (DHIS). Both cases are medium-sized systems that are in actual industrial use. Our solution to the practical problems we have encountered during our work with these systems are a set of guidelines and prototypes. In the following, we will introduce an overall conceptual framework which incorporates the guidelines and prototypes into a coherent method: the *coordination method*.

Coordination method

The goal of the coordination method is to solve the coordination problem and thereby provide better support for development with multiple languages. To properly appreciate the method, it is important to understand that it does not prescribe the design of individual languages. Plenty of methods and tools have already been proposed for this. This method assumes the existence of a setup with multiple domain-specific languages. Given such a setup, one can apply the method. The method consists of three steps as shown in the diagram below. Each step consist of a couple of parts which are shown as little boxes in the diagram. Boxes with dotted lines represent automatic processes and boxes with solid lines represent manual ones. In the following, we will explain these steps in a bit more detail.



Step 1: identification

The goal of the identification step is to identify language overlaps. As described in the example, an overlap is an area where the concerns of two languages intersect. The *soft references* from Form DSL to Service DSL and from Service DSL to Entity DSL in the create survey use case are examples of such overlaps. Another example is the case where a customized code snippet is used to extend a model. Such overlaps are frequent when the expressiveness of general-purpose languages is needed to implement specialized requirements that are beyond the scope of the model. The identification step can either be a manual or an automatic process depending on the complexity of the overlaps. When the overlaps have been identified and made explicit, this information is used as input to the second step in the method: the specification step.

Step 2: specification

The goal of the specification step is to create a *coordination model* which specifies how languages interact. The references across language boundaries in a system constitute the coordination model for that particular system. It is created by mapping the main software artifacts into a common representation. Additional information such as domain- or application-specific constraints may also be encoded to provide a rich representation. The coordination model is based on generic information such as language grammars and constraints as well as application-specific information such as concrete models and

application-specific constraints. This means that even though the same languages are used across several products, each product has a specification of its own unique coordination model. The coordination model is used as basis for various forms of reasoning in the final step of the method: the application step.

Step 3: application

The goal of the application step is to take advantage of the coordination model. The coordination model allows tools to derive three layers of useful information. First, the coordination model can be used to enforce consistency across multiple languages. The coordination model specifies consistency relations such as how elements from different languages can refer to each other. Tools can enforce referential integrity and perform static checks of the final system before deployment. Second, the consistency relations are used to navigate, visualize and map the web of different languages in a development setup. This information is used to quickly link and relate elements from different languages and to provide traceability among different models. Third, based on consistency relations and navigational information about how elements are related, tools can provide guidance, specifically completion or assistance. Model completion can for instance be provided in a generic manner across domain-specific tools.

Evaluation of the coordination method

The coordination method can best be seen as a conceptual framework that prescribes a certain workflow when working with multiple languages. The three successive steps that constitute this workflow are not supported by an integrated workbench or development environment. The focus is rather on extending the developer's existing environments to add support for (1) identification, (2) specification, and (3) application. The main advantage of this approach has been that developers have actually tested our work and given us feedback. This kind of evaluation of the method is valuable because it reduces the risk of solving a purely hypothetical problem. Several papers introduce the different steps of the coordination method, report on this evaluation, and elaborates on the technical aspects of each individual experiment. Overall, the results have been promising: a significant number of errors have been found in production systems and given rise to a constructive dialog with developers on future tool requirements. A development process based on these guidelines and supported by tools constitutes a serious attempt to solve the coordination problem and make domain-specific multimodeling a practical proposition.

Chapter 8

Domain-Specific Language

In software development and domain engineering, a **domain-specific language** (**DSL**) is a programming language or specification language dedicated to a particular problem domain, a particular problem representation technique, and/or a particular solution technique. The concept isn't new—*special-purpose programming languages* and all kinds of modeling/specification languages have always existed, but the term has become more popular due to the rise of domain-specific modeling.

Examples of domain-specific languages include Logo for children, Verilog and VHSIC hardware description languages, R and S languages for statistics, Mata for matrix programming, Mathematica and Maxima for symbolic mathematics, spreadsheet formulas and macros, SQL for relational database queries, YACC grammars for creating parsers, regular expressions for specifying lexers, the Generic Eclipse Modeling System for creating diagramming languages, Csound for sound and music synthesis, and the input languages of GraphViz and GrGen, software packages used for graph layout and graph rewriting.

The opposite is:

- a *general-purpose programming language*, such as C or Java,
- or a *general-purpose modeling language* such as the Unified Modeling Language (UML).

Creating a domain-specific language (with software to support it) can be worthwhile if the language allows a particular type of problems or solutions to them to be expressed more clearly than pre-existing languages would allow, and the type of problem in question reappears sufficiently often. Language-Oriented Programming considers the creation of special-purpose languages for expressing problems a standard part of the problem solving process.

Overview

A domain-specific language is created specifically to solve problems in a particular domain and is not intended to be able to solve problems outside it (although that may be

technically possible). In contrast, general-purpose languages are created to solve problems in many domains. The domain can also be a business area. Some examples of business areas include:

- domain-specific language for life insurance policies developed internally in large insurance enterprise
- domain-specific language for combat simulation
- domain-specific language for salary calculation
- domain-specific language for billing

A domain-specific language is somewhere between a tiny programming language and a scripting language, and is often used in a way analogous to a programming library. The boundaries between these concepts are quite blurry, much like the boundary between scripting languages and general-purpose languages.

In design and implementation

Domain-specific languages are languages (or most often, declared syntaxes or grammars) with very specific goals in design and implementation. A domain-specific language can be either a visual diagramming language, such as those created by the Generic Eclipse Modeling System, programmatic abstractions, such as the Eclipse Modeling Framework, or textual languages. For instance, the command line utility grep has a regular expression syntax which matches patterns in lines of text. The sed utility defines a syntax for matching and replacing regular expressions. Often, these tiny languages can be used together inside a shell to perform more complex programming tasks.

The line between domain-specific languages and scripting languages is somewhat blurred, but domain-specific languages often lack low-level functions for filesystem access, interprocess control, and other functions that characterize full-featured programming languages, scripting or otherwise. Many domain-specific languages do not compile to byte-code or executable code, but to various kinds of media objects: GraphViz exports to PostScript, GIF, JPEG, etc., where Csound compiles to audio files, and a ray-tracing domain-specific language like POV compiles to graphics files. A computer language like SQL presents an interesting case: it can be deemed a domain-specific language because it is specific to a specific domain (in SQL's case, accessing and managing relational databases), and is often called from another application, but SQL has more keywords and functions than many scripting languages, and is often thought of as a language in its own right, perhaps because of the prevalence of database manipulation in programming and the amount of mastery required to be an expert in the language.

Further blurring this line, many domain-specific languages have exposed APIs, and can be accessed from other programming languages without breaking the flow of execution or calling a separate process, and can thus operate as programming libraries.

Programming tools

Some domain-specific languages expand over time to include full-featured programming tools, which further complicates the question of whether a language is domain-specific or not. A good example is the functional language XSLT, specifically designed for transforming one XML graph into another, which has been extended since its inception to allow (particularly in its 2.0 version) for various forms of filesystem interaction, string and date manipulation, and data typing.

In model-driven engineering many examples of domain-specific languages may be found like OCL, a language for decorating models with assertions or QVT, a domain specific transformation language. However languages like UML are typically general purpose modeling languages.

To summarize, an analogy might be useful: a Very Little Language is like a knife, which can be used in thousands of different ways, from cutting food to cutting down trees. A domain-specific language is like an electric drill: it is a powerful tool with a wide variety of uses, but a specific context, namely, putting holes in things. A General Purpose Language is a complete workbench, with a variety of tools intended for performing a variety of tasks. Domain-specific languages should be used by programmers who, looking at their current workbench, realize they need a better drill, and find that a specific domain-specific language provides exactly that.

Domain-specific language topics

Usage patterns

There are several usage patterns for domain-specific languages:

- processing with standalone tools, invoked via direct user operation, often on the command line or from a Makefile (e.g., the GraphViz tool set)
- domain-specific languages which are implemented using programming language macro systems, and which are converted or expanded into a host general purpose language at compile-time or read-time
- **embedded** (or **internal**) **domain-specific languages**, implemented as libraries which exploit the syntax of their host general purpose language or a subset thereof, while adding domain-specific language elements (data types, routines, methods, macros etc.). The distinction between an embedded DSL and a generic library or API is fuzzy and mostly relates to stylistic and pragmatic concerns: most APIs are designed to expose their features in a straightforward and transparent way, rather than create a usable and distinctive 'language'.
- domain-specific languages which are called (at runtime) from programs written in general purpose languages like C or Perl, to perform a specific function, often returning the results of operation to the "host" programming language for further processing; generally, an interpreter or virtual machine for the domain-specific language is embedded into the host application

- domain-specific languages which are embedded into user applications (e.g., macro languages within spreadsheets) and which are (1) used to execute code that is written by users of the application, (2) dynamically generated by the application, or (3) both.

Many domain-specific languages can be used in more than one way.

Design goals

Adopting a domain-specific language approach to software engineering involves both risks and opportunities. The well-designed domain-specific language manages to find the proper balance between these.

Domain-specific languages have important design goals that contrast with those of general-purpose languages:

- domain-specific languages are less comprehensive.
- domain-specific languages are much more expressive in their domain.
- domain-specific languages should exhibit minimum redundancy according to the following subjective definition.

Redundancy of a program is defined as the average number of textual insertions, deletions, or replacements necessary to correctly implement a single stand-alone change in requirements. For a language, this is averaged over programs in the problem domain. This measure is useful because, the smaller it is, the less likely that bugs can be introduced by incompletely implementing changes.

Idioms

In programming, idioms are methods imposed by programmers to handle common development tasks, e.g.:

- Ensure data is saved before the window is closed.
- Before conducting expensive tests, perform cheap tests that can rule out need for expensive tests.
- Edit code whenever command-line parameters change because they affect program behavior.

General purpose programming languages rarely support such idioms, but domain-specific languages can describe them, e.g.:

- A script can automatically save data.
- A smart test harness can learn what good tests are.
- A domain-specific language can parameterize command line input.

Examples

Unix shell scripts

Unix shell scripts give a good example of a domain-specific language for data organization. They can manipulate data in files or user input in many different ways. Domain abstractions and notations include streams (such as stdin and stdout) and operations on streams (such as redirection and pipe). These abstractions combine to make a robust language to talk about the flow and organization of data.

The language consists of a simple interface (a script) for running and controlling processes that perform small tasks. These tasks represent the idioms of organizing data into a desired format such as tables, graphs, charts, etc.

These tasks consist of simple control-flow and string manipulation mechanisms that cover a lot of common usages like searching and replacing string in files, or counting occurrences of strings (frequency counting).

Even though Unix scripting languages are Turing complete, they differ from general purpose languages.

In practice, scripting languages are used to weave together small Unix tools such as AWK (e.g., gawk), ls, sort or wc.

ColdFusion Markup Language

ColdFusion's associated scripting language is another example of a domain-specific language for data-driven websites. This scripting language is used to weave together languages and services such as Java, .NET, C++, SMS, email, email servers, http, ftp, exchange, directory services, and file systems for use in websites.

The ColdFusion Markup Language includes a set of tags that can be used in ColdFusion pages to interact with data sources, manipulate data, and display output. CFML tag syntax is similar to HTML element syntax.

Erlang OTP

The Erlang Open Telecom Platform was originally designed for use inside Ericsson as a domain specific language. The language itself offers a platform of libraries to create finite state machines, generic servers and event managers that quickly allow an engineer to deploy applications, or support libraries, that have been shown in industry benchmarks to outperform other languages intended for a mixed set of domains, such as C and C++. The language is now officially open source and can be downloaded from their website.

FilterMeister

FilterMeister is a programming environment, with a programming language that is based on C, for the specific purpose of creating Photoshop-compatible image processing filter plug-ins; FilterMeister runs as a Photoshop plug-in itself and it can load and execute scripts or compile and export them as independent plug-ins. Although the FilterMeister language reproduces a significant portion of the C language and function library, it contains only those features which can be used within the context of Photoshop plug-ins and adds a number of specific features only useful in this specific domain.

Software engineering uses

There has been much interest in domain-specific languages to improve the productivity and quality of software engineering. Domain-specific language could possibly provide a robust set of tools for efficient software engineering. Such tools are beginning to make their way into development of critical software systems.

The Software Cost Reduction Toolkit is an example of this. The toolkit is a suite of utilities including a specification editor to create a requirements specification, a dependency graph browser to display variable dependencies, a consistency checker to catch missing cases in well-formed formulas in the specification, a model checker and a theorem prover to check program properties against the specification, and an invariant generator that automatically constructs invariants based on the requirements.

A newer development is Language-oriented programming, an integrated software engineering methodology based mainly on creating, optimizing, and using domain-specific languages.

Metacompilers

Complementing language-oriented programming, as well as all other forms of domain-specific languages, are the class of compiler writing tools called metacompilers. A metacompiler is not only useful for generating parsers and code generators for domain specific languages, but a metacompiler is also itself a domain-specific language for the domain of compiler writing. The feature that sets a metacompiler apart from a standard compiler-compiler is that a metacompiler is written in its own language and translates itself—the grammar productions defining itself written in its own specialized language—into the executable form of itself. Defining itself and translating itself constitute the *meta*-step that sets a metacompiler apart from other compiler-compilers.

Besides parsing domain-specific languages, metacompilers are useful for generating a wide range of software engineering and analysis tools.

Metacompilers that played a significant role in both computer science and the computer industry include Meta-II and its descendent TreeMeta.

Unreal Engine and other games

Unreal and Unreal Tournament unveiled a language called UnrealScript. This allowed for rapid development of modifications compared to the competitor Quake (using the Id Tech engine). The Id Tech engine uses standard C code meaning C had to be learned and properly applied, while UnrealScript was optimized for ease of use and efficiency. Similarly, the development of more recent games introduced their own specific languages, one more common example is Lua for scripting.

Rules Engines for Policy Automation

Various Business Rules Engines have been developed for automating policy and business rules used in both government and private industry. ILOG, Oracle Policy Automation, DTRules, Drools and others provide support for DSLs aimed to support various problem domains. DTRules goes so far as to define an interface for the use of multiple DSLs within a Rule Set.

The purpose of Business Rules Engines is to define a representation of business logic in as human readable fashion as possible. This allows both subject matter experts and developers to work with and understand the same representation of the business logic. Most Rules Engines provide both an approach to simplifying the control structures for business logic (for example, using Declarative Rules or Decision Tables) coupled with alternatives to programming syntax in favor of DSLs.

Advantages and disadvantages

Some of the advantages:

- Domain-specific languages allow solutions to be expressed in the idiom and at the level of abstraction of the problem domain. Consequently, domain experts themselves can understand, validate, modify, and often even develop domain-specific language programs.
- Self-documenting code.
- Domain-specific languages enhance quality, productivity, reliability, maintainability, portability and reusability.
- Domain-specific languages allow validation at the domain level. As long as the language constructs are safe any sentence written with them can be considered safe.

Some of the disadvantages:

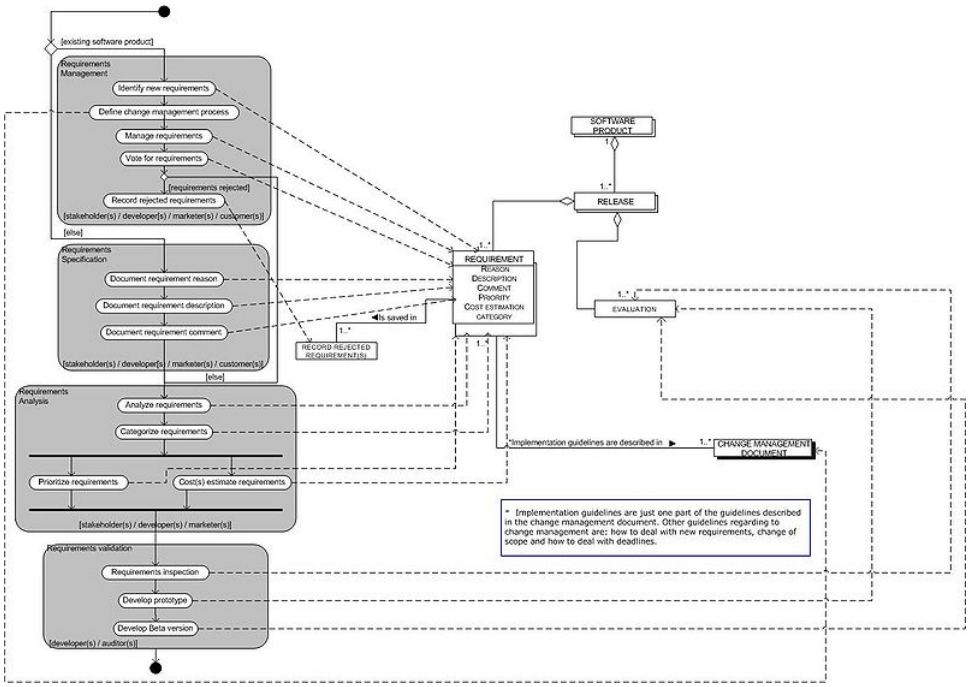
- Cost of learning a new language vs. its limited applicability
- Cost of designing, implementing, and maintaining a domain-specific language as well as the tools required to develop with it (IDE)
- Finding, setting, and maintaining proper scope.

- Difficulty of balancing trade-offs between domain-specificity and general-purpose programming language constructs.
- Potential loss of processor efficiency compared with hand-coded software.
- Proliferation of similar non-standard domain specific languages, i.e. a DSL used within insurance company A versus a DSL used within insurance company B.
- Non-technical domain experts can find it hard to write or modify DSL programs by themselves.

Chapter 9

Process-Data Diagram and Meta-Object Facility

Process-data diagram

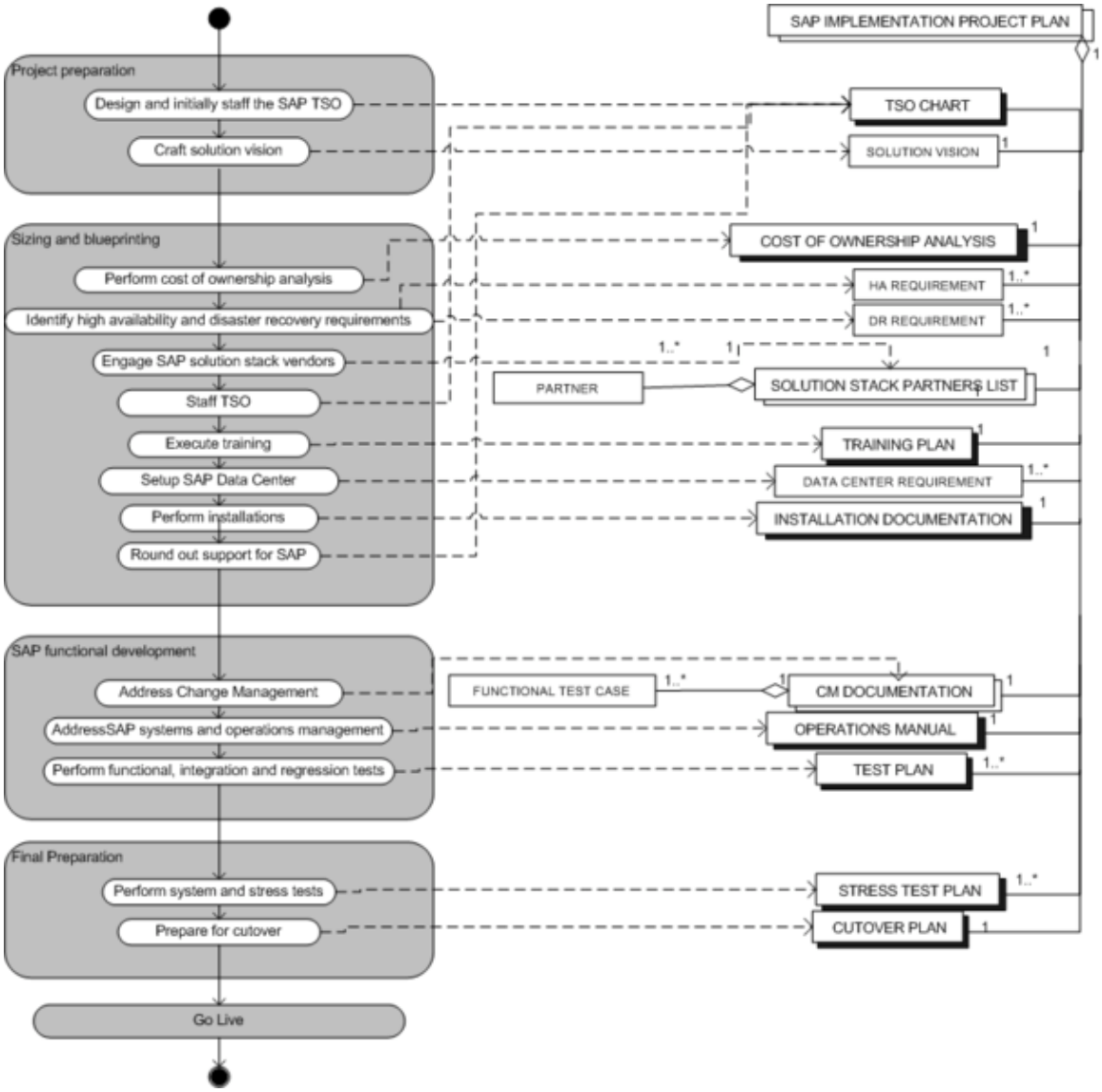


The process data diagram

A **process-data diagram** is a diagram that describes processes and data that act as output of these processes. On the left side the meta-process model can be viewed and on the right side the meta concept model can be viewed.

A process-data diagram can be seen as combination of an business process model and data model.

Overview



SAP Implementation process-data diagram

The process-data diagram that is depicted at the right, gives an overview of all of these activities/processes and deliverables. The four gray boxes depict the four main implementation phases, which each contain several processes that are in this case all sequential. The boxes at the right show all the deliverables/concepts that result from the processes. Boxes without a shadow have no further sub-concepts. Boxes with a black shadow depict complex closed concepts, so concepts that have sub-concepts, which however will not be described in any more detail. Boxes with a white shadow (a box behind it) depict open closed concepts, where the sub-concepts are expanded in greater detail. The lines with diamonds show a has-a relationship between concepts.

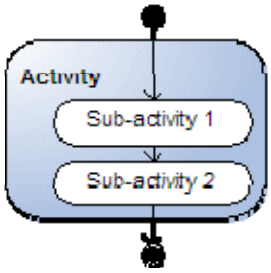
The SAP Implementation process is made up out of four main phases, i.e. the project preparation where a vision of the future-state of the SAP solution is being created, a sizing and blueprinting phase where the solution stack is created and training is being performed, a functional development phase and finally a final preparation phase, when the last tests are being performed before the actual go live. For each phase, the vital activities are addressed and the deliverables/products are explained.

Process-data diagram building blocks

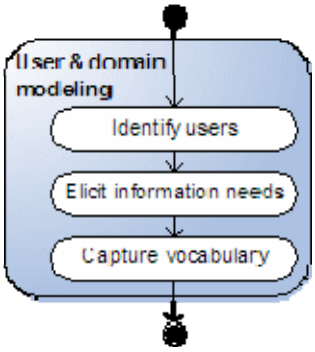
Sequential activities

Sequential activities are activities that need to be carried out in a pre-defined order. The activities are connected with an arrow, implying that they have to be followed in that sequence. Both activities and sub-activities can be modeled in a sequential way. In Figure 1 an activity diagram is illustrated with one activity and two sequential sub-activities. A special kind of sequential activities are the start and stop states, which are also illustrated in Figure 1.

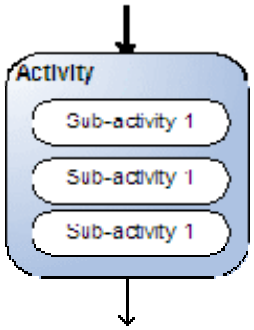
In Figure 2 an example from practice is illustrated. The example is taken from the requirements capturing workflow in UML-based Web Engineering. The main activity, user & domain modeling, consists of three activities that need to be carried out in a predefined order.



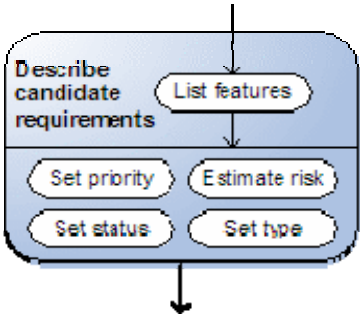
1: Sequential activities



2: Example



3: Unordered activities



4: Example

Unordered activities

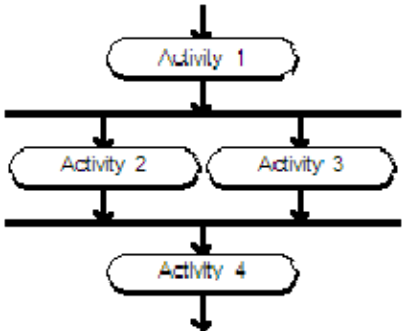
Unordered activities are used when sub-activities of an activity do not have a pre-defined sequence in which they need to be carried out. Only sub-activities can be unordered. Unordered activities are represented as sub-activities without transitions within an activity, as is represented in Figure 3.

Sometimes an activity consists of both sequential and unordered sub-activities. The solution to this modeling issue is to divide the main activity in different parts. In Figure 4 an example is illustrated, which clarifies the necessity to be able to model unordered activities. The example is taken from the requirements analysis workflow of the Unified Process. The main activity, “describe candidate requirements”, is divided into two parts. The first part is a sequential activity. The second part consists of four activities that do not need any sequence in order to be carried out correctly.

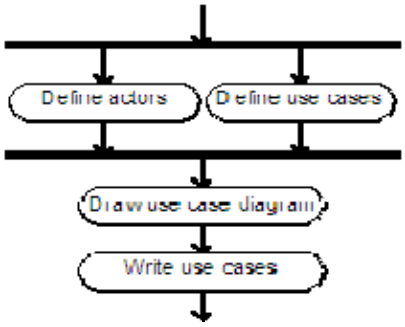
Concurrent activities

Activities can occur concurrently. This is handled with forking and joining. By drawing the activities parallel in the diagram, connected with a synchronization bar, one can fork several activities. Later on these concurrent activities can join again by using the same synchronization bar. Both activities and sub-activities can occur concurrently. In the example of Figure 5, Activity 2 and Activity 3 are concurrent activities.

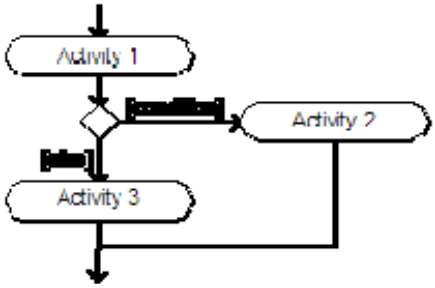
In Figure 6, a fragment of a requirements capturing process is depicted. Two activities, defining the actors and defining the use cases, are carried out concurrently. The reason for carrying out these activities concurrently is that defining the actors influences the use cases greatly, and vice versa.



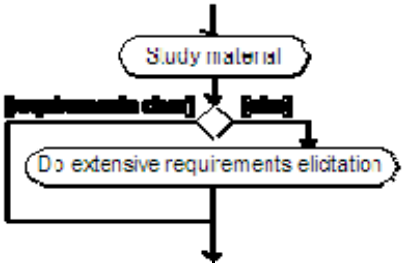
5: Concurrent activities



6: Example



7: Conditional activities



8: Example

Conditional activities

Conditional activities are activities that are only carried out if a pre-defined condition is met. This is graphically represented by using a branch. Branches are illustrated with a diamond and can have incoming and outgoing transitions. Every outgoing transition has a guard expression, the condition. This guard expression is actually a Boolean expression, used to make a choice which direction to go. Both activities and sub-activities can be modeled as conditional activities. In Figure 7 two conditional activities are illustrated.

In Figure 8 an example from practice is illustrated. A requirements analysis starts with studying the material. Based on this study, the decision is taken whether to do an extensive requirements elicitation session or not. The condition for not carrying out this requirements session is represented at the left of the branch, namely [requirements clear]. If this condition is not met, [else], the other arrow is followed.

Process-data diagram

The integration of both types of diagrams is quite straightforward. Each action or activity results in a concept. They are connected with a dotted arrow to the produced artifacts, as is demonstrated in Figure 9. The concepts and activities are abstract in this picture.

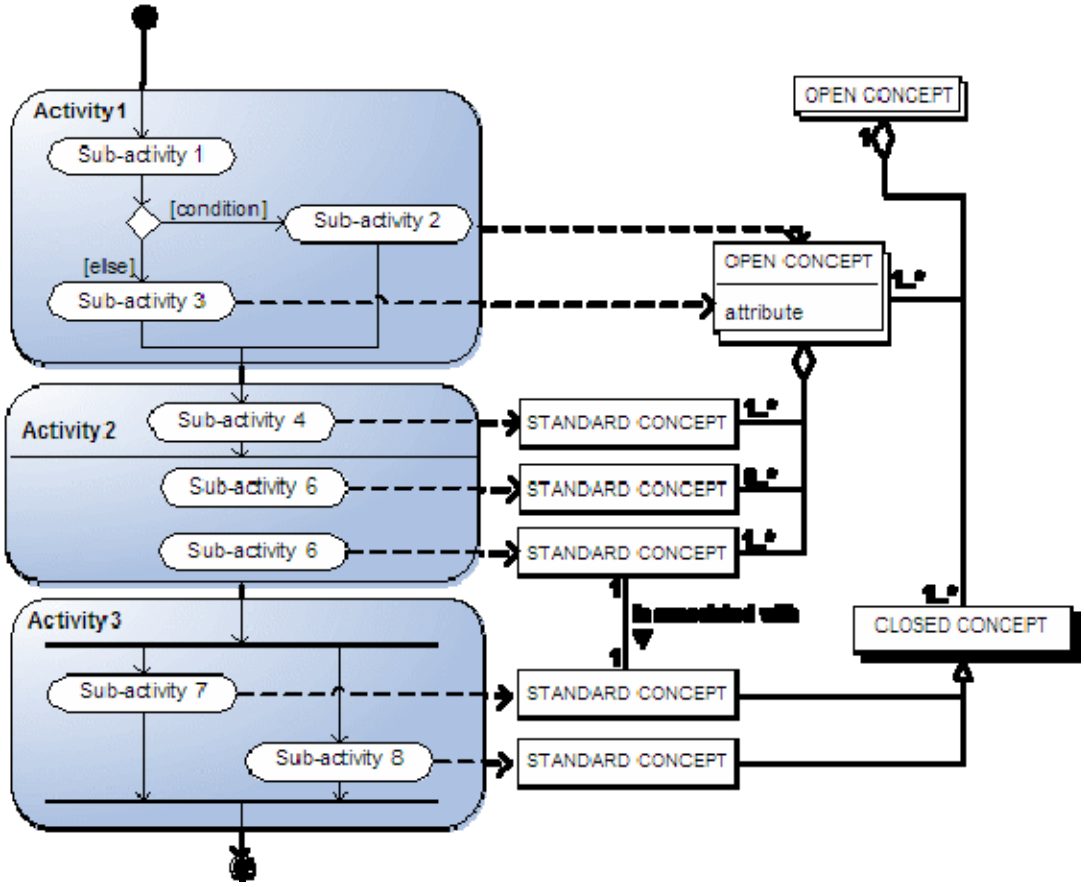


Figure 9: Process-Data Diagram

In Table 1 a generic table is presented with the description of activities, sub-activities and their relations to the concepts. In section 5 examples are given of both process-data diagram and activity table.

Activity	Sub-Activity	Description
Activity 2	Sub-activity 4	Sub-activity 4 results in a STANDARD CONCEPT

Table 1: Activity table

Example of a process-data diagram

In Figure 10 an example of a process-data diagram is illustrated. It concerns an example from a the orientation phase of complex project in a WebEngineering method.

Notable is the use of open and closed concepts. Since project management is actually not within the scope of this research, the concept CONTROL MANAGEMENT has not been expanded. However, in a complex project is RISK MANAGEMENT of great importance. Therefore, the choice is made to expand the RISK MANAGEMENT concept.

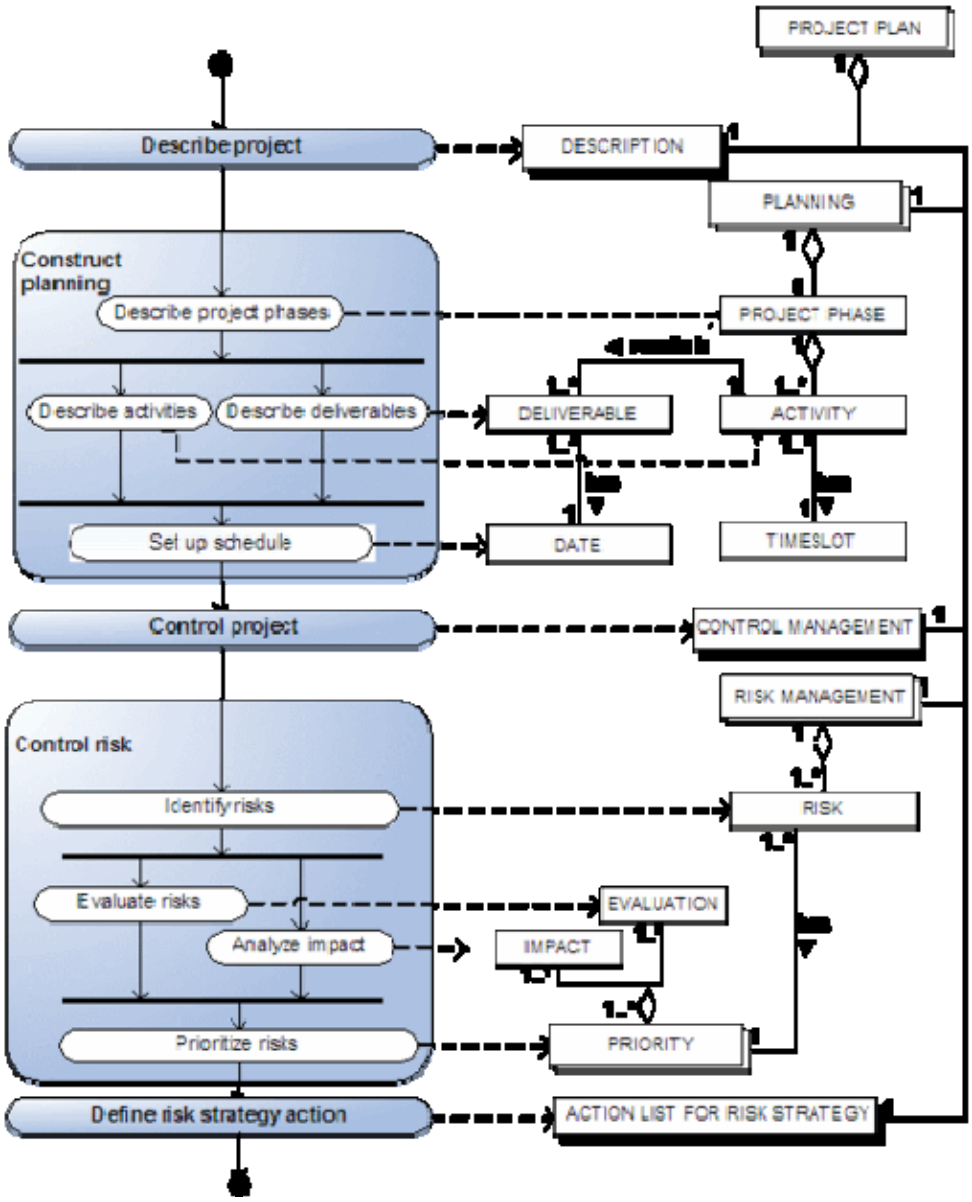


Figure 10: Example Process-Data Diagram - Orientation phase in a complex project

In Table 2 the activities and sub-activities, and relation to the concepts are described.

Activity	Sub-Activity	Description
Describe Project		Describing the project is done in terms of participants, targets, products, scope and assumptions. This information is derived from the proposal, but with more emphasis on the project management issues. The activity end in a project DESCRIPTION.
Construct	Describe	The PLANNING is divided into five PROJECT

planning	project phases	PHASES, which should be shortly described.
Construct planning	Describe activities	The project ACTIVITIES are described and grouped into PROJECT PHASES.
Construct planning	Describe deliverables	The DELIVERABLES that result from the project ACTIVITIES are described.
Construct planning	Set up schedule	For every DELIVERABLE a DATE is set and for each ACTIVITY a TIME SLOT is estimated.
Control project		Controlling the project results in a CONTROL MANAGEMENT artifact. This artifact is not further explained here, since it concerns regular project management issues, like communication management, progress management, change management and problem management that lie outside the scope of this research.
Control risk	Identify risks	Identifying risks can be done by using standard checklists or organizing risk workshops. The RISKS are included in the PROJECT PLAN.
Control risk	Evaluate risks	Every RISK is provided with an EVALUATION; a description and estimation about the complexity or uncertainty of a project is given.
Control risk	Analyze impact	Analyzing the impact of a risk handles about the IMPACT, a risk has on the success of a project. The evaluation and risk values are indicated by selecting a value: low, moderate or high.
Control risk	Prioritize risks	Prioritizing the risks is done by combining IMPACT and EVALUATION in a table. High priority is then given to RISKS with the highest scores.
Define actions for risk strategy		Risk strategy actions can be obtained from experience or from relevant literature. The project manager adapts the actions to the project in the ACTION LIST FOR RISK STRATEGY. RISKS with the highest priority are on top of the list and need to be handled first.

Table 2: Activities and sub-activities in a complex orientation phase

Meta-Object Facility

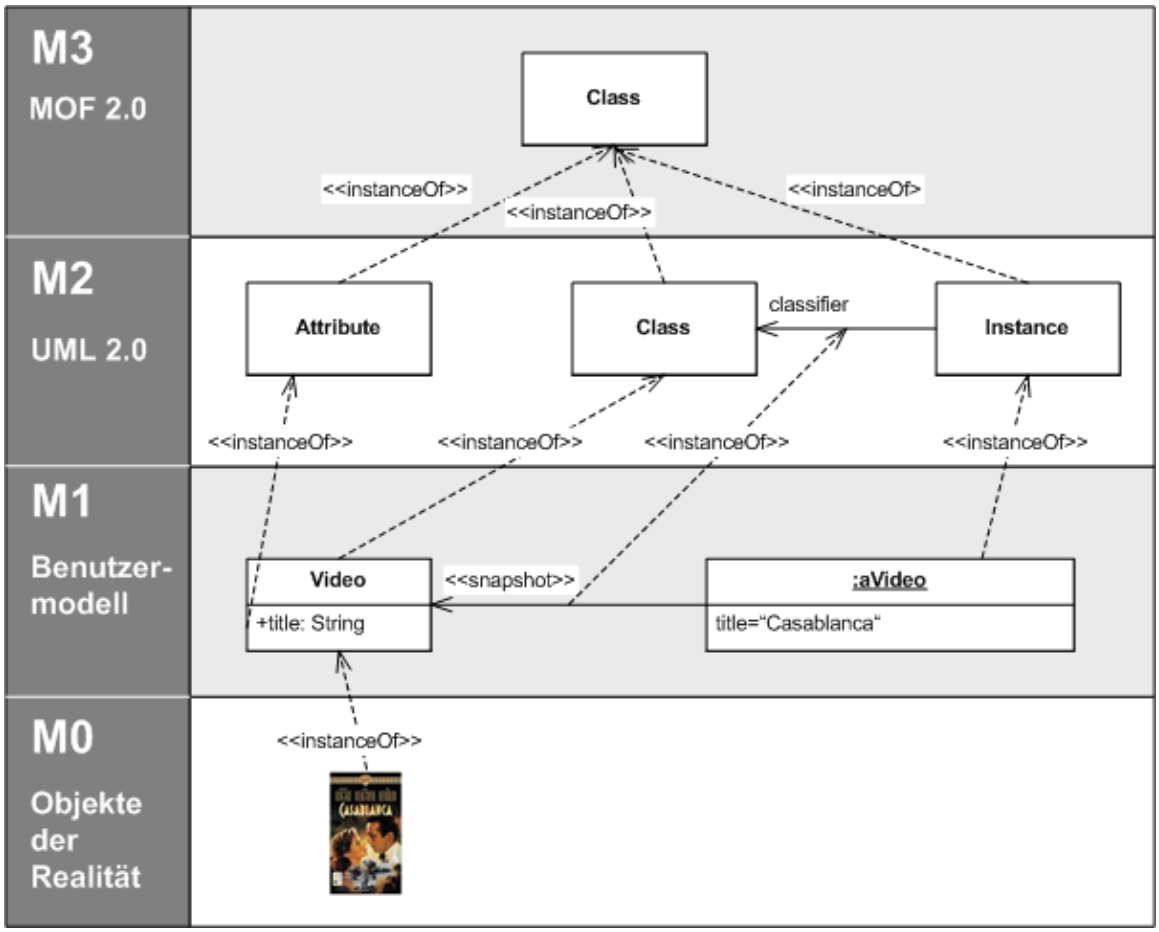


Illustration of the Meta-Object Facility.

The **Meta-Object Facility** (**MOF**) is an Object Management Group (OMG) standard for model-driven engineering. The official reference page may be found at OMG's website.

Overview

MOF originated in the Unified Modeling Language (UML); the OMG was in need of a metamodeling architecture to define the UML. MOF is designed as a four-layered architecture. It provides a meta-meta model at the top layer, called the M3 layer. This M3-model is the language used by MOF to build metamodels, called M2-models. The most prominent example of a Layer 2 MOF model is the UML metamodel, the model that describes the UML itself. These M2-models describe elements of the M1-layer, and thus M1-models. These would be, for example, models written in UML. The last layer is the M0-layer or data layer. It is used to describe real-world objects.

Beyond the M3-model, MOF describes the means to create and manipulate models and metamodels by defining CORBA interfaces that describe those operations. Because of the similarities between the MOF M3-model and UML structure models, MOF metamodels are usually modeled as UML class diagrams. A supporting standard of MOF is XMI, which defines an XML-based exchange format for models on the M3-, M2-, or M1-Layer.

Metamodeling architecture

MOF is a *closed* metamodeling architecture; it defines an M3-model, which conforms to itself. MOF allows a *strict* meta-modeling architecture; every model element on every layer is strictly in correspondence with a model element of the layer above. MOF only provides a means to define the structure, or abstract syntax of a language or of data. For defining metamodels, MOF plays exactly the role that EBNF plays for defining programming language grammars. MOF is a Domain Specific Language (DSL) used to define metamodels, just as EBNF is a DSL for defining grammars. Similarly to EBNF, MOF could be defined in MOF.

In short MOF uses the notion of **MOF::Classes** (not to be confused with **UML::Classes**), as known from object orientation, to define concepts (model elements) on a metalayer. MOF may be used to define object-oriented metamodels (as UML for example) as well as non object-oriented metamodels (as a Petri net or a Web Service metamodel).

As of May 2006, the OMG has defined two variants of MOF:

- EMOF for Essential MOF
- CMOF for Complete MOF

In June 2006, a *request for proposal* was issued by OMG for a third variant, SMOF (Semantic MOF).

The variant **ECore** that has been defined in the **Eclipse Modeling Framework** is more or less aligned on OMG's EMOF.

Another related standard is OCL, which describes a formal language that can be used to define model constraints in terms of predicate logic.

A very important new standard is QVT which introduces means to query, view and transform MOF-based models.

International standard

MOF is an international standard:

ISO/IEC 19502:2005 Information technology -- Meta Object Facility (MOF)

MOF can be viewed as a standard to write metamodels, for example in order to model the abstract syntax of Domain Specific Languages. Kermeta is an extension to MOF allowing executable actions to be attached to EMOF meta-models, hence making it possible to also model a DSL operational semantics and readily obtain an interpreter for it.

JMI defines a Java API for manipulating MOF models.

OMG's MOF is not to be confused with the Managed Object Format (MOF) defined by the Distributed Management Task Force (DMTF) in section 6 of the Common Information Model (CIM) Infrastructure Specification, version 2.5.0.

Glossary of Unified Modeling Language Terms

This **glossary of Unified Modeling Language terms** covers all versions of UML. Individual entries will point out any distinctions that exist between versions.

A

- **Abstract** - An indicator applied to a classifier (e.g., actor, class, use case) or to some features of a classifier (e.g., a class's operations) showing that the feature is incomplete and is intended not to be instantiated, but to be specialized by other definitions.
- **Abstract class** - A class that does not provide a complete declaration, perhaps because it has no implementation method identified for an operation. By declaring a class as *abstract*, one intends to prohibit direct instantiation of the class. An abstract class cannot directly instantiate objects; it must be inherited from before it can be used.
- Abstract data type
- **Abstract operation** - Unlike attributes, class operations can be abstract, meaning that there is no provided implementation. Generally, a class containing an abstract operation should be marked as an abstract class. An Operation must have a method supplied in some specialized Class before it can be used.
- **Abstraction** is the process of picking out common features and deriving essential characteristics from objects and procedure entities that distinguish it from other kinds of entities.
- **Action** - An action is the fundamental unit of behaviour specification and represents some transformation or processing in the modeled system, such as invoking a method of a class or a sub activity
- **Activation** - the time during which an object has a method executing. It is often indicated by a thin box or bar superimposed on the Object's lifeline in a Sequence Diagram

- **Activity diagram** - a diagram that describes procedural logic, business process or work flow. An activity diagram contains a number of Activities and connected by Control Flows and Object Flows.
- **Active class** - a class defining active objects
- **Active object** - an object running under its own thread
- **Activity** - carrying out behaviour in a State machine diagram
    - **Do** - a type of Activity which may be interrupted, as opposed to normal Activities which may not be interrupted
    - **Internal** - an Activity that is executed within a State
        - **Entry** - an Activity that is executed when a State is entered
        - **Exit** - an Activity that is executed when a State is exited
- **Activity final** - the end point of an activity diagram. When a thread reaches an Activity Final node, all the threads of the activity terminate. (Contrast with Flow Final node, which marks the end of one thread.)
- **Aggregation** - a special type of association used to represent a stronger relationship between two classes than a regular association; typically read as "owns a", as in, "Class A owns a Class B". A hierarchy of classes where the child object may or may not continue to exist if the parent object is destroyed.
- **Artifact** - items that model physical pieces of information in your system, such as a user's manual, training material, or password file
- **Association** - a relationship with 2 or more ends, where each end is on a class (or other classifier). Each end is called a Role, and may have a role name, Multiplicity, and may be Navigable.
- **Association class** - a class that describes an association.
- **Asynchronous** - The sender of an asynchronous message does not wait for a response.
- **Attribute** - a significant piece of data owned by a Class, often containing values describing each instance of the class. Besides the attribute name and a slot for the attribute value, an attribute may have specified Visibility, Type, Multiplicity, Default value, and Property-string.

C

- **Cardinality** - the current number of occurrences of a Property. The cardinality must be a value that is allowed by the multiplicity
- **Class** - the primary declarative construct of Object-Oriented Programming; a cohesive unit of Attributes and Operations; a compile-time template for an Object
- **Class diagram** - a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes and the relationships between the classes.
- **Classifier** - a category of UML elements that have some common features, such as attributes or methods.
- **Communication diagram**
- **Component** - A component represents a software module (source code, binary code, executable, DLL, etc.) with a well-defined interface. The interface of a component is represented by one or several interface elements that the component

provides. Components are used to show compiler and run-time dependencies, as well as interface and calling dependencies among software modules. They also show which components implement a specific class.
- **Composition** - a specific type of relationship describing how one Object is *composed of* another Object; a form of Aggregation where the child object is destroyed if the parent object is destroyed.
- **Constraint** - natural language, programming language or Object Constraint Language boolean condition which may not be false if a Class is to be considered valid
- **Containment** - containment by value and containment by reference. Containment by value implies that an object contains another object; containment by reference implies that an object contains a pointer to another object.

D

- **Decision** - a point in an Activity diagram where a Flow splits into several, mutually exclusive, Guarded flows. A Merge marks the end of the optional behaviour started by the Decision
- **Dependency** - a dependency exists between two defined elements if a change to the definition of one would result in a change to the other. In UML this is indicated by a line pointing from the dependent to the independent element.
- **Deployment diagram**
- **Derived property** - a property that can be calculated or inferred from other properties
- **Diagram** - a visual representation of a subset of features of a UML Model
- **Domain** - a logical grouping that explicitly declares a rule which defines ownership of objects based upon some type or property.

E

- **Edge** - a synonym for Flow
- **Enumeration** - a set of constant values for a new data type
- **Event** - when it occurs on an Object it may cause a Transition in a State machine diagram
- **Expansion region** - a set of Actions in an Activity diagram that occur once for each of a collection of input Tokens to the Expansion Region

F

- **Final state** - the state at which an object ceases to exist
- **Flow** - a navigational connection between two Actions
- **Flow Final** - the point at which a Flow ends without ending the complete Activity
- **Fork** - a point in an Activity diagram where a Flow of logic splits into several concurrent Flows (Threads)
- **Found Message** - starting point for a Sequence diagram

G

- **Generalization** - a relationship between a *specific classifier* (typically a class) to a more *general classifier* asserting that the *general classifer* contains common features among both the *specific classifier* and the *general classifier*. Features include, for example, properties, and constraints. The use of generalization is often logically restricted to cases where the specific classifier is a "kind-of" or "sort-of" the general classifier: for example, a Boxer is a "kind-of" Dog. When the classifiers involved are software engineering classes, generalization usually involves reusing code; it is often implemented using inheritance, where the more specific code reuses the more general code.
- **Generalization Tree** - Several specialized classifiers may point to the same general classifier, forming a generalization tree, where the general classifer contains common features shared by all the specialized versions. As generalization is a relationship, it is possible for a classifer to participate in several generalizations, often being on the specific end or on the general end, forming a directed acyclic graph (DAG) (i.e., no loops).
- **Guard** - a boolean test that must be satisfied for a Flow of an activity diagram or a Transition of a state machine diagram to be allowed to start

H

- **History pseudostate** - points to the initial state of an object where no previous state history was saved

I

- **Inheritance** - where a new more specific Class derives part of its definition from an existing more general Class
- **Initial node** - the start point of an Activity diagram
- **Initial pseudostate** - points to the initial State of an Object
- **Interaction diagram**
- **Interaction overview diagram**
- **Interaction Frame** - a section of a Sequence diagram, divided into fragments, which is subject to an algorithmic Operator such as iteration, parallelism or optionality.
- **Interface** - a defined communication boundary.

J

- **Join** - a point in an activity diagram where several concurrent flows (threads) synchronize, waiting until all are complete before continuing with a single flow

L

- **Lifeline** - indicates a participating Object or Part in a Sequence diagram. The Lifeline may show activation, Object creation, and Object deletion.
- **Link** - a relationship between objects. While an Object is an instantiation of a Class a Link can be seen as an instantiation of an Association.

M

- **Merge** - a point in an Activity diagram marking the end of the optional behavior started by a Decision
- **Message** - a signal from one object (or similar entity) to another, often with parameters. Often implemented as a call to a Method, including the Constructor and Destructor, of an Object in a Sequence diagram.
- Metadata -
- Metamodel -
- Metamodeling -
- Metamodeling technique -
- Meta-Object Facility -
- **Modeling** -
    - **Domain** - the representation of real world conceptual entities
    - **Design** - the representation of software Classes and Interfaces
    - **Dynamic** - use of Interaction diagrams to describe collaborations and behavior
- Model-driven architecture (MDA) -
- **Multiplicity** - a specification of the number of possible occurrences of a property, or the number of allowable elements that may participate in a given relationship. In UML 1.x, it was also possible to have a discrete list of values, but this was eliminated in UML 2.0.
    - **Mandatory** - A required multiplicity, the lower bound is at least one, usually 1..1 or 1
    - **Optional** - The lower bound is at most zero, usually, 0..1
    - **Many** - A multiplicity with no upper limit, either 0..* or *
    - **Forbidden** - No elements allowed, 0..0 (in UML 2.2)

N

- **Namespace** - a context in which an identifier exists.
- **Navigable** - the ability for objects of a Class at one end of an Association to retrieve Objects from the other end. Associations need not be navigable.
- **Note** - It is an explanatory part of UML models. A note is a symbol for rendering constraints and comments attached to an element.

O

- **Object** - a runtime instance of a Class. Objects are rarely shown on diagrams (because there are usually too many) unless they are used to illustrate some scenario, test, etc. Such Objects are often shown with the Attributes of the Class populated with sample data
- **Object Constraint Language (OCL)** - a declarative language for describing rules and constraints that apply to UML models
- **Object diagram**
- **Operation** - the signature of a Method of a Class; consists of the Operation Name, Visibility, Parameter list, Return Type, and Property-string
- **Operator** - an algorithmic feature of Interaction Frame that defines the behavior of that frame. Examples include:
    - **alt** - multiple alternatives each with a guard condition. Only one alternative can be true.
    - **critical** - a fragment within a larger parallel Interaction Frame that when entered suspends the interleaving of events from the other fragments.
    - **loop** - the fragment iterates according to a guard condition.
    - **neg** - an invalid interaction.
    - **opt** - single alternative with a guard condition.
    - **par** - each fragment is run in parallel.
    - **ref** - an interaction defined in another diagram.
    - **strict** - a fragment with the ordering of reception events across multiple lifelines follow strictly their graphical arrangement.

P

- **Package** - A package is a collection or grouping of related classes or of classes with related functionality.
- **Parameter** - data passed in a Message to be used within the Method
- **Partition** - section of an Activity diagram or Sequence diagram occupied by a single Class or Object
- **Pin** - a parameter of an Action
- **Polymorphism** - the ability of Objects belonging to different Classes to respond to Operations of the same name, each one according to the right Class-specific behaviour
- **Profile** - Provides a generic extension mechanism for building UML models. Other extension mechanisms include stereotypes.
- **Property** - an Attribute or an Association
- **Property-string** - a qualifier for Attributes and Operations. Examples include {ordered}, {readonly}, {unique}.

Q

- **Qualified association** - tightens the multiplicity or role of an association between 2 classes by dividing the set of objects into subsets based on a value of the

qualifier—an attribute of the target objects, association objects, or some a derived attribute of the target or association objects.

R

- **Realization** - Realization shows the relationship between an Interface and the class that provides the implementation for the interface.
- **Return** - a reply that may be issued from a Method following a Message
- **Role** - description of the part played in an Association by one of the Classes in the Association

S

- **Scenario** - a narrative describing foreseeable interactions
- **Self-call** - a Message from an Object to one of its own Methods
- **Sequence diagram** - describes the Messages sent between a number of participating Objects in a Scenario
- **Signal** - an Event which can occur in an Activity diagram in three different ways: as a Time Signal, as a signal which an Activity can listen for and a signal which an Activity can send
- **State** - an Object exists at one of the States described in a State machine diagram
- **State diagram** - synonym for State machine diagram
- **State machine diagram** - describes the lifetime behaviour of a single Object in terms of in which State it exists and the Transition between those States
- **Static attribute** - an attribute that does not relate to a specific object but is at class level; that is, an attribute that is common among all objects of that class
- **Static modeling** - use of class diagrams to describe structure
- **Static operation** - an operation that does not relate to a specific object but is at class level
- **Stereotype** - a notation allowing the extension of UML symbols. Some are defined within Profiles. Examples of predefined UML stereotypes are Actor, Exception, Powertype and Utility.
- **Structure diagram**
- **Superstate** - construct allowing several States which share common Transitions and Internal Activities
- **Swim lane** - synonym for Partition
- **System model** - The logical UML model being represented through one or more UML diagrams

T

- **Tagged values** - In extensibility
- **Template** - a Class that accepts a compile-time parameter defining the Type to be used within the Class; often to implement Collections of any Type
- **Thread** - a sequence of instructions whose execution is being scheduled by the Operating System and may run in parallel with other threads

- **Timing Diagram**
- **Token** - symbolises the Thread of an Activity diagram
- **Transition** - movement from one State to another in a State machine diagram. The transition is specified by its *Trigger-signature [Guard]/Activity*
- **Type** - the options are: an elementary Value type such as integer, string, date, or boolean or a Reference type defined in a Class

U

- **Use case** - a technique for capturing functional requirements of systems

V

- **Visibility** - the availability for access of elements in a model. Typically used to limit the visibility of features defined by a Class (e.g., attributes, operations). When applied to features defined by a class, the standard options are:
    - private (-): available only within the Class in which it was defined. This is the most limited visibility
    - protected (#): available within the Class in which it was defined, and within any subclass of this class
    - package (~): available only within the Package which directly contains the defining Class
    - public (+): available to any Class that can see the defining Class. The is the least limited visibility.

W

- **Workflow** - Set of sequential steps which must be done to get a job done

X

- **XMI** - An OMG standard for exchanging metadata information via Extensible Markup Language (XML).