A Taxonomy on Component-Based Software Engineering Methods

Christian Bunse[1], Felix C. Freiling[2], and Nicole Levy[3]

[1] Fraunhofer Institut Experimentelles Software Engineering, Fraunhoferplatz 1, 67663 Kaiserslautern, Germany
`Christian.Bunse@iese.fraunhofer.de`
[2] Universität Mannheim, Informatik 1, 68131 Mannheim, Germany
[3] PRiSM, University of Versailles, 45 Av. Des Etats-Unis, 78035 Versailles, Cedex, France

Abstract. The component paradigm promises to address many of the productivity and quality problems currently faced by the software industry. However, its correct application requires systematic, methodological support. A wide range of theoretical and practical methods have been developed in the context of the component paradigm. A taxonomy of these methods can provide a tool for increasing the understanding of the ways in which component-based development is currently addressed and directions for future development. This paper outlines a taxonomy based on the fundamental criteria and definitions, and provides examples to justify this classification. It can therefore serve as a first orientation for new researchers interested in the area of component-based software engineering.

1 Introduction

Software is of growing importance in human society since it is contained at the core of nearly any modern product or service. However, the development processes of such software is undergoing a tremendous change due to market requirements for time-to-market and cost. This has been the major reason for the development of object technology and subsequently component-based software engineering techniques. These promise that software systems can be created with significantly less effort than in traditional approaches, simply by assembling the appropriate prefabricated parts. In popular computer terminology this is captured by the "plug and play" metaphor. As soon as the relevant parts have been "plugged" together, they should be able to "play" with each other in the resulting system.

To obtain the goals of the component paradigm, systematic methodological support is required. For this reason, over the years there has been a vast amount of research and development which incrementally established a large body of knowledge, known as Component-based Software Engineering (CBSE). Unfortunately, the number of available approaches is rapidly growing, and often developers are disoriented and unable to select and adopt the appropriate tools that can best facilitate their work.

This paper offers a coherent and comprehensive view of methods and technologies supporting CBSE activities. According to [14] any attempt to provide an abstract view of a complex and composite entity is inevitably exposed to risks: on the one hand, it is possible to oversimplify or confuse different issues and topics; on the other, there is a risk of providing a flat presentation with limited insight and abstraction. The approach presented in this paper is pragmatic, and tries to attempt to find a reason-able and useful compromise between these two opposite forces.

The paper is organized as follows. Section 2 shortly sketches existing approaches to the classification of software engineering methods. Section 3 illustrates the classification scheme that has been adopted in this paper by defining the basic terms, providing characterization criteria, briefly discussing the chosen approach, and introducing a simple approach for visualizing and analyzing the results. Section 4 describes the main existing methods that support and guide component-based software development activities. However, these only represent a small sample of the existing methods. Section 5 presents the results of applying the proposed taxonomy and briefly sketches some important topics that will likely have an impact to future research on methodological support for component-based software development. Finally, Section 6 proposes some concluding remarks.

2 Related Work

The field of software development methodologies is large and still rapidly growing. For example a popular link-list [7] contains links to more than 50 different methodologies. Even if the focus is narrowed to methods suitable for component-based development, the number of methods is still remarkably large. Therefore, several attempts have been made in the past to provide an overview of the field, the goal being threefold:

1. To inform those interested in understanding the technology (e.g., [7]).
2. To justify avoidance or acceptance of the technology (e.g., [11], [13], [17]).
3. To reveal open research issues (e.g., [4], [14]).

Unfortunately, these surveys are either outdated and/or address the field of (object-oriented) software development in general. No surveys, specifically targeted at open research issues, are currently available which present an in-depth analysis of component-based software development methods. This might be due to the fact that objectively evaluating methodologies is a difficult and complex task because of several reasons:

1. Comparing methodologies is often like comparing apples and oranges (e.g., differences in terminology which have a significant impact on the appropriateness and application).
2. Many methods are targeted or strongly influenced by specific context constraints (e.g., programming languages). Thus, the evaluation of methods requires an understanding of the target platform for which the methodology is intended.
3. Certain methodologies assume a "greenfield" development context (i.e., the project is separate, stand-alone, and has no need to integrate with existing applications). This assumption removes certain constraints the methodology may have to deal with [4].

4. The completeness of various methodologies varies drastically (i.e., some methods simply describe a process, others present a graphical notation, while still others combine both graphics and a process). The depth and completeness of each of these components varies significantly from method to method.

Thus, on the one hand, an in-depth comparison or survey requires an understanding of the culture underlying a particular development approach. However, on the other hand, before such a comparison may start a general taxonomy is needed to classify existing methods. First applications of such a taxonomy may already reveal some open research issues, but only an in-depth examination may reveal limitations and restrictions of methods and can be used as a basis for a research agenda in component-based software development.

3 Taxonomy

The goal of this paper is to present a taxonomy for classifying component-based development methods as a basis for further analysis. Each method presented in this and the following sections can be regarded as an attempt for formalizing the process of software development following specific principles. The taxonomy is based on experience, user needs, and the published state of the art/practice. In addition, following [4], as technology changes, methods will evolve. Consequently, a taxonomy flexible enough to capture the dynamic nature of development methods must avoid rigid and precise definitions. Its structure, will depend more on judgment than on scientific objectivity [4]. This means that the taxonomy will remain partly subjective.

3.1 Definitions

Component-based software development methods aim at enabling *humans* to perform software engineering *processes* to produce software *products* that are of value to their customers. Thus, the integration of people, processes and products is a key enabler and has to be reflected in a taxonomy of such methods. However, although terms like method, process, or product are widely used in software engineering there are many conflicting definitions. In the following we define the major terms used in the context of this chapter.

- *Method:* Following [21] a method is a systematic approach for developing a product which provides a definition of the activities to be performed and the artifacts (requirements specification, design, etc.) to be developed. In other words, a method is a codified set of practices (i.e., a series of steps, to build software) that may be repeatable carried out to produce software, and which are accompanied by additional material such as templates, tools, best-practice know-how, etc. In this sense, a method can be seen as a combination of consistent process and product models, enriched by experience.
- *Process model:* In general, a process model describes the tasks that are undertaken within a software project, and it shows how and what information needs to be communicated between tasks [6]. Typically, a process model is instantiated from a software life-cycle model (e.g., waterfall) and can be used to

expose the manner in which the defined development activity is going to be conducted. In the context of component-based software development, the process defines basic development steps as well as the composition of components, their quality, assurance, and deployment.

- *Product Model:* In general, the Product Model is the entire product information resource that describes the product completely and unambiguously. According to [21], a software product model includes any of the artifacts generated during a software project: those that are delivered (e.g., manuals or code) and those that are usually not delivered (e.g., specifications, design, etc.). For each of these there can be a variety of models (including notations) that characterize attributes of the product. In general, product models can be broken down into several categories, with static (i.e., based on static properties or structure) and dynamic models (i.e., based on the execution behavior) being the most important ones. Together they provide a comprehensive view of a software product. In the context of component-based software development, products are viewed as a composition of components, whereby some components are atomic and some are composed of simpler components. In addition, software components of various kinds exhibit tangible properties that impact the quality of software.
- *Framework:* In object-oriented systems a framework represents a set of classes that embodies an abstract design for solutions to a number of related problems [21]. Transferred to the component-based development, a framework can be viewed as a collection of components with extension and composition mechanisms regarding a certain type of application

3.2 Criteria

Software development is a complex task which requires experience and knowledge. In order to systematically obtain valuable results it therefore has to be based on solid methodological grounds. It is simply impossible to develop component-based systems effectively by simply "writing code". In this sense the term "component-based software engineering methods" means all concepts, notations, guidelines, and techniques that can be used in creating better and more effective software systems. Many of these are embedded in or supported by a specific technology (e.g., design and coding toolsets). In other cases, methods are just concepts and knowledge that software developers have to in their daily work.

According to [14] methods include four different entities: principles, development techniques, meta-methods, as well as styles and patterns. Principles are essential ingredients of all software engineering methods and include concepts such as modularity, flexibility, robustness, interoperability, and quality. In order to obtain modularity other principles (e.g., information hiding, hierarchical decomposition, cohesion and decoupling) have to be in place. Flexibility in turn, also known as design for change, requires properties like extensibility, and scalability. Quality is represented by principles such as usability, reliability, or efficiency.

Software development activities are carried out according to a number of different development techniques. They can be roughly organized in three classes:

- *Informal* approaches are not based on any formal syntax or semantics. They simply state a number of guidelines and principles that should be followed in software development activities.
- *Semi-formal* approaches are techniques that include some more formal concepts. Typically, they exploit notations that do have syntax, but lack formal semantics.
- *Formal* approaches not only exploit formal notations but also use formal semantics based on mathematics and logic.

Strongly related to the formality of a method is the level of abstraction it typically operates on. On a rough scale three different levels (i.e., low, medium, and high) can be distinguished, corresponding to the level of detail typically obtained in different life-cycle phases (e.g., *high* corresponds to the requirements/analysis phase whereby *low* corresponds to the implementation phase.).

Another important property, especially in the area of component-based development, is the underlying development strategy of a method. Here we simply can distinguish between top-down and bottom-up strategies or a mixture of these two. In the component domain the strategy describes in which order components are specified and assembled.

The goal of each method is to support humans in developing a product. However, software systems are diverse in nature and have specific needs concerning methodological support. Therefore it is important to know which properties of a system are addressed. Here we can distinguish between system properties such as support for functional and non-functional properties, or development properties such as process and product.

Finally, it is important to mention the domain a method is targeted at. A method domain represents a scientific classification of software systems [26] which are targeted by a method, and should not be confused with the application domain of a system (e.g., logistics). Typically different domains have specific requirements concerning documentation, modeling languages, standards, and concepts. Following the SEI classification schema [26], domains can roughly be organized in the following groups:

- *General* – This group contains those methods which are more a process framework than a concrete method (e.g., Unified Process) and which claim to support software development in every domain.
- *Artificial Intelligence* – Systems concerned with basic models of behavior and the building of virtual and actual machines to simulate animal and human behavior.
- *Information Systems* – Systems concerned with file systems, database systems, and database models.
- *Human-Computer Interaction* – Systems concerned with user interfaces, computer graphics, and hypertext/hypermedia.
- *Numerical and Symbolic Computing* – Systems concerned with methods for efficiently and accurately using computers to solve equations for mathematical models.

- *Computer Simulation* – Systems concerned with the basic aspects of lmodeling and simulation (i.e., statistical models, queuing theory, variable generation, discrete simulation, etc.).
- *Real-Time Systems* – Systems concerned with knowledge about the development of real-time and embedded software systems (e.g., automotive).

In summary, Fig. 1 presents an overview on the proposed taxonomy for component-based software engineering methods.

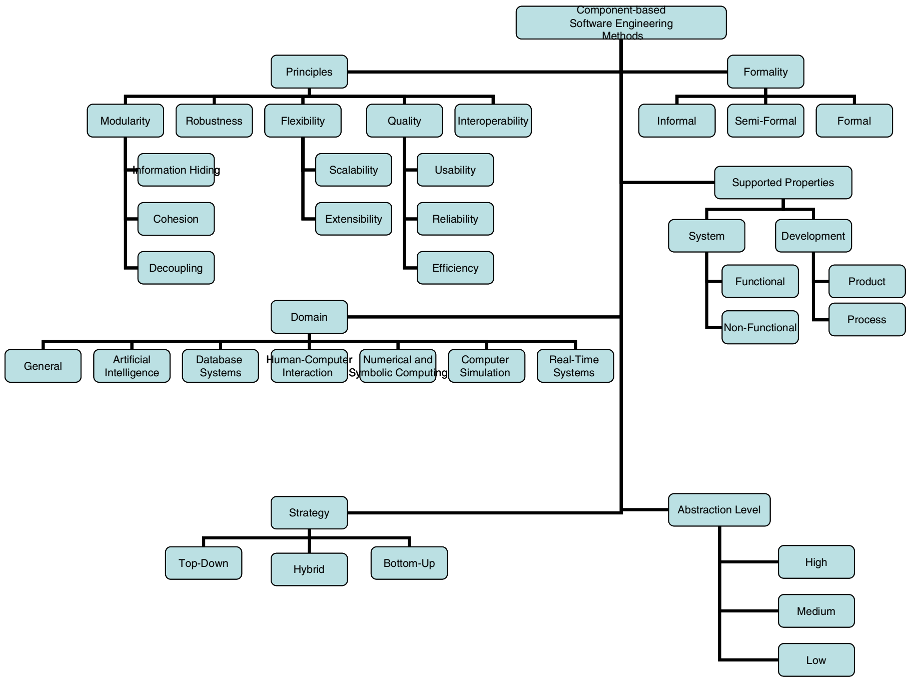


Fig. 1. CBSE Taxonomy

3.3 Applying the Taxonomy

In the previous subsections, basic definitions and criteria have been given to characterize component-based software development methods. Together these define the dimensions of the taxonomy and provide a schema to characterize every available method. However, it is important to stress that the taxonomy should not be considered exhaustive or a finished work. In the first instance, it deliberately did not address all possible aspects of component-based development. In the second instance, the taxonomy itself is subject to continuous evolution, since the elements that it classifies continue to evolve, due to scientific and technological advances in the field. In the following it is discussed how to apply the taxonomy to position some concrete methods, and how to draw conclusions and recommendations from the collected data.

In principle, the taxonomy will be applied to the methods presented in section four. These methods have been selected because of their different nature and coverage of the field. As such, this should also be reflected in their comparison based on the taxonomy. This comparison can then be used to identify to which extent the methods complement each other. Another typical use of the taxonomy is to compare methods that share the same or a similar purpose. This allows the identification of differences, strengths and weaknesses.

3.4 Presentation

The goal of a taxonomy is not only to provide a comprehensive overview (e.g., in form of tables) but also to identify white-spots and areas where future work should take place. The latter requires a simple and easy to compare representation of the collected and characterized data. This can be achieved by the application of radar or spider-web charts (see Fig. 2 for an example). These are not only useful to look at several different factors related to one item, but also to overlay several of them to have a quick overview on multiple items.

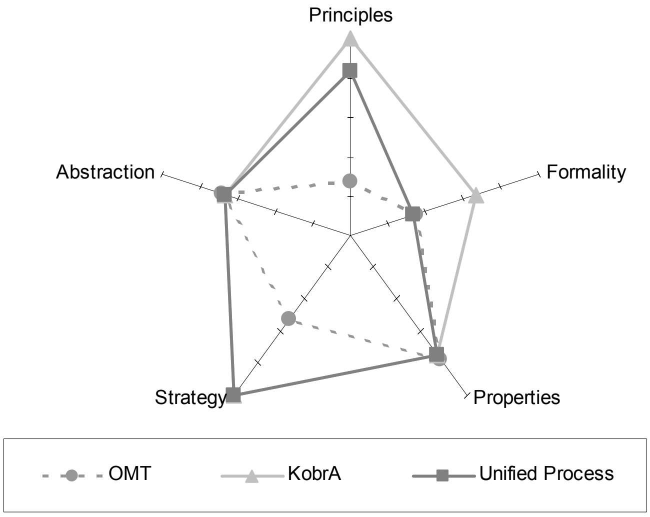


Fig. 2. Spider-Web Chart Example

Spider-web charts have multiple axes along which data can be plotted. A point close to the center on any axis indicates a low value, and a point near the edge a high value. Within this paper we define a general spider-web chart which has an axis for each taxonomy criteria. The single values are then plotted along these axes and connected by a single line, resulting in an individual shape for each characterized method. By placing the single charts on top of each other a common method chart is created which allows to quickly identifying white-spots and areas which warrant future research.

4 Methods for Component-Based Software Development

4.1 First Generation Object-Oriented Methods

Almost all modern software development approaches have roots in the first generation of object-oriented methods. The explosion in the number of object-oriented methods in the early 1990's, and the subsequent cross-fertilization of ideas, makes it very difficult to trace the precise influence and contribution of each method. Other well known first generation methods such as Shlaer/Mellor [28], Objectory [18], Booch [5] and Coad/Yourdon [9] are not discussed explicitly because their influence has been less direct. Since they were developed relatively early in the history of object technology, none of these methods directly considers components, design patterns and product lines. Nevertheless, they all embody important ideas that can be helpful in using these technologies.

4.1.1 OMT

Probably the single most influential method in the evolution of object-oriented development methods is the OMT (Object Modeling Technique) method developed by Rumbaugh et al. [25]. Many of its ideas have been adopted in ensuing generations of methods and notations. Its two most significant legacies are the UML [23], whose notational concepts are primarily based on OMT and its approach to analysis. In detail, OMT identifies three distinct but overlapping models to be generated during the development of a system:

- Object Model, which identifies the user visible data abstractions that the system manipulates, and the relationships between them.
- Dynamic Model, which describes the dynamic behavior of the system in terms of states, events and object interactions.
- Functional Model, which shows the computations or data transformation performed by the system.

The analysis phase is followed by a design phase in which the system is divided into subsystems and algorithms are designed for the methods of the identified classes. Although it is the strongest part of the method, OMT's analysis approach has one major weakness. The functional model is difficult to be kept consistent with the other analysis models. In practice, therefore, OMT users often ignore this model. The design process is the major shortcoming of OMT as a general purpose method. It is much less well defined than the analysis process, and lacks any kind of support for incremental development.

4.1.2 Fusion

Fusion [10] is a descendent of OMT, which fixed some of the problems with OMT's analysis approach, and significantly enhanced the design process. Fusion also pointed the way towards increased rigor and prescriptiveness in object-oriented development.

The analysis phase adopts the same three basic viewpoints as OMT, but with slightly different models and terminology. One of the main innovations was to require that textual operation schemata describe the effects of system operations in terms of the concepts in the object model. This improved the presentation of the functional model and introduced a set of consistency rules between the functional and object model.

The other main contribution of Fusion was the use of interaction modeling. It showed that the design process could be organized in a highly systematic fashion based on the elaboration and documentation of interaction scenarios on an operation-by-operation basis. The basic idea in the Fusion design phase is to create a collaboration diagram that documents how each system operation is realized in terms of lower level interactions. Unfortunately, Fusion suffers from the same problem as OMT in having a "flat" waterfall-based process model.

4.1.3 ROOM

The Real-Time Object-Oriented Modeling (ROOM) [27] method views software systems based on the concept of interacting processes. The basic building blocks of ROOM, known as actors, are active "logical machines", rather than simple ADTs, and typically encapsulate an active thread or process as well as state information. At the language level they therefore correspond to active constructs such as tasks in Ada or threads in Java.

Although ROOM was published before component-based development became a buzzword and dedicated component technologies (e.g., JavaBeans) became available, it contains all the basic characteristics of a classic run-time component model. Actors represent self-contained, autonomous components that can be realized as hardware elements as well as software elements, ports represent independently defined interfaces that allow components to be connected together in arbitrary configurations, and bindings represent concrete connectors that link components together to solve particular problems.

The main problem with ROOM is its lack of integration with data-modeling. Although ROOM defines an advanced and highly systematic way of using state machine diagrams, it makes little use of the core object modeling concepts such as associations, attributes and multiplicities etc. and gives little indication of how they fit in. This is a symptom of the fact that ROOM is more process (i.e., thread) oriented than data oriented.

4.1.4 HOOD

The Hierarchical Object-Oriented Design (HOOD) method [16] is largely limited to the European Space Agency (HOOD's creator) and its contractors. However, it has powerful and unique concepts not found in any other methods.

HOOD views a system as a community of objects organized in terms of two hierarchies: the seniority and the usage hierarchy. The first reflects the containment of objects within one another, and always yields a tree structure. The second reflects the usage of one object by another, in the sense of a client-server relationship. The key idea is to organize the development steps around the containment hierarchy (or seniority hierarchy). The overall development approach is thus one of recursive, top-down refinement in which progressively smaller objects are identified, modeled and implemented.

4.1.5 OORAM

OORAM [24] introduced some important ideas relating to the way in which systems can be modeled. The key innovation is to focus on roles throughout the modeling process as a way of tying different perspectives of a system together. A role essentially defines how a client object sees a server object, including its operations, behavior and needs.

Unfortunately the overall lifecycle process adopted in OORAM is essentially a waterfall model, although there is a high level of iteration within the major phases such as analysis and design. Object containment and hierarchical development play no part in the process, and thus it is essentially a "flat" method like OMT and Fusion.

4.2 Component-Oriented Methods

In view of the importance of the component paradigm, most modern methods aim to accommodate them in one form or another, but generally components are viewed as just a convenient implementation tool rather than an integral part of the overall software development cycle. In the last few years, however, methods have emerged which orient the whole development process around components and view them as richer abstractions than just binary code modules.

4.2.1 Catalysis

Catalysis [12] was one of the first methods developed specifically to leverage the UML [23] in connection with component based development, and embraces many of the other reuse technologies (i.e., architectural styles, design patterns and frameworks). The method either introduced or popularized many of the ideas that today are considered natural ingredients of component-based development, and several of these have been explicitly adopted in the UML.

Catalysis uses an iterative and incremental process based on cleanly defined abstraction and refinement mechanisms. These mechanisms are applied throughout system development from early analysis to implementation and set up the basis for recursive relationships between models, which then support forward- and re-engineering of systems. Catalysis makes use of the UML [23] with strong semantic consistency and completeness criteria based on a small set of ’core’ constructs.

4.2.2 KobrA

The KobrA method [1], developed at Fraunhofer IESE, propagates the use of components throughout all phases of the software life cycle. This goal is achieved by integrating the three most important software-engineering paradigms today: Components, Product Lines, and Model Driven Architectures (MDA). In addition, the KobrA method comes equipped with powerful means to achieve continuous, model-driven quality assurance. So far the KobrA method has only mainly been developed for system engineering in the domain of ERP systems. A common problem in all component-based development methods is their complete lack of capability to support the non-functional requirements.

4.2.3 MARMOT

MARMOT (Method for Component-Based Real-Time Object-oriented Development and Testing) [22], a descendant of KobrA, is specifically geared towards embedded and real-time system development in an object and component-oriented context. It subsumes the powerful principles of the KobrA, but provides additional features, that are particularly important in embedded, real-time application construction. MARMOT is based upon fundamental principles (i.e., software/hardware integration, aspect-orientation, real-time specification and scheduling, etc.) that are fully in line with the KobrA method's meta-model.

4.2.4 Select Perspective

Select Perspective [2] emphasizes the importance of business process modeling as the starting point of development, and follows a clean process that transforms system-independent business processes into implementation-oriented models of the system of interest in a step-by-step way. This includes the explicit identification of components, as well as the potential integration of legacy systems.

Select Perspective is particularly rich in practical recommendations and guidelines. It defines most of the essential ingredients needed for component-based development in the early stages of the software life cycle, and provides some useful guidelines for their application. Unlike other methods, it also explains the role that component technology can play in integrating legacy systems into new applications, and suggests how this can be achieved. However, its main weakness is that it is not always clear which aspects of the underlying business objects are being described by which models. In other words, the distribution of the information describing a business object is somewhat arbitrary.

4.2.5 UML Components

UML Components [8] focuses on the specification of components using the UML [23]. The method identifies two main phases (or workflows): the requirements workflow which captures the basic needs that the system must fulfill in terms of use cases and high level business classes, and the specification workflow which documents the business types, interfaces, and components that have been chosen to satisfy these requirements.

In essence, UML components offer a subset of Catalysis concepts but with a much simpler, UP-flavored process. This is both its strength and weakness. On the one hand it packages a core subset of the Catalysis concepts in a more accessible and prescriptive way, but on the other, it loses some of the key ideas of Catalysis, including the nesting of components to arbitrary depths, the recursive application of development concepts, and the use of frameworks to package larger-grained reusable structures than interface and components. Nevertheless, the early emphasis on the definition of component architectures in terms of component instances and their connections, and the enhancement of the idea of focusing diagrams on individual components or interfaces, represent valuable insights.

4.3 Product-Line Oriented Methods

With the possible exception of Catalysis, the methods described to this point are focused on the development of single systems. The creation of system variants takes place as part of the maintenance activity, and is generally viewed as a repeated application of the method rather than an integral part of the method itself. With the growing recognition of the value of a product line approach to the software life cycle, several methods have emerged in recent years that focus on product-line oriented software development and maintenance. Catalysis can support such an approach thanks to its advanced framework concepts, but product-lines are not its main focus. The product-line oriented methods vary in their degree of customizability, and the level of abstraction at which they address the variability's and commonalities in a product family.

4.3.1 FODA

Feature-Oriented Domain Analysis (FODA) [19], published by the Software Engineering Institute, relies on the basic idea that a domain is analyzed to identify the features which a system in this domain must or may provide. These features are hierarchically represented within a feature model. Features are recursively composed of other features, with some features being optional or alternatives to other features. A feature model thus serves as a useful input to the designers of a reference architecture for the domain.

Unfortunately, FODA is not described in sufficient detail to be easily applied without guidance. Nevertheless, the feature model is a useful way of capturing commonality and variability within a system family, and adds value to methods that focus on software reuse and product lines.

4.3.2 FAST

The core of FAST [29] is the commonality analysis to identify commonalities in a family of systems in terms of general textual statements. This is used as the basis for "implementing the domain", which involves the creation of domain-specific languages, architectures, generators, etc. to facilitate the low-effort creation of new members of the product family.

One problem of FAST is that it only addresses the product line issues at a very high level of abstraction, akin to the analysis level in conventional development methods. The critical connection to concrete implementation technologies is not directly addressed. Moreover, the guidelines provided for the domain analysis process are vague and unprescriptive.

4.3.3 PuLSE

PuLSE [3], developed at Fraunhofer IESE, splits the life-cycle of a system into four phases: initialization, product line infrastructure construction, usage, and evolution. It provides technical components for the different deployment, which itself are customizable to the context. Unfortunately PuLSE suffers from the same basic problem as other approaches due to its focus on the description of family properties at a very high-level of abstraction without giving concrete guidance on how the required flexibility should be realized at the implementation level.

4.4 Object-Oriented Method Frameworks

Object-oriented methods have come a long way since the early approaches mentioned above. Not only have they become more sophisticated, they have had to embrace a significant set of new concepts, such as components, architectures, frameworks, use cases and incremental development. One strategy for accommodating all these ideas in a coherent way is to raise the level of abstraction in which a process is described and make it more generic. This leads to methods that are compatible with a large number of specific development strategies, but are not ready to use "out of the box". They must consequently be tailored to the needs of specific projects. Such approaches, of which there are two main examples, are therefore often characterized as method frameworks.

4.4.1 Unified Process

The Unified Software Development Process [20] has been developed to provide a unified process to support the full power of the UML [23]. It can be characterized as a component-based, use-case-driven, architecture-centric, iterative, and incremental software development method. In principle the Unified Process iterates a series of cycles, whereby a cycle consist of four phases: Inception, Elaboration, Construction, and Transition. In addition, the Unified Process defines various workflows, the most prominent being Requirements, Analysis, Design, Implementation, and Test, which are carried out to a specific extent in each phase of a cycle. In general the Unified Process focuses more on management (e.g., workflows planning, evaluation, business modeling, etc.) than on technical issues, and provides most support, due to its origins in OMT, the Booch method, and Objectory, to modeling with only a high-level add-on for other phases of development.

4.4.2 OPEN

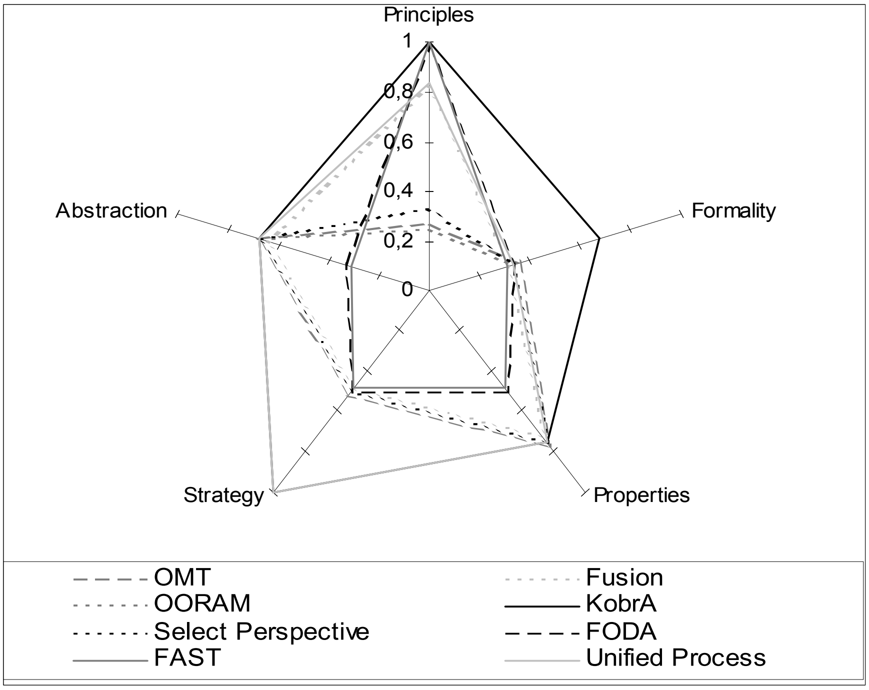
OPEN (Object-Oriented, Process, Environment and Notation) [15] initially encompassed a unified notation, known as the OML as well as a process, but the former has been subsumed by the UML standardization effort. In its current form OPEN can be characterized as a highly generic process framework oriented towards development with the UML.

Like the Unified Process, the generic nature of OPEN is a double edged sword. On the one hand it means that the ideas of OPEN are applicable, when properly instantiated, to a very wide range of domains. On the other hand, it means that much of the difficulty in using the method is wrapped up in the instantiation process. Poor or incorrect instantiation can easily lead to an incoherent process with very little chance of success. Unfortunately, the instantiation of the OPEN process is still one of its least well-developed parts. Also, in trying to integrate all acclaimed object-oriented techniques, including components, architectural styles and design patterns, OPEN suffers from the same "feature overload" problem as the Unified Process.

5 Results

The methods presented in the previous section represent only a small sample of all available development methods, although the list seems to be quite complete concerning component-based development. However, even this small selection shows that the knowledge required to analyze existing methods and to identify areas for future research is large and quite diverse. The taxonomy presented in Section 3 was used to get a first overview by applying it to the methods presented in the previous section.

Instead of presenting large tables we use a two-part radar chart based on the highest level taxonomy items (i.e., principles, formality, abstraction, strategy and properties).



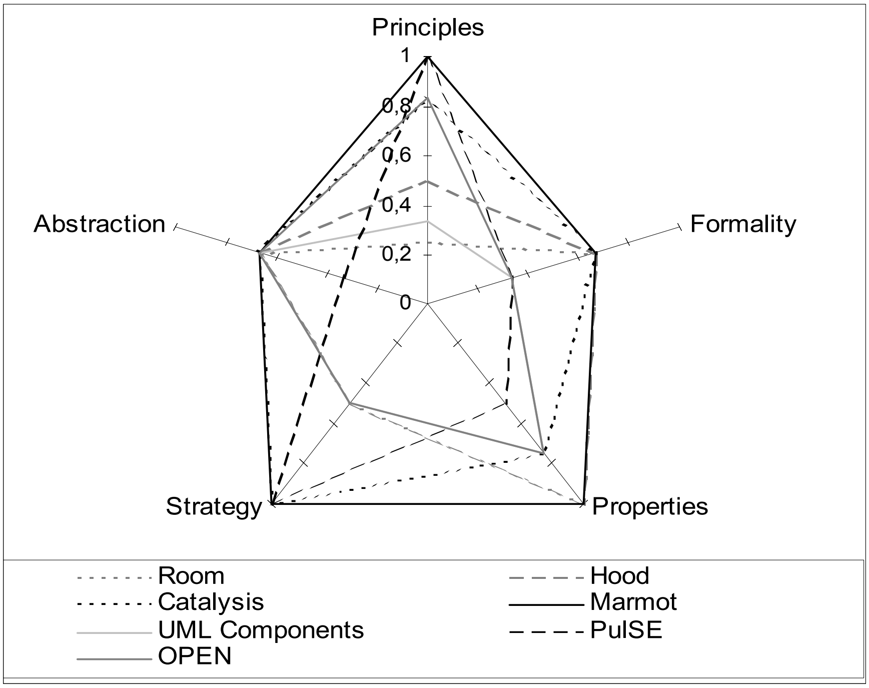


Fig. 3. Spiderweb - Development Methods

The two parts of this chart with together 15 methods are presented in **Fig. 3**. To obtain values for the top-level items the following process was applied: (1) Atomic values have been assigned a 0 if non-existent or 1 if existent. (2) For all sub-items (i.e., modularity) the mean of its atomic values was calculated. (3) Finally, the mean of all sub-items for each of the top items was calculated and the visualized in the radar chart.

For example, for the OMT method, we assigned for Formality the following values: 1 for Informal and 0 for Semi-formal and Formal. Therefore, the value assigned to OMT for the Formality axis is the average of these 3 values, which is 0.33.

At a first glance **Fig. 3** shows that the required properties, principles and strategies for component-based development are covered. Especially component specific methods such as Catalysis or KobrA have made significant advances. However, most methods fail when it comes to non-functional properties or support for the lower levels of abstraction. The latter might be based on the fact that methods tend to focus on the early phases of development (i.e., requirements, analysis, and design), neglecting the link to the implementation phase. Often this link is seen as a responsibility of tools (code generation) or developers. However, the recent advent of the Model-Driven-Architecture (MDA) approach has shown models and code can be tightly linked and that this not only increases development speed but also has a positive impact on the overall quality.

Software becomes more and more prominent also in domains such as aviation, automotive, or even consumer-electronics. However, these systems, often characterized as embedded systems, have specific requirements concerning safety, performance, or timing. These non-functional properties dominate the development of such systems to a large extent. Therefore, systematic development methods have to provide support for handling and quality assurance of such properties. In the area of component-based development this becomes even more important. Assembling systems out of pre-fabricated components requires mechanisms to assess specific non-functional properties as well as actions to optimize and handle them properly.

6 Summary and Conclusions

This paper has briefly presented a taxonomy for classifying component-based software engineering methods on a high level of abstraction. This taxonomy has then been applied on a small selection of published methods. However, it has to be stated that the paper focused on some key criteria in order to identify areas which warrant future research, rather than presenting specific features of a method in detail.

In summary the taxonomy showed that methodological support for component-based software development has made significant advances, compared to early methods, such as OMT. However, the methods available today are no silver-bullets and sensitive against the requirements of different domains and system types. More specifically, the area of safety-critical systems requiring formal development and support for addressing non-functional properties warrants future research. Another problem is the level of detail or abstraction covered by single methods. It seems that methodological support is missing for the lowest abstraction levels (e.g., implementation). The reason might be that the link between models, as used in analysis and design, and the corresponding source-code is weak. The advent of the MDA paradigm might offer a solution.

However, it must be clear that, as in any similar effort, this is just one step forward in a never-ending process. New results in technology development as well as in software engineering theory require further extensions and enhancements concerning the studied methods. Thus, the classification attempt made in this chapter may have to be revised and, eventually, deeply changed.

Following [14] it must be stated, that software engineering is a dynamic and challenging discipline, where novel approaches and technologies emerge as our understanding of software increases and deepens. Thus the classification of methods is an essential aid to researchers and practitioners.

References

1. Atkinson, C., Bayer, J., Bunse, C. et al. Component-Based Product Line Engineering with UML. Pearson, 2001.
2. Aonix A.: Service and component based development: using the select perspective. Addison-Wesley Longman Publishing Co, Inc., Boston, MA, 2003 ACM Computing Reviews 9 (2003)
3. Authors: Bayer, J., Flege, O., Knauber, P., Laqua, R., Muthig, D., Schmid, K., Widen, T., DeBaud, J.M., Proceedings of the Fifth ACM SIGSOFT Symposium on Software Reusability (SSR'99), (Los Angeles, CA, USA), May 1999, pp. 122-131
4. Blum, B.I., A Taxonomy of Software Development Methods, Communication of the ACM Vol 37, No. 11, 1994
5. Booch. B., Object Oriented Analysis and Design with Applications. Benjamin/Cummings, Redwood City, California, 2nd edition, 1994.
6. Bunse, C., von Knethen, A., Vorgehensmodelle Kompakt, Spektrum Verlag, 2001
7. Links on Objects & Components, Pages at the WWW last visited June 2005, http://www.cetus-links.org/
8. Cheesman, J. and Daniels, J., UML Components: A simple Process for Specifying Component-Based Software, Addison-Wesley, 2000.
9. Coad, P., Yourdon. E., Object-Oriented Analysis. Prentice Hall, 1991.
10. Coleman, D. Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H., Hayes, F., Jeremaes, P. Object-Oriented Development: The Fusion Method. Prentice Hall, 1993.
11. J. Cribbs, C. Roe, and S. Moon, An Evaluation of Object-Oriented Analysis and Design Methodologies, SIGS Books, New York, New York, 1992. 75 pages.
12. D'Souza, D. F. and Wills A. C., Objects, Components and Frameworks with UML: The Catalysis Approach, Addison-Wesley, 1998.
13. R.G. Fichman and C.F. Kemerer, Object-Oriented and Conventional Analysis and Development Methodologies: Comparison and Critique, Center for Information Systems Research, Sloan School of Management, M.I.T., CISR WP. No. 230, 1991. 38 pages.
14. Fuggetta, A., Sfardini, L., Software Engineering Methods and Technologies, Technical Report, Cefriel, 2004
15. Graham, I., Henderson-Sellers, B., and Younessi, H., The OPEN Process Specification, Addison Wesley 1997.
16. HUM Working Group, HOOD User Manual, HOOD User Group, July 1994
17. Hutt, T.F. (ed.), Object Analysis and Design – Description of Methods, OMG Press, 1994
18. Jacobson, I., Christerson, M., Jonsson,P. Object-Oriented Software Engineering - A Use Case Driven Approach, Addison-Wesley, 1992

19. Kang, K,C., Cohen S.G., Novak, W.E,. Peterson, A.S., Feature-Oriented Domain Analysis (FODA) Feasibility Study, Tech. Report CMU/SEI-90-TR-21, Software Engineering Institute (SEI), November 1990
20. Kruchten, P. B., The Rational Unified Process. An Introduction, Addison-Wesley, 2000.
21. Marciniak, J.J. (Ed.), Encyclopedia of Software Engineering (2nd ed.), John Wiley & Sons, 2002
22. MARMOT homepage. to be found at www.marmot-project.org, 2005.
23. Object Management Group. Unified Modeling Language Specification. 2000.
24. Reenskaug, T., Wold, P., Lehne, O., Working with Objects: The OOram Software Development Method, Manning/Prentice Hall 1996.
25. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W. Object-Oriented Modeling and Design. Prentice Hall, 1991.
26. Software Engineering Institute. Software Engineering Body of Knowledge Version 1.0, available at www.sei.cmu.edu/publications/documents/99.reports/99tr004/99tr004sd.html
27. Selic, B., Gullekson, G. and Ward, P.T., Real-Time Object-Oriented Modeling, John Wiley & Sons, 1994.
28. Shlaer, S., Mellor, S.J.. The shlaer-mellor method. Pages on the WWW which can be found at: http://www.projtech.com/, 1998.
29. Weiss, D. M. and Lai, C. T. R., Software Product Line Engineering: A family Based Software Engineering Process, Addison-Wesley, 1999