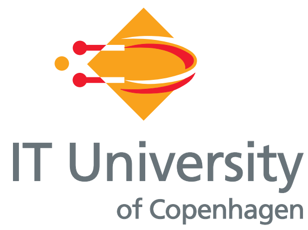


The 4P Taxonomy

A Survey of Software Development Environments

Anders Hessellund

IT University Technical Report Series **TR-2006-86**

ISSN 1600–6100 **June 2006**

Copyright © 2006, Anders Hessellund

IT University of Copenhagen
All rights reserved.

Reproduction of all or part of this work is permitted for educational or research use on condition that this copyright notice is included in any copy.

ISSN 1600–6100

ISBN 87-7949-127-8

Copies may be obtained by contacting:

IT University of Copenhagen
Rued Langgaards Vej 7
DK-2300 Copenhagen S
Denmark

Telephone: +45 72 18 50 00
Telefax: +45 72 18 50 01
Web `www.itu.dk`

Contents

1 Introduction **3**

2 Theoretical framework **4**
2.1 Work metaphors 5
2.2 A taxonomy of concepts 7

3 Program **9**
3.1 Source artifacts 10
3.2 Representation 12
3.3 Consistency 16
3.4 Target product 19

4 Platform **20**
4.1 Toolkit 20
4.2 Presentation 23
4.3 Function 28
4.4 Inconsistency Management 31

5 People **33**
5.1 Area of Expertise 34
5.2 Level of Expertise 35
5.3 Organization 36
5.4 Mediation 36

6 Process **37**
6.1 Life Cycle 37
6.2 Method 39
6.3 Configuration Management 40
6.4 Decomposition Techniques 42

7 Applying the taxonomy **42**
7.1 Program 44
7.2 Platform 46
7.3 People 47
7.4 Process 48

8 Related work **49**

9 Conclusion **51**

1 Introduction

The conceptual architecture of *software development environments*[1] has a tremendous impact on the way we perceive the nature of software development. Our environments form the way we construct software systems. More importantly, they also direct and to a large extent restrict our perception of possible implementation choices. A software development environment consist of the tools that we have at our disposal. Strengths and weaknesses of the tools in our toolkits often influence the nature of final products as well as the methods we employ in order to create these final products. The ever growing complexity of software projects only increases our dependency on environments and thereby enhance their effect on our work. Without a proper conceptualization of this field, we are unable to understand, distinguish and choose between the environments available.

In a classical survey *Contemporary Software Development Environments* [1] from 1982, Howden observes that there are already more than 400 software development tools available on the market. The field has only grown during the 24 years that have passed. Consequently, it is simply not possible to write a complete survey of the plethora of existing tools. Nevertheless, developers and managers still have to make decisions about which tools to use, and predict how well these tools will fulfill their concrete requirements. Such decisions are often based on vague intuitions, media hype and hear-say from random fora. In order to properly understand the implications of such a decision, it is necessary to understand what the tools actually offer and how they fit into the broader context of software development.

The purpose of this paper is to introduce a conceptual framework that will allow us to gain a deeper understanding of the nature and effect of software devlopment environments. We will use the words software development environment in a broad sense to denote a collection of tools, the development method and the people involved in the actual development. The key component in our framework is a taxonomy of software development environments called the *4P Taxonomy*. The four dimensions, our four Ps, of a software development environment are Program, Platform, People and Process. In this paper we will introduce and define these four dimensions and use examples from concrete environments to illustrate our ideas. This should provide the reader with a powerful cognitive tool that can be used to understand and distinguish between different concrete environments. Our ambition is that the framework should help our reader make more informed

[1]Note to the reader: The term *software development environment* only appears in italics here and will be written in regular text in the rest of the paper. The reader should note that the term denotes our broad definition which is explained in the rest of the paper.

choices between tools in the future.

In this paper, we will show that our proposed taxonomy can actually be applied to a small set of modern software development environments. This proves that our taxonomy can be used to understand actual environments and hence potentially provide us with the overview that we require. The key contributions of this paper are therefore a comprehensive taxonomy, an example of its application and an in-depth survey of a large part of the literature in this area.

This paper is organized in the following way: Section 2 introduces the overall theoretical framework in the form of a high-level view of the 4P taxonomy. Sections 3, 4, 5 and 6 elaborate on each of the four components in the taxonomy - Program, Platform, People and Process. Section 7 shows how the previously introduced taxonomy can be applied on a small selection of actual software development environments. Section 8 describes related work and shows some connections between our work and previous studies in this field. Section 9 provides a summary of the key contributions and suggest some directions for further research and development.

2 Theoretical framework

In order to understand the field of software development environments, we must establish a common frame of reference in the form of a set of key definitions. In this section, we will first motivate the need for conceptualization and examine the concept of a tool by considering some interesting analogies. Then we will introduce the 4P taxonomy that forms the conceptual foundation for our work.

Previous conceptual studies of software development environments have, as described in section 8, mainly been concerned with grouping tools into general categories. Few attempts have been made at drawing a map of the individual concepts which often span several such tool categories. Nørmark's report on *Programming Environments* [2] is one notable exception. Where Nørmark puts emphasis on the programming activity, we will try to take a wider perspective and include more of the actual development process. The central intention in the framework that we will describe here is to reveal the wealth of features, trade-offs, and distinctions that can be observed under the general heading of software development environments.

Initially, we will have to define the somewhat blurred concept of a software development environment. A software development environment consist of a platform of tools and infrastructure, a program under development, people that are involved, and a process that guides and directs all activities. It

should be observed that this is a much broader definition than the ordinary meaning of software development environment. Typically, software development environment is used in the sense *integrated development environment* which only covers a certain part of the whole process and of the people involved, viz., the programming activity and the programmers. As software projects become increasingly complex and development processes extend beyond simply the *write-compile-execute* style of working, we need to take a larger perspective on software development.

We need to look at the complete life cycle of a project from early notes on paper napkins over programming of thousands of lines of source code to the final configuration and deployment of a finished product. These different phases of the life cycle and the people involved in each of them form an intricate web of relations. Software development environments are required to manage this process and carefully handle the gradual refinement of a vague idea over detailed blueprints to a full executable system. Furthermore, the complex interactions of different people in various roles should be managed and the their work should be integrated seamlessly into the overall development.

Tool builders are faced with a daunting challenge if they aspire to deliver the perfect tool. Computer-Aided Software Engineering (CASE) tools and various forms of augmented Integrated Development Environments (IDE) have delivered great value to developers and managers, but the all-encompassing tool is yet to be seen. In order to produce promising future solutions, we must conceptualize the very idea of a tool and then slowly the different dimensions and facets of this idea. One approach can be to look at more traditional tools and gain some insights by analogy.

2.1 Work metaphors

When we examine our intuitions about the very idea of a tool, three analogies or metaphors come to mind. The first is the *artist* who relies on a few primitive tools, such as brush and paint, and his unconstrained creativity. The second is the *craftsman* with his stable toolkit and practical experience. The third is the *engineer* or *architect* who relies on models, abstraction and rigorous, theoretical education. We will examine these analogies or metaphors and extract the mode of work which is embedded in each of them.

The *artist* works with primitive tools on unique products. The guiding principle is creativity. An example is the single canvas painter who uses simple brushes and jars of paint to create great beauty. The tools are simple and flexible. They serve short-term goals and are not intended for complex functionality. It can be argued that exactly for short term, unique

product development, these tools are ideal. This, of course, assumes an extremely talented creator since there is rarely any methodological guidance. Collaboration is typically not required in this mode of working because the talented painter is a *lone ranger*[2].

In terms of software development environments, the artist is the expert programmer who single-handed creates amazing and efficient products which fulfills immediate requirements. The history of programming has several examples of such artists, for instance Don Knuth[3] (TEX [4]) and Jim Kent (GigAssembler [5]). The toolkit does not have to contain complex tools since the guiding principle is personal creativity rather than a computer-assisted method. The obvious problem with this work mode is that when programmers of lesser quality attempt to develop software in this fashion, the odds of a successful outcome are extremely bad. Mere humans have to rely on more than intuition and creativity in the face of complex problems. The division of labor that is needed to handle complexity is not possible in this work mode and that eliminates the hope of realistic project planning.

The second analogy is the often mentioned idea of the *craftsman*. The craftsman typically works with a fixed, well-known toolkit and follows methods based on experience and training. The products can be unique or mass-produced. Production always happens according to patterns and practices of the trade. Collaboration is sometimes a part of the craftsman's work because he is specialized within a field and recognizes his reliance on others in larger projects. A good example of this craftsman is the carpenter who can both create a unique desktop or fit a wooden door into a brick frame. In the first case he accomplishes his work alone and in the second case he produces and adapts a door to a frame created by others.

Programmers who develop programs according to the craftsman's work mode are skilled but not the programming equivalents of renaissance masters. The prototypical example of a craftsman in programming is the consultant. They rely on good tools and training. The tools might not assist them in every part of the work, but they are never unimportant. Similarly, the method is perhaps not formal or rigorous, but there are often certain patterns, heuristics and guidelines to follow. Collaboration is often handled informally, but a *lone ranger* attitude is not acceptable. The craftsman-programmer often handles complexity better although a project can of course become overwhelming and require a better tool and method. Nevertheless, in many concrete scenarios with medium-size problems the craftsman is often the most

[2]This does, of course, not apply to renaissance masters like Michelangelo who *directed* the painting of the Sistine Chapel ceiling rather than *painting* it singlehanded.

[3]Interestingly, Knuth devoted his Turing Award lecture in 1974 to theme of programming as an art [3].

cost-effective and reliable person.

The third and final analogy in this section is the *engineer* or *architect* as this role is sometimes also called. An engineer's toolkit consist of complex tools and rigorous methods. A central theme in this analogy is the idea of high-level abstractions. Where the artist and the craftsman can rely on intuition and experience, the engineer relies on explicitly defined higher-level abstractions or models. Complexity is handled by describing a problem and its solution at various levels and then dividing labor accordingly. Collaboration is a necessity and the concept of defined process replaces previous notions of heuristics and guidelines.

The choice of name for the field of *Software Engineering* accurately captures the aspiration of software development since the first international conference [6] where the word *engineering* appeared in the title. Software should be developed according to the engineer analogy. The growth of computational power and complexity requirements have pushed software development toward an engineering practice. Typical tools are high-level languages and models. Collaboration is structured by means of formal methods and rigorous procedures. This mode of work facilitates solution to extremely complex problems but also introduces the risk of over-engineering when simple problems are solved in unnecessarily complex ways.

In summary, we can conclude that the choice of software development environment should be made with respect to our current and future problems. If we expect a series of complex problems, we should choose tools that support us in a variety of ways and adopt methods that can help us manage the entire process. If, on the other hand, our problems are minor then we should reject the complex work modes and embrace simple, agile ways of problem solving. The requirements of collaboration, defined process and project planning also play an important role in our choice.

Our tool analogies suggest that the previously mentioned idea of a perfect tool is probably not a meaningful ideal. Tools should fit our mode of work rather than dreams of a universal tool. Programmers faced with tasks that resemble the artist's or the craftsman's are simply better off relying on their traditional tools. The engineers are the most obvious market segment for the universal tool. But even in this case, there are also requirements of an adaptable process and flexible tools which make any one-size-fits-all tool seem less desirable.

2.2 A taxonomy of concepts

With our three metaphors of work in mind, we can now begin to conceptualize the main dimensions of a software development environment and then unfold

these dimensions such that different facets, features and trade-offs become visible. Choosing a certain environment usually implies a rejection of a set of features. The purpose of a conceptual taxonomy is to reveal the often hidden implications of such a choice. In this section, we will present a high-level view of the taxonomy with a special focus on the four main dimensions. In the next sections, we will then unfold these dimensions in greater detail.

We have choosen to use the *cardinality-based feature modeling notation* [7, 8] to visualize our taxonomy. Feature diagrams are useful cognitive aids when developing taxonomies as they capture common and variable features in a hierarchic fashion. Table 1, which is adapted from Czarnecki et al. [7, 8], provides brief explanations of the notation. It should be noted that we do not pretend that the taxonomy is complete. The variability of our feature diagrams only cover the software development environments that we have looked at. The field of software development environment research is full of weird and exotic prototypes which are important but hard to capture in a general taxonomy. Our efforts should therefore only be seen as the first steps toward a general taxonomy.

Symbol	Explanation
F	Solitary feature with cardinality [1..1], i.e., *mandatory* feature
F	Solitary feature with cardinality [0..1], i.e., *optional* feature
[n..m] F	Solitary feature with cardinality $[n..m]$, $n \geq 0 \wedge m \geq n \wedge m > 1$, i.e., *mandatory clonable* feature
F	Feature model reference F
	Feature group with cardinality $\langle 1\text{–}1 \rangle$, i.e. *xor*-group
	Feature group with cardinality $\langle 1\text{–}k \rangle$, where k is the group size, i.e. *or*-group

Table 1: Cardinality-based feature modeling notation [7, 8]

We propose a taxonomy with four main dimensions as sketched in figure 1: Program, Platform, People and Process. A software development environment can be analyzed with respect to these four dimensions. First, the *program* is the artifact that is being made. It can consist of various different components and sub-artifacts which fit together in special ways. The program dimension covers this artifact from early inception and construction to deployment and maintenance. It is therefore necessary to reason about the status and form of the program as it evolves. Second, the *platform* is the foundation on which the program is created. The platform consist of the toolkit and technical facilities that enable the developer to realize a vision. Third, the *people* dimension describes the skills and roles of those using the

platform. Fourth, the *process* dimension covers the methods, procedures and management techniques by which the development progress and the collaboration is organized.

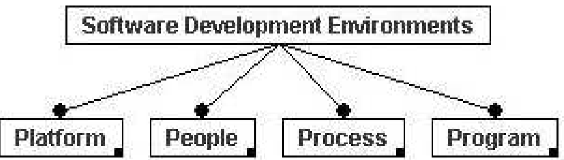


Figure 1: The four dimensions of a software development environment.

The analogy of the craftsman from the previous section can be used to illustrate these four dimensions. When a carpenter produces a wooden door, the door is his program. It is composed of different pieces of wood and evolves from unassembled materials to a finished and polished product. The carpenter uses a hammer, saw, screws and other tools from his toolkit and he conducts his work at a workbench. This toolkit and workbench is the platform that facilitates his work. The people dimension is simple in this case as we have only discussed the carpenter. He is endowed with a skill set and a certain degree of expertise which he employs in his work. Finally, the process dimension is visible in the steps he performs as well as his coordination with the people who create the frame. We encourage the reader to perform a similar analysis for the artist and the engineer to convince himself that these four dimensions are actually present in all three modes of work.

It should be obvious that these four dimensions are closely woven in most environments and that it can be extremely difficult to keep them separated. Nevertheless, the point of forcing concept into categories is exactly to be able to discuss and evaluate them. Tool builders have an interest in this since they have to consider how the different dimensions are represented in their concrete tool designs. If all design considerations have gone into establishing a strong platform then there might be a risk that the future users (i.e., the people) and their way of organizing work (i.e., the process) are left out. In order to create both powerful and useful environments we need to take all four dimensions into account.

3 Program

It seems a natural choice to start by unfolding the *program* dimension as shown in figure 2. The program is in effect the *raison d'être* of software development environments and should therefore be examined at some length.

In this section, we will conduct this examination by looking at four of the central facets that constitute this dimension. These four facets are the *source artifacts* that embody the sources of a program, the *representation* of these source artifacts within the environment, the nature and level of *consistency* of the source artifacts and representation, and finally the *domain* that the program targets. We do not pretend that these four dimensions constitute the totality of the program dimensions but they are sufficiently important to be treated here.

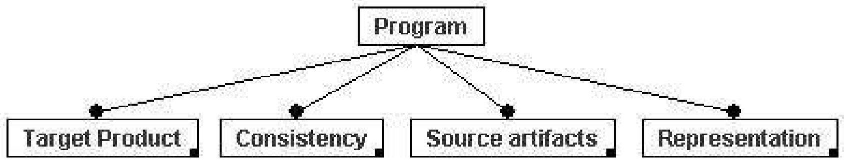


Figure 2: The *Program* dimension.

3.1 Source artifacts

In order to understand the concept of *source artifacts*, we must first distinguish between artifacts that are manually created and artifacts that are derived. We will denote the first category *sources* and the second category *targets*. The final stage of program development results in an artifact that we denote the *product*. This product is typically derived from various sources which are created and maintained throughout the development. During development, artifacts can appear both as sources and as targets. In code generation a target is derived from a generation process. This target can be source code which is then specialized and later used as a source artifact in a new derivation process.

In any derivation process, the target is the whole and the sources are the parts. It can be argued that the whole is more than the sum of its parts since the target, especially the final product, actually adds value to its consumers whereas the individual sources often are useless in isolation. On the other hand, it can also be argued that the parts are more than the whole because several of the sources never actually appear in the derived product. Initial design documents and sketches on paper napkins in the early phases of development are often left out of the final product. The program is more than what the customers receive. Many intermediate products of the development are of course not of any interest to the final customer. We claim that important design decisions are often lost due to the lack of support in the software development environments.

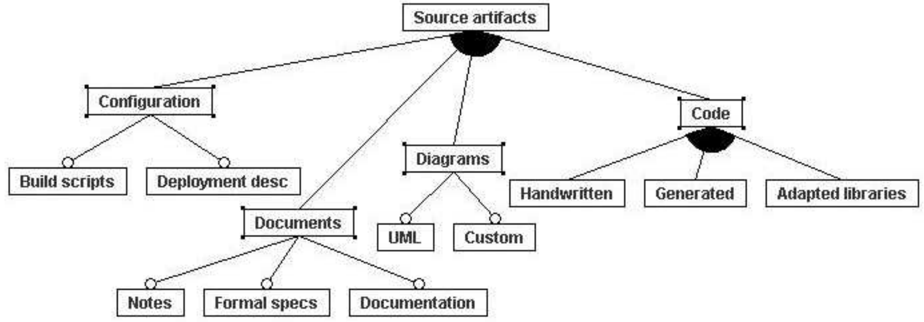


Figure 3: A few categories of source artifacts

The source artifacts, which we have ordered into categories in figure 3, can be requirement documents, analysis specifications, design diagrams, source code, adapted third party libraries, documentation and many other kinds. These source artifacts are either created or adapted during program development. Memos, notes, emails and chats are also source artifacts as they often embody relevant ideas, reasons and decisions. A software development environment must manage all these source artifacts and present the user with both the important parts and the essential relations. At the same time the environment must be able to sort out irrelevant details, such that all source artifacts are retrievable but only the important ones are immediately visible. Views and navigation features serve these purposes but we will return to these subjects in section 4.2 in the discussion of presentation.

The wave of *Computer-Aided Software Engineering* (CASE) tools that appeared in the eighties [9] was an important move toward management of a broader spectrum of different software source artifacts. Before that, software development environments had mainly been toolkits, such as the Programmer's Workbench [10] which extended the Unix operating system with various tools or Emacs [11, 12] which provided a thin, uniform interface to a range of tools. These tools did not provide any general scheme for management of different software source artifacts. One notable predecessor to CASE tools is the DRACO approach [13, 14] which introduces the idea of domains and transformations between domains in order to manage and integrate different source artifacts. We will discuss CASE tools in more detail in section 4.3.3.

Traditionally, source code artifacts have been the center of attention in research into software development environments. The current interest in model-driven development and model-driven architecture [15] have spawned an increased interest in analysis and design source artifacts. Analysis specifications and design diagrams that used to be merely documentation

have been promoted to source artifacts from which targets can be derived automatically. We consider this a recognition of the fact that software development environments must accommodate not only multiple types of source artifacts but also source artifacts at multiple levels of abstraction.

3.2 Representation

The source artifacts of a software development environment can be represented in various ways. *Representation*, which is illustrated in figure 4, covers the medium in which the source artifacts are stored, the computational form or forms which contain it, and the principle according to which developers can actually interact with this representation. This part of the program dimension is naturally closely related to the source artifacts as described in section 3.1. The design choices regarding representation also have a strong causal relation to both the way tools are integrated as described in section 4.1 and for the possibilities of establishing various views of the representation as described in section 4.2.

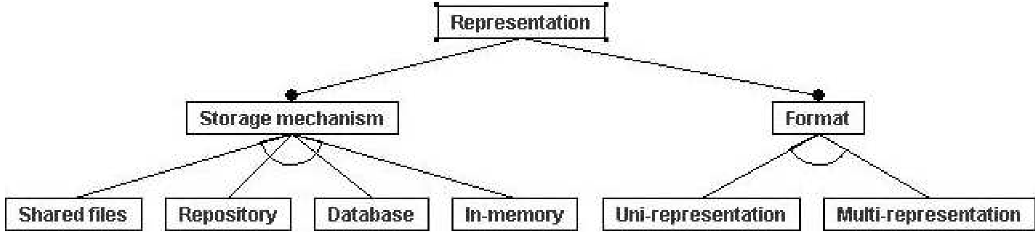


Figure 4: Representational facets

3.2.1 Storage mechanisms

There are various *storage mechanisms* which can be provided in a software development environment. Meyers [16] offers a detailed discussion of the storage trade-offs and design choices which we will use as reference. Often environments even provide several of these possibilities in order to meet the requirements of the developers.

The simplest form is *shared files* where *shared* can either denote sharing between tools or between users. Although the idea of using shared files as the main storage medium can seem primitive, it is actually surprisingly widely used. The main advantage of using shared files is that the developer has a feeling of control and understands the storage medium. The developer can interact with shared files through tools or simply by the mechanisms offered by the file system.

There are numerous disadvantages with this approach. First, different tools or users can overwrite changes in the shared files. Second, the translation process back and forth from various different files to a general, high-level abstraction that can be manipulated by other tools and displayed at the front end is difficult and error-prone. As an example of this problem, Meyers [16] states that shared files are not well-suited to support multiple views because different views may manipulate the file contents in incompatible ways. Third, duplicate effort can occur if different tools have to load and process the same files. Meyers [16] uses the example of editors and compilers that both have to parse source code which causes the environment to essentially perform the same task twice.

A more advanced storage form is a central *repository*. This can basically also be shared files but the key difference is that a repository further provides management facilities. Examples of this are Concurrent Version System (CVS) [17] and Subversion [18] which will be covered in greater depth in section 6.3. The main advantages of repositories are: first, the support for collaboration where the repository serves as standardized integration mechanism, and second, the administrative facilities which for instance provide audit trails and logging of all activities.

These repositories solve the problem of sharing files between users by giving each user a local copy (*sandbox*) that can be synchronized with the central repository. The synchronization is either based on a pessimistic or optimistic concurrency model depending on whether files in the repository can be locked. There is of course a range of potential problems related to the synchronization process, but case studies [17] have shown that these problems are, even with optimistic concurrency, not as frequent as one might fear. Repositories do not solve the problem of tool sharing within a developer's local copy. The local copies have similar characteristics as the previous model with shared files. Different tools can interact with the same file and cause both problems and duplicate work.

A third form of storage facility is a *database*. Databases offer advanced transactional facilities as well as strong support for keeping data consistent. These facilities solve both the problem of sharing data between users and tools as they are all subject to the transactional mechanisms of the database. Users and tools ideally work directly on the representation in the database and hence there is no requirement of keeping local and remote copies synchronized as in the repository model. This means that there is no real distinction between representation and storage in this case. Tools can work directly on the representation without needing to load and process stored files first. Another advantage is the possibility of representing data in a very fine-grained manner. Individual fragments of source artifacts can be stored

separately instead of being pressed into large files.

There are several disadvantages with databases. First, tools are as Meyers [16] observes often not capable of working directly on the representation in the database. Instead, tools often extract data and manipulate it in a tool-specific format. This basically corresponds to using the database as a repository which of course re-introduces all the previously mentioned synchronization issues. The root cause of the problem is that tools have to be tailored to use the representation in the database schema. Such a tailoring activity typically requires an understanding of the representation. Second, the transactional mechanisms of the database can turn against the user if multiple tools cause a deadlock. Unless the tools are coordinated on a higher level by the environment as described in section 4.1, deadlocks can be hard to avoid.

Some of the problems that databases face can be alleviated with the use of views as suggested by Garlan [19]. Tools interact with views rather than with the entire database. Garlan introduces the idea of compatibility maps as a means to share data among views. Views can be bundled into features for more advanced requirements but - as Meyers [16] states - the complexity can still overwhelm the developer when trying to comprehend a large structure. Another disadvantage is that it is difficult to add new tools as tool builders have to develop both new views and compatibility maps in order to integrate a tool into the environment.

A fourth storage mechanism is an *in-memory* representation. Like the database approaches, the in-memory model does not distinguish between storage and representation. Tools work directly on this single in-memory representation which Meyers [16] calls the *Holy Grail* of environment integration. This model is not directly suitable in multi-user scenarios. In an individual developer's software development environment, the in-memory representation can provide logging, transactional mechanisms and tool integration. Interlisp [20] and Smalltalk environments like Visual Works [21] and Squeak [22] are good examples. In these environments, the entire workspace state is the representation and different tools can interact with this representation. The entire state of the workspace is typically saved between sessions.

A key advantage is the possibility of supporting various interesting views which we will discuss in greater detail in section 4.2. The main disadvantage is that these environments are typically targeted for a single user and only cover a very narrow spectrum of source artifacts. Another typical design problem is choosing the best-suited representation. We will return to this problem in the next section.

3.2.2 Representation form(s)

In the previous section, we mentioned the distinction between *representation* and *storage mechanism*. Here we will elaborate a bit on this distinction by discussing the difference between *multi-representational* and *uni-representational* environments. In the shared files and repository models, it is easy to distinguish between representation and storage. Tools build their representations by processing stored source artifacts. In the database and the in-memory models, the distinction is less clear. Both models of course store data, but it is stored in a format that is immediately suitable as a representation for tools.

The distinction between multi- and uni-representation becomes relevant when we move from storage medium to a representation. The tools in an environment require a representation that they can examine and manipulate. For instance, a traditional compiler builds an abstract syntax tree from source files in order to perform its operations. When several tools are active within an environment, they can either use a single common or canonical representation, or each tool can have its own representation in some idiosyncratic format.

Multi-representational environments build several different representations in order to satisfy the demands of multiple tools. This approach is common when the storage medium is either shared files or a repository. The main advantage is that it is typically easy to add new tools as they can use their own representation based on the lowest common denominator - the file system. Obvious disadvantages are lack of integration between tools and duplicate effort when building similar representations.

Uni-representational environments focus on the integration of tools on the data-level which will be covered in section 4.1. A central, canonical representation must be chosen and all tools are required to work directly on this representation. The obvious benefits are better support for data consistency, tool coordination that eliminates duplicate effort and the ability to quickly extend the environment with new views. Disadvantages include the problem of choosing a representation format and the difficulties in adding new tools. A canonical representation must be chosen with great care as it forms the foundation on which tools operate. Adding tools can be difficult as they must be adapted to the chosen representation. Interlisp [20] is a classic example of such an uni-representational or *residential* environment. The program resides as data representation which allows extensions like Masterscope to query it and DWIM (*Do What I Mean*) to correct input errors on the fly.

Abstract syntax trees have often been suggested as *best* canonical repre-

sentation in software development environment. This has been implemented in environments such as Cornell Program Synthesizer [23], Gandalf [24], and PECAN [25]. Other researchers, such as Garlan [19] and Meyers [16], have claimed that abstract syntax trees are probably too limiting when an environment has to handle more than simple source code. Reiss [26, 27] has recently gone a step further and proposed that the lack of an accepted, single canonical representation actually suggests that the very concept of uni-representation is wrong. These discussions are still active in the academic tool community.

3.3 Consistency

An important part of the program dimension is the issue of *consistency* which is shown in figure 5. To produce an executable target system, the various sources has to be in a consistent state. Inconsistencies can either lead to the inability to build the final product or perhaps the deployment of an unstable, erroneous product. A software development environment has to take consistency into account. In this section, we will discuss different classes of inconsistencies that source artifacts can contain. We will also discuss some inconsistencies among different source artifacts. Later, in section 4.4, we will continue this discussion by examining how the platform dimension detects, represents, and resolves inconsistencies.

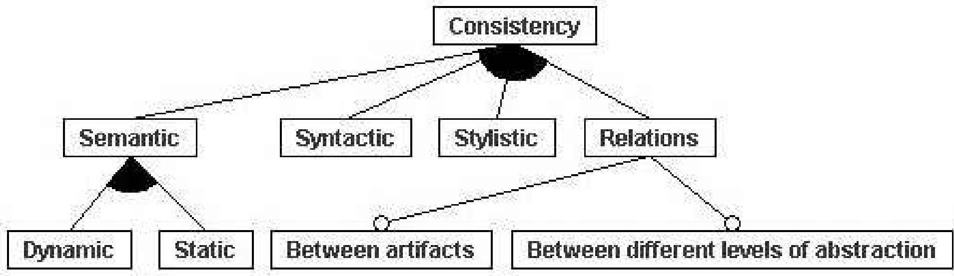


Figure 5: Different classes of consistency

3.3.1 Classes of inconsistency

The first class of inconsistencies are syntax errors. Source artifacts that are written in specification or programming languages are typically subject to a set of grammatical rules. Grammatical rules can for instance be specified in Extended Backus-Naur Form [28] for certain textual languages or in the Unified Modeling Language's Meta Object Facility [29] for certain visual

languages. The rules are typically embedded in parsers, and the inability to parse a source artifact often stems from syntax errors. Syntax errors are inconsistencies between the grammar of the language and the contents of the source artifact.

Syntax errors can be resolved manually and modern-day parsers usually provide detailed assistance in the form of location information and error type. Syntax errors cause compilers or interpreters to fail, but they are rarely of any real significance as they indicate sloppy editing rather than deep design errors. Software development environments often provide different kinds of assistance in order to prevent the occurrence of syntax errors. We will discuss some of these in section 4.2.

The second class of inconsistencies arise when the contents of a source artifact do not follow the stylistic rules and conventions among a set of developers. In minor groups that follow the artist, or craftsman, approach, this might not be a problem. In larger scenarios where an engineering approach is taken, there are good reasons to follow certain standards of writing. Requirement documents can be expected to follow standards that both the customer and developers agree on. Such conventions establish the source artifacts in a common frame of reference which promotes a single, unambiguous interpretation. These conventions can be enforced by reviews.

The style of source code artifacts can also be guided by conventions. Most larger development organization encourage adherence to code standards. This makes code more readable and supports collaboration. Tools such as fxCop [30] can automate the process of reviewing code and enforce an extendable set of style rules. The notion of *Bad Code Smells* [31] which was introduced by Fowler and Beck denotes another category of style errors. *Long Method* is an example of such a smell where an object-oriented program is written in a procedural style. Visualization tools such as CodeCrawler [32] can detect and visualize lack of compliance with conventions, such as *long method*, *large class* or other bad code smells.

A third class of inconsistencies are semantic errors. These errors are caused by lack of consistency between the developer's mental model of the program and the actual meaning of the program as it is embedded in source artifacts. In source artifacts written in natural language, semantic errors occur when the content is written in ambiguous ways. Requirement and analysis documents are typically subject to such errors. The field of formal methods attempt to eliminate these ambiguities by the use of formal specifications such as VDM [33], Z [34] and RAISE [35].

Semantic errors on the code level can be distinguished by whether they concern the static or the dynamic properties of a program. In other words, these errors are detectable either at compile- or runtime-time. In statically

typed languages such as Ada, C++ and Java, semantic errors can arise from inconsistency between the type system in the language and the contents of source code artifacts. Modern software development environments such as Eclipse Java Development Tools [36] and Visual Studio.NET [37] detect these kinds of errors by continually building the source artifacts and running the compiler's type checks. More elaborate type checking and static analysis can be performed by a tool like Findbugs [38] for Java which has more powerful static analysis capabilities than the ordinary Java compiler. The dynamic semantics of a program can only be tested by execution. Several tools, such as debuggers, profilers, and unit test frameworks, attempt to detect the presence of runtime anomalies.

3.3.2 Inconsistency among source artifacts

A large set of inconsistencies concerns relations between source artifacts as opposed to inconsistencies or integrity problems in individual source artifacts. Inconsistencies can arise either between source artifacts at the same level of abstraction or at different levels of abstraction. An example of the first kind is when a set of source code files are inconsistent. Changes may for instance have introduced inconsistencies between source code and configuration files, such as for instance between java source files and deployment descriptors in J2EE applications. In enterprise applications, complex configurations must for instance match large code bases. A given configuration and its corresponding code can get out of synch and lead to errors or non-optimal performance.

Inconsistencies between source artifacts at different levels of abstraction are even more common. In a development project where source artifacts are produced in all phases, it is extremely complex to ensure that source artifacts are not lost or ignored. Analysis and design decisions which are written or sketched informally may be forgotten during implementation such that the final source code is inconsistent with the initial design. These decisions can furthermore be hard to share among members of a team since everyone must agree on an interpretation. Maintenance and reengineering efforts of running systems are often hampered by the lack of source artifacts from the early development phases. Methods and tools have been developed with the sole purpose of recovering design because it is unfeasible to work solely with source code level abstractions. The FAMOOS project [39] is one of the most important research activities in this area .

It is important that a software development environment offers both the possibility of checking for inconsistency in individual source artifacts as well as across several source artifacts. Nuseibeh et al. [40] have demonstrated

that local consistency does not guarentee global consistency, so the problem of inconsistency among different source artifacts should be actively handled by the environment.

3.4 Target product

The target product which results from creating, processing and building source artifacts is typically created with some domain in mind and on a given platform. Figure 6 shows just a few possible domains and platforms which software systems can be created for.

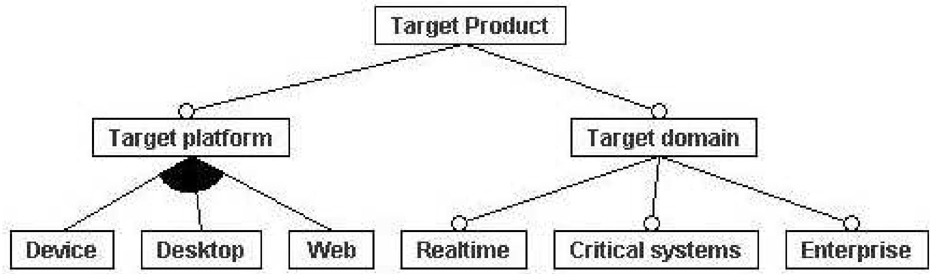


Figure 6: Some sample target domains and platforms

Environments are often evaluated according to how well they support development in a given domain and platform. A software development environment for mobile phones should for instance be able to emulate the target phones and offer special support for programming on a memory-constrained platform. Another example could be software development environments for web applications. Such environments must support the multitude of languages which are part of webprogramming. This requires special tool support as described by Fraternali [41]. These two examples illustrate that the target domain influence our requirements for an environment.

General-purpose environments can often be used for several domains and platforms because they are not tailored for any one domain or platform. Specialization of such environments is typically achieved through extensibility mechanisms where plugins can extend general programming capabilities with features that support domain-specific development. Tools such as Eclipse Java Development Tools [36] and Visual Studio.NET [37] can encompass platforms such as device, desktop, and web when extended with appropriate plugins. General-purpose environments should be designed such that the support offered for each domain is not the least common denominator.

Critical systems, such as those in the defence, medical or traffic sector, often have zero tolerance for errors. To produce applications for these domains, rigorous formal methods are typically applied. Software development environments such as Overture [42] are based on formal methods which guarantee a very high-level of correctness. The software development environment should have explicit method support as described in section 6.2 in order to offer real value in this target domain.

4 Platform

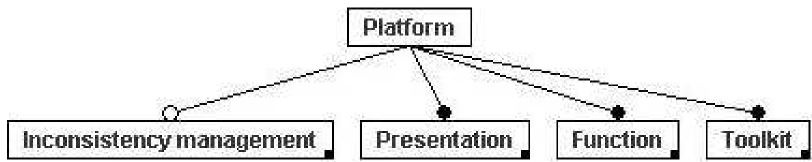


Figure 7: Facets of the *platform* dimension

The *platform* dimension of an environment is the second of our four Ps. Where the program dimension concerns the material that we are giving a form, the platform dimension is the foundation on which our work takes place. It is metaphorically our workbench and tools. This dimension is often described as if it was the complete environment. This mistake is caused by the fact that the platform dimension mainly consists of the *tools* and the *presentation* aspect. These are tangible and visible aspects of the environment and they form the foundation on which development takes place. In this section, we will examine the platform dimension, as shown in figure 7, by looking at the tools, their presentation or graphical interface, the environment's primary function and its ability to manage inconsistency during development.

4.1 Toolkit

The *toolkit* and its organization is one of the most important parts of an environment. Definitions and categorizations of tools can be found in surveys by Reiss [43] who solely looks at programing tools and Grundy and Hoskings [44] who include a wider range of tools. In this section, we will cover two general aspects of tools: *integration* and *openness* which are shown in figure 8.

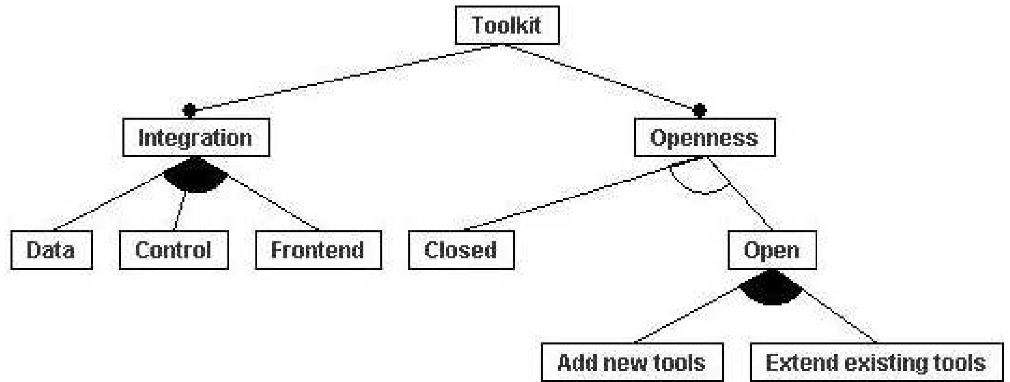


Figure 8: Integration and openness features in the *toolkit*

4.1.1 Integration

In early environments such as the Programmer's Workbench [10], the environment simply consist of a set of loosely bundled tools. Later developments have led to integrated environments where the integration between tools is a major concern. Integration can - according to Wasserman [45] - take place at three different levels. First, the underlying data and program representation can be integrated. Second, the tools can be coordinated by messaging. Third, the front end or graphical user interface can be integrated.

Data integration has already been covered in section 3.2 where the problems and strengths of different storage and representation forms were discussed. This kind of integration belongs to the program dimension of environments because the glue that binds tools together is the program itself.

Control integration is achieved by enabling the individual tools to communicate. Messaging either happens directly between the tools or through a central broadcasting mechanism. The first approach requires tools to be able to communicate with all other tools in their environment through idiosyncratic protocols. The latter approach taken by FIELD [46] where tools inform each other by selective broadcasting in a so-called *hub'n'spoke* architecture. The broadcast mechanism is by far the easiest to implement since each tool only has to understand the interface to the central hub. Tools can subscribe to and react on events in the environment. This can lead to better and more responsive environments since the user does not have to initiate all operations. The main disadvantage is that tools must be adapted to communicate. If new tools are added to the environment then a range of changes might cascade through the system.

Front end integration is achieved by providing a uniform interface to

the tools of an environment. This approach is taken in most integrated programming environments because it provides a single use context for the developer. A uniform interface can also help the developer by showing the relationships between different tools. A build sequence can for instance be created by composing tools like parsers, compilers, linkers etc on a single screen. The central trade-off with regard to front end integration is that one can either choose to show the full potential of a few tools or a limited part of the features of many tools within a single screen. If the environment provides a detailed view into the capabilities of a single tool then the relationships between tools become less visible. If the environment favors showing a large number of tools then they can only exhibit a limited part of their functionality. This may lead the developer to use only a subset of the inherent functionality in the tool.

One elegant solution to the above problem is to provide multiple compositions of tool features, such that the developer can switch between different views on the tools depending on his current task. This approach is taken in Eclipse Java Development Tools [36] where the notion of *perspectives* enables the developer to switch between programming, debugging or testing mode. Each mode displays different features of the same underlying tools in order to support the task at hand.

The three levels of integration can of course be combined in various ways. Data, control and front end integration are all present in some environments and only partially present in others. It should be noted that there also different degrees of integration. In a control integrated environment, the degree of integration depends on how large a percentage of the tools that are actually aware of each others' presence and their ability to communicate semantically rich messages. These issues are discussed in greater length in the Object-Oriented Tool Integration Services approach [47] where a distinction between coarse-grained and fine-grained integration is introduced. Another excellent source of information is Brown and Penedo's annotated bibliography of articles in this field [48].

4.1.2 Openness

The *openness* aspect of an environment describes how well it can be extended with new tools, views or different features. It is often necessary to extend an environment. The environment itself can evolve and hence need to be updated. The individual tools can evolve or new tools can be added. The environment should be capable of incorporating to such changes and if necessary adapt to new features.

We distinguish between three slightly overlapping models of openness.

First, closed environments are unable of being extended. They are often tightly integrated and can at most be subject to updates. These environments are most frequently found in proprietary and very domain-specific niches where a single vendor rules the market. The advantage is that the complete architecture is typically designed for optimal cooperation between tools. The disadvantage is the lack of adaptability and flexibility.

The second and third possible openness models are respectively the ability to extend existing tools and to add new ones. These two models are often overlapping since new tools can for instance extend several existing ones in order to provide new functionality. These openness models are especially relevant to general-purpose environments where the environment has to adapt to new tools on the market and more specialized demands from the developers. Implementations of these openness models are Emacs [11, 12] or Interlist [20] where everything can be rewritten in various LISP dialects, and Eclipse Plug-in Development Environment [49] where the entire environment is built around the concept of plug ability. The Eclipse Plug-in Development Environment is especially interesting in this context because it actually offers both a runtime in which to test tool extensions as well as a detailed set of guidelines [50] on how the environment should be extended.

4.2 Presentation

In section 3.2, we covered how an environment can store and represent a program. In order to actually make the underlying representation visible and tangible to the developer, the environment needs to present this representation. Presentation denotes both the passive viewing of the program as well as the interactive editing of the program. In this section we will describe a few of the important facets of presentation, as shown in figure 9, in software development environments. Meyers [16] distinguishes between environments where different views interact or are dependent on each other and environments where views can relate to entirely different parts of the program and hence be independent of each other. We will follow this distinction by distinguishing between *uniparadigm* and *multiparadigm* views. Uniparadigm views are typically related solely to the coding phase whereas multiparadigm views span several development phases. We will furthermore examine some different editing modes that various views provide.

4.2.1 Uniparadigm views

In early environments where the level of integration was very low, each tool typically built an idiosyncratic representation which supported a single

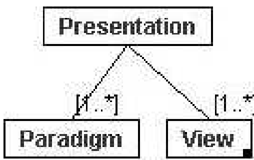


Figure 9: Presentation is ordered in paradigms and views.

view. A simple text editor for instance offers a textual view into the representation of a text file. As integrated environments began to appear, more advanced views such as layered or multiple views became available. Multiple representations can support layered views which each add some sophistication to the bottom layer. An example of a layered view is when our simple text editor is extended with syntax highlighting. In this case the bottom layer is the text file view where the representation is a sequence of characters. A parser can then parse this representation according to some grammar and add a view layer which emphasizes keywords by displaying them in a special font. Such layered views in the programming phase aids the developer in getting an overview and detecting syntax and style errors.

A more advanced use of the underlying representation is multiple views which are very common in modern-day programming environments. If the environment has built a sophisticated representation of a program, then it is possible to support different views which all display some special properties of the underlying representation. An abstract syntax tree of a Java program can for instance support views such as a syntax highlighted source code view, a list of method signatures, a call graph and other views displaying various kinds of interdependencies. Views can also relate bookmarks, tasks and similar metadata to locations in the source code. In complex programs, such views can make all the difference for a developer trying to comprehend the program at hand. One especially used feature is navigation by hyperlinks where a developer can click on a method call a be moved to the location of the definition of this method.

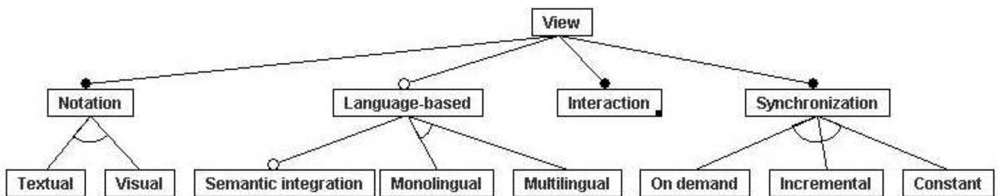


Figure 10: View features.

Views can be textual, visual or even hybrids as shown in figure 10. Textual

notation are used for requirements, specifications, and source code listings. Programming tools traditionally favors textual views. Visual notations are suitable for use cases, early diagram sketches, flows and various other high-level views on the representation. Hybrid forms can be found for instance in the Intentional Programming Environment [51, ch.11] where for instance a IF-ELSE construction within a source code listing can be displayed visually as a branch.

When multiple views are in use, it is necessary to consider synchronization and consistency issues. Read-only views must be updated regularly and read/write-view must be capable of both updating the representation and being updated. Each view is synchronized with the underlying representation. In multirepresentational environments, the synchronization between views is inherently delayed as a view's underlying representation must be stored before a change can propagate to other views.

The synchronization procedure can happen either on demand, incrementally or constant as shown in figure 10. On demand synchronization happens when the developer initiates the operation. Incremental synchronization is initiated by the environment either in fixed intervals or in an event-based manner where certain actions trigger synchronizations, such as in the SMILE environment [52]. Instantaneous synchronization is possible in some unirepresentational environments where views interact directly with the underlying representations.

4.2.2 Edit modes

Read/write views provide different kinds of editing modes as shown in figure 11. Especially views based on a single representation support more advanced forms of editing. In textual views, we distinguish between two main kinds of editing: free and structured. These editing modes can furthermore be augmented with automatic support of various kinds. Graphical views are usually structured.

A textual view which provide free editing is the simplest form. Any text inputs can be entered and changes do not propagate until the contents of the view are saved either by user action or incremental building. Free editing is preferred among most developers as there is a complete sense of control. Most source artifacts from the early phases of development are edited in completely free form. Source code artifacts can be edited in free mode but with support in the form of automatic code completions and on demand insertion of code templates.

Structured editing of textual views is a bit more advanced. Environments with frequent incremental builds or instantaneous synchronization use the

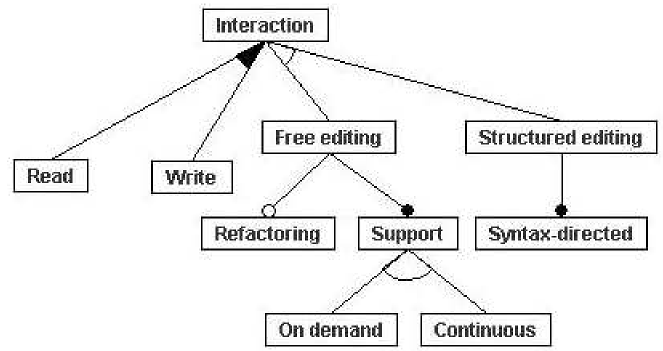


Figure 11: Edit modes and interaction features of views.

underlying representation to guide the developer in the editing activity. In language-based environments where the representation is tailored to a specific language, structured views enforce syntactically correct programs. The Emily editor [53], the Cornell Program Synthesizer [23] and Gandalf [24] all use a grammar-based representation to direct the developer. Such systems are also known as syntax-directed editors.

The main advantage of structure-oriented and syntax-directed editors is the enforcement of syntactical and stylistic correctness. The program appears to the developer as a kind of template where a set of placeholders can be filled in with a limited set of possible completions. From early research projects such as the Cornell Program Synthesizer [23] and Gandalf [24] to modern-day commercial prototypes such as JetBrains' Meta Programming System [54], a central goal has been to create editor-generators such that a syntax-directed editor can be generated for arbitrary languages.

Another frequently mentioned advantage is speed of development. If the developer only has to fill in templates then a lot of the manual typing is eliminated and hence development speed is higher. This has been questioned by actual programming practice because there are situations where tolerance of temporary syntactical inconsistency can actually speed things up. A central critique of the initial versions of the Cornell Program Synthesizer was that some editing operations were complicated exactly because of the enforced syntactical correctness. If the programmer could manually override these editing constraint, he could achieve his goals faster.

Nørmark [2] discusses a related problem which he calls *the feeling of claustrophobia*. The programmer feels a lack of control and editing impotence. This is especially bad in situations where editing operations are complicated solely by the editor constraints. Even JetBrains' recent prototype the Meta Programming System [54] does not seem to offer a

solution to these old problems. It is tempting to conclude that syntax-directed editing should be dismissed. One possible exception could be programming for non-programmers where correctness can be prioritized above the developer's need for a feeling of control [55].

A support feature which deserves to be mentioned is wizard-based editing which relies on code generation schemes to speed up development. This is in effect an advanced form of on demand support. The editing process typically starts working through a flow of wizards where all input is verified. The wizards generate code skeletons which can then be extended by either free or structured editing in ordinary editors. The wizard-based approach is especially powerful in situations where complex configuration has to be specified or where high-level constructs require lots of *boiler-plate* code. A modern programming environment like Visual Studio .NET [37] uses wizards to generate boiler-plate code for graphical user interface, database access and application configuration.

A special case of editing operations which are typically provided by wizard-based approaches are machine support for refactoring [56]. A machine supported refactoring is a re-structuring of code which preserves the program semantics [57, 31]. Wizards are useful in relation to refactorings as they enable a certain parameterization and fine-tuning of the general refactorings. Refactorings can be considered high-level editing operations and they are a standard requirement for programming environments today. A rather recent suggestion for future editing modes is Example Centric Programming [58] and the Subtext prototype [59]. This form of editing use the copy and paste operations of free editing as first class entities in a structured environment. This editing mode is similar to refactoring because all editing happens through transformations of the program source artifact.

4.2.3 Multiparadigm views

The uniparadigm views and editing operations that we covered in the previous sections are typically related to the programming phase. The complexity of software development projects often require visualizations that span the individual project phases. Multiparadigm views provide these broad perspectives. The different views do not interact with each other as they embody different concerns, and editing often takes place in different languages. In a multiparadigm environment, the domain expert will use another language and another set of views than the programmer. These different views are all integrated in the final product.

An early system which supports multiparadigm views is the Draco system by Neighbors [13, 14]. The Draco system introduces a set of domains

which each has its own notation. Each domain concerns a specific part of the system, and these domains can be composed to produce the final system. Reiss' so-called conceptual programming environment Garden [60] follow similar ideas by letting the user specify new notations to fit the various required paradigms. The wave of Meta CASE tools, which we will describe in section 4.3.4, shows renewed interest in environments based on multiparadigm views.

4.3 Function

The platform dimension of a software development environment can typically be described to some extent by stating what its *function* is. Until someone succeeds in delivering the universal software development environment, we will have to settle for environments where the platform is specialized toward one or a few goals. In this section, we will enumerate some of these possible goals as shown in figure 12. The list will necessarily be limited. A more detailed discussion of possible primary functions can be found in the survey by Grundy and Hoskings [44] where several concrete examples are listed.

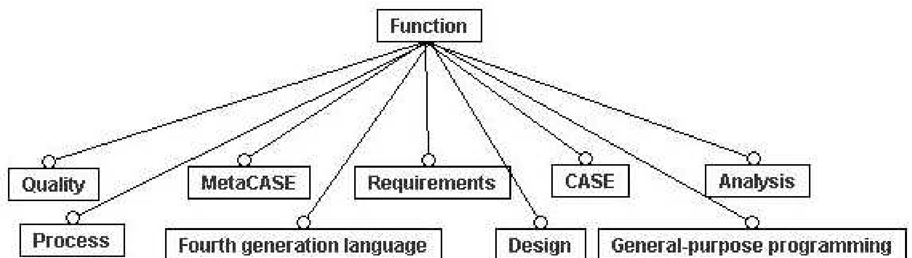


Figure 12: Supported functions.

4.3.1 General-purpose programming

Environments that are primarily oriented toward supporting the programming phase can usually be ordered into two general categories: light- and heavyweight environments. The lightweight environments are relatively simple tools organized according to the needs of an artist or craftsman approach as described in section 2.1. The heavy-weight environments follow the engineering approach and are typically called Integrated Development Environments (IDEs). These environments offer a broader range of programming support features.

Probably the best and classical example of a lightweight environment is Vi [61]. Vi supports the basic file-based, textual approach which replaced

the *type writer editor* paradigm of the early days of programming. Vi is a paradigmatic example of a lightweight environment where the creative, unconstrained mode of working is in the center. Heavyweight environments such as Eclipse Java Development Tools [36], IDEA [62] and Visual Studio .NET [37] have gradually taken the center of the stage in recent years. Due to the exponential growth in computational power, it is now possible to offer multiple different views in addition to the editors of lightweight environments. Heavyweight environments typically also offer configuration management facilities which will be described in section 6.3 and support for monitoring the quality and integrity of the program.

4.3.2 Fourth Generation Languages

The term *fourth generation language* (4GL) was coined by James Martin in 1982 [63]. These languages raise the level of abstraction to a higher degree than ordinary, block-structured programming languages. Fourth generation languages are typically embedded in special environments that leverage the full strengths of these languages. Apart from residing in specially tailored environments they also typically imply a incremental development method where simple components are gradually modified and refined [64].

Typical usages of fourth generation languages are database querying, report and forms generation. These areas are characterized by being very domain-specific and requiring tool-specific training. Classical examples of fourth generation languages can be found in Horowitz et al.'s survey of application generators [64]. More modern examples are Oracle Reports [65], ABAP [66] and C/AL [67] for financial report generation, Microsoft Access [68] for database management, SAS [69] for data mining and Visual Basic [70] for user interfaces.

The success of a fourth generation language depends on how well the development problem fits within the anticipated domain and implied method. The main disadvantage of these tools is usually the lack of flexibility caused by a strict focus on a specific kind of problems. fourth generation languages typically only allow a limited range of customizations and changes beyond this often lead to inelegant solutions.

4.3.3 Computer-Aided Software Engineering (CASE)

Computer-Aided Software Engineering (CASE) tools are to a large extent the realization of the engineering work mode as described in section 2.1. CASE tools span several project phases and supports the management of multiple kinds of source artifacts. A central motivation for CASE tools was the rise of

development methods where some steps could be automated. The high-level design of entity and behavior diagrams could be used to generate skeleton code which could be customized in subsequent steps [71, 72].

Another factor distinguishes CASE tools from the previously described IDEs and fourth generation languages is the central role of collaboration. The engineering work mode relies on the successful collaboration of multiple people, so CASE tools are used to support this aspect. Methods and process models are often an explicit part of a CASE environment. This enables managers to perform more accurate estimates and programmers to get a better overview of the program architecture.

CASE tools have not been adopted at the rate that one might expect from the impressive lists of technical features. Reports have shown adoption of CASE tools is slow and that the tools are often not used to their full extent after introduction [73]. Jarzabek and Huang [74] have suggested that the main reason for this lack of adoption is that CASE tools only supports *hard* aspects of software development. The tools are typically not suited to a flexible process and are incapable of adopting the programmer's mental model of the project.

4.3.4 Meta CASE

Meta CASE or domain-specific modeling environments have emerged during the last 10 years as a form of higher-order CASE tools. Meta CASE environments are tool-builders which produce CASE tools. These environments provide a generic set of possibilities which can be used to compose more domain-specific environments. It might be argued that the term meta-fourth generation languages is equally appropriate since the produced environments often have capabilities similar to classic fourth generation languages. The output of meta CASE tools can be tailored to an individual company's concepts and methods which makes them very attractive.

The users of meta CASE tools can be split into two categories: tool adapters and tool users. The tool adapters adapt use the meta CASE tool to build specialized CASE tools. The tool users use the specialized CASE tools, i.e., the adapted meta CASE tool. A central problem is that the meta CASE tools require highly skilled developers in order to be adapted properly. It is no easy task to create environments that are both specific and flexible from a generic toolbox.

Tools such as MetaEdit+ [75], Generic Modeling Environment [76], XMF-MOSAIC [77], and Visual Studio DSL Tools [78] are examples of meta CASE environments. These tools all promise higher productivity and greater conceptual precision due to their ability to produce highly specific

environments. The technology is still too young to be evaluated properly on empirical basis. One possible critique that could be leveraged is that these tools suffer from the same problems as fourth generation languages and CASE tools, and they will therefore not be able to replace traditional IDEs. Fowler [55], who has coined the term *language workbenches* for these tools, argue that they are not intended to replace IDEs but that they are instead aimed at non-programmers. This suggests a two-staged development method where tool adapters in the first stage develop domain-specific environments and tool users in a second stage use these environments to develop actual applications.

4.4 Inconsistency Management

An important part of the program dimension concerns the state of the program or specifically its degree of consistency. In section 3.3, we described the various kinds of inconsistencies that can be present in a program under development. In this section, we will examine how the platform can manage such inconsistencies by providing certain services to the developer. We will discuss the detection of, representation of, reasoning about, presentation of, and resolution of inconsistencies in a program as shown in figure 13. This discussion is inspired by the work of Grundy et al. [79] on inconsistency management in multi-view software development environments.

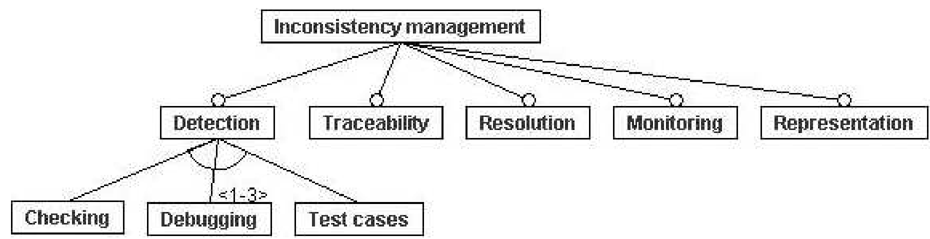


Figure 13: Inconsistency management features

In order for the developer to realize the existence of inconsistencies in the program, the platform must offer means of detecting such inconsistencies. The detection process must be automated to some degree if it is to be implemented in the platform. Requirement documents can for instance rarely be automatically checked as they demand human reviews. Source artifacts that appear in later development phases often lend themselves easier to automatic checking.

Syntactical, stylistic and static semantic errors can be checked by parsers, type checkers and similar tools. Dynamic semantical errors can to some extent be detected by debuggers and test cases. Consistency relations

between source artifacts at the same level of abstraction are usually also checked by the mentioned tools. Consistency relations between source artifacts at different levels of abstraction typically requires specialized tools, such as the CLIME environment [27].

When an inconsistency has been detected, the environment must be able to represent it in some way. Most consistency checkers usually run through the program representation in a single pass and the environment must capture inconsistencies in a new representation in order to be able to manage them. If the inconsistency representation is only kept in-memory then it becomes hard to understand especially the maintenance of systems. Previous inconsistencies might have been repaired in an unsound manner. Such repair operations must be stored along with their causes if one is to understand the maintenance history of a source artifact. Practically no environments provide this kind of historical information on inconsistencies.

Automatic reasoning about inconsistencies can provide advanced traceability support in an environment. If we are to understand why a program transitioned into an inconsistent state and how/if this inconsistency was resolved, we need traceability. Several researchers have worked on the problem of reasoning on the basis of inconsistent programs. Inconsistency in classical logic allows us to conclude anything on the basis of inconsistent premises. Inconsistency in software does not lead to a situation which is quite as extreme. Nevertheless, when the logical structure of our programs is inconsistent then our ability to logically infer program properties is reduced.

Research [80, 81] have suggested that our environments must be able to tolerate some degree of inconsistency in order to effectively support the development process. A series of editing operation can for instance introduce inconsistency temporarily. Environments which rigidly prevent temporary syntactic inconsistency, such as the Cornell Program Synthesizer [23], are often a cause of frustration to developers.

Nuseibeh et al. [40] describes the case of the Ariane-4 rocket where an inconsistency between the safety requirements for exception handling and the lack of floating-point exception handling in the implementation was tolerated because it was concluded that floating-point overflow could never occur. This story has a tragic twist since the Ariane-5 rocket reused the implementation of Ariane-4 but the floating-point exception handling inconsistency was not re-evaluated and the flight of Ariane-5 ended in disaster. The morale of this story is that tolerance of inconsistency is a valid option in software development environments but only if the developer is made aware of this tolerance by the environment.

The presentation of inconsistency has been a very low priority in tool- and environment development. The most frequent presentations of

inconsistencies are compiler warnings and errors about source code in modern IDEs. These environments provide lists with descriptions of the problem and easy access to the location of the error. The step-based replay-feature of debuggers provide a presentation of runtime errors but these tools often require an understanding of the execution platform and data representation in order to be effectively used. The least supported presentation form is the presentation of inconsistencies across different levels of abstraction. These kinds of inconsistencies are complex because they are caused by the interplay of several source artifacts at different levels of abstraction and it is often not possible to pinpoint any single root cause of the problem.

Finally, the resolution of inconsistencies is also a matter of concern in environments. To resolve an inconsistency, the environments must be able to trace its origin. It is not always possible to trace a single root cause, and even if such a root cause can be found, there can be several possible resolution strategies. In short, human intervention is usually required. The automatic elimination of inconsistencies can also be dangerous because the developers may lose control if the environment is too aggressive in automatic inconsistency resolution. The optimal solution seems to be that the environment presents detected inconsistencies and possible resolution strategies, but leaves it to the developer to actually execute one of these strategies.

5 People

Nygaard has stated that *We need to develop systems in which the computer based components only are subsystems, even if important ones* [82]. This quote leads us to the *people* dimension which is the third of our four Ps as shown in figure 14. This dimension often plays a minor role in the design of software development environments. A classic misconception is that if we design our tools with the best possible technical design then this will necessarily lead to adoption and successful usage. Experience with CASE tools has however [73, 74] shown that even superb technical and highly innovative solutions do not stand a chance if they do not take users into account. This section will introduce four of the most important parts of the people dimension which are *area of expertise*, *level of expertise*, *organization* and *mediation*.

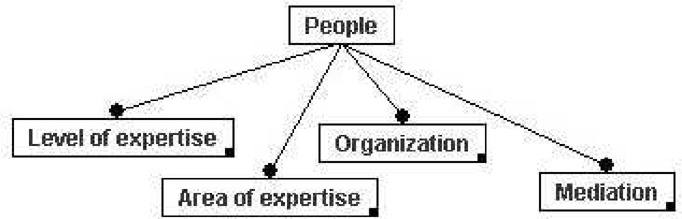


Figure 14: The *People* dimension.

5.1 Area of Expertise

The involved parties in a development project can have different *areas of expertise* which determine their responsibilities. We have chosen a table representation, as shown in table 2, instead of a feature diagram to make room for descriptions of each role. Typical roles are managers who oversee the project, analysts who describe the problem domain and requirements, designers and developers who implement the program and testers who perform activities like integration, user and acceptance tests. Each user group has different needs and tasks. A software development environment should support as broad a range of these roles as possible in order to integrate the diverse efforts of many different people. Especially when work is structured according to our metaphor of the engineer, it is necessary to divide the work into domain-specific tasks which can be handled by people with the relevant expertise.

The Draco system [13] distinguishes between roles such as *system builders*, *domain builders*, *domain users* and *system specialist* which all play a different role in the Draco method of developing software. Each of these roles view the system in a different way. The domain builder describes some business domain in a custom notation. The system specialist compose and refine general domain descriptions into an executable system using a different view. Systems such as Garden [60] and meta CASE tools, section 4.3.4, takes these ideas further by providing advanced visual or textual tools to define domain-specific languages that suit a particular role.

Another aspect of how software development environments adapt to specific roles can be seen in a tool such as Visual Studio 2005 Team System [83] where the environment is composed of a general platform and a set of role-specific extensions. These extensions are tailored to suit the needs of a *manager*, an *architect*, a *tester*, and a *developer*. Similarly, the Rational suite [84] targets roles such as *analyst*, *architect*, *developer*, *tester*, and *deployment manager*. These products are tailored for specific roles but share a common underlying platform.

Role	Area of expertise
Analyst	Scoping requirements and business concepts in cooperation with the customer.
Designer	Mapping of requirements and business concepts to implementation.
Developer	Actual coding, implementation and low-level design choices.
Tester	Unit, integration, usability and acceptance tests.
Deployment manager	Deployment, configuration and maintenance issues.
Project manager	Management of project progress and plan.

Table 2: Areas of expertise may be associated to roles.

5.2 Level of Expertise

Apart from having different areas of expertise, developers often also master their skill with different *levels of expertise*. Some programmers are for instance known among the team members to be real *hardcore* problem solvers while others are inexperienced and not yet ready for a major responsibility. The user interfaces of software development environments should take these different levels of expertise into account, such that novices can be guided by hints, balloon tips and other forms of assistance while experts can cut straight through to the core of the problem without unnecessary distractions.

Kelleher and Pausch [85] have created a useful taxonomy of programming environments for novice programmers. Their taxonomy shows several different ways a software development environment can support novices, such as changing the edit modes, introducing graphical views and/or using different languages (visual or textual). Generally, these different approaches mainly focus on the presentation facet of the software development environment. In other words, a well-designed user interface is a key factor in supporting developers with weaker skills.

Support for skilled programmers must take our work metaphors into account. Developers working according to the artist or craftsman metaphor will typically prefer a minimal interface with a free editing form. Developers in the engineering tradition will need more powerful views. A central concern here should be that the user interface and toolkit must be adapted

to the individual's requirements. An environment such as Eclipse Java Development Tools [36] offer the combination of plugin architecture for tools and customizable *perspectives* in the user interface. Such features support the demanding developer as they allow him to adapt the environment to meets his current needs.

5.3 Organization

The *organization* of developers becomes important as soon as their number rises above one. If there is only a single developer who plays all roles in the project then the environment should be lightweight and suited to his needs. If several developers are involved then organizational issues such as roles, location and responsibility must be taken into account. The organization of the developers should ideally be explicitly represented and supported in environments.

In large projects, the development parties are typically not co-located. Sometimes the group is even distributed geographically and support for collaboration is therefore needed. Software development environments provide this kind of support through source artifact repositories, messaging systems, and web-based knowledge sharing facilities such as portals, newsgroups and wikis. It is a great strength if an environment is able to capture the messaging traffic in such a way that for instance contractual information and design discussions are not lost.

The communication between different parts of the organization often happens in natural language. An interesting approach to enhance the value of threads of emails would be to formalize and integrate them into software development environment. If an email discussion about a bug for instance can be traced back to source artifacts such as source code and design diagrams then we can increase transparency between developers and end-users. Such traceability features requires strong communication infrastructure in the environment.

5.4 Mediation

The fourth facet of the *people* dimension is *mediation* which deals with the software development environment's usability, i.e., the system's ability to support the user in his tasks. The user interface is typically the focal point of attention when discussing mediation. User interface widgets, screen and workflows must be intuitive and support the user in a straight forward manner. Apart from usability issues, mediation also concerns the software development environment's ability to effectively establish correct mental

models of the program that is being developed. This requires a clear presentation design, as described in section 4.2, and an adaptability to the skills of the individual user, as described in section 5.2. We will not discuss the mediation facet further in this paper as the topic of usability alone is an entire research that is beyond our scope.

6 Process

The inherent complexity in software development is due to the multitude of source artifacts, tools, and people that interact in various ways. A software development environment encompasses these three dimensions as described in the previous sections. The fourth and last dimension of an environment, according to our taxonomy, is the *process* dimension as shown in figure 15. The process dimension captures the principles that guide and manage the complex interactions of the previously described dimensions. In this section, we will examine how the process dimension is structured and what role it plays within an environment. We will look at four central facets of this dimension: the project *life cycle*, the guiding *method*, *configuration management* practices and the cognitive *decomposition techniques* that developers apply.

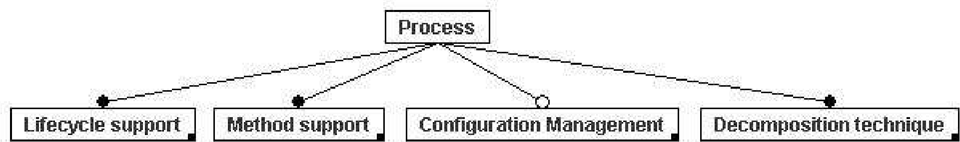


Figure 15: The *Process* dimension.

6.1 Life Cycle

Regardless of the guiding method, it is often useful to consider a development project as a set of connected phases. We have, in figure 16, not chosen a feature diagram in order to show the sequential relation between the different phases. The project may not progress in this manner, but the waterfall conception of a project forces one to take all the different phases into consideration. Typical phases are the requirements phase, analysis and design phase, implementation phase, testing phase, deployment phase and maintenance phase. Software development environments can be evaluated in terms of their *life cycle* support, i.e., their coverage of the different phases.

Those that cover more than one phase should also be evaluated on how well they integrate the efforts and source artifacts of the different phases.

Figure 16: A classic waterfall representation of the project lifecycle.

The requirements phase produces the documentation of and agreement on initial requirements for the final system. The participants are usually customers, managers and analysts. These requirements are typically specified in natural language, possibly guided by specification templates. Few environments actually support this phase even though the requirements serve as central evaluation criteria for the final product. The main problem is that it is difficult to relate natural language requirements to design diagrams and source code because of the different levels of formalization. On the other hand, formalizing requirements at too early a stage might be counterproductive as customers may be unfamiliar with formal notations.

The analysis and design phase concerns the documentation of general concepts from the problem area and the mapping of those into the solution area. Various notations are possible ranging from strict formal methods to loosely defined sketches. The common factor among these notations is that some sort of formalization is taking place. Requirements articulated in natural language are translated to a more formalized notation in order for programmers to be able to start working. Classical software development environments do not support this phase very well, but with the advent of CASE tools this situation has, in principle, changed. In practice, however, the CASE tools have not been adopted as described in section 4.3.3.

The implementation phase concerns the programming and purely technical sides of development. This phase has been and continues to be the central area for most software development environments. Programmers must be supported while tackling the challenging activity of translating design to executable code. As programs are being developed, the complexity of a project can quickly become overwhelming, so developers need multiple views on different levels of abstraction in order to properly conceptualize the system. The main goals in this phase are getting the system implemented correctly and on time as well as ensuring that the finished product is in accordance with the requirements and the design. The main problem is often that the requirements and design are either ambiguous or not well understood. Hence it is not possible to properly align the implementation with these requirements and the design.

The testing and deployment phases concern the final approval and delivery of the program. Testing ensures, on a technical level, that the program meets the requirements and, on the human level, that users are actually capable of using it and benefit from using it. Deployment of the program concerns the final build and integrity checks as well as installation at the site of customers. The degree of support for these phases vary a lot in current software development environments. Some have rigorous testing schemes built in while others simply assume that an executable or compilable program is a correct program. The ability to test a program rigorously often depends on the method that has been employed while constructing the program. Formal methods offer mathematical proof techniques whereas *ad hoc* methods can only offer rather arbitrary tests.

The maintenance phase concerns work on the program after deployment. This includes creating patches and bug fixes as well as managing the different versions and releases. Customer feedback, design modifications and adaptation to individual customers can also be part of this phase. Software development environments with strong configuration management support are very valuable in this phase. Feedback and change requests are often harder to handle. The software development environment must be able to trace a user complaint to a design decision or faulty requirement. Several tools offer advanced traceability features, such as Bugzilla [86] which is widely adopted in the open source community.

6.2 Method

The *method* of a project denotes the overall guiding principles. A project can be guided in a purely ad hoc manner, in an agile and/or iterative manner, by the waterfall process or even by purely formal means as shown in figure 17. The method determines the order of different phases and how these phases are executed. In this section, we will briefly discuss how software development environments can offer method support.

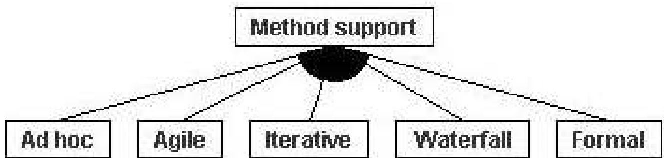


Figure 17: Different type of method support

We distinguish between three degrees of method support in a software development environment. First, some environments are completely agnostic

with regard to the method employed. This is very common in general-purpose environments as described in section 4.3.1. Environments like Emacs [11, 12], Eclipse Java Development Tools [36] and Visual Studio.NET [37] do not offer any explicit method support. This does not mean that they can not be used in conjunction with some particular method. The main point is that the method employed is not enforced by the environment.

Second, software development environments at the other end of the spectrum are customized to a specific method. Such environments are typically implemented in order to promote the adoption of a certain method. This can be achieved by making methodological concepts and notations explicit and enforcing a certain process in the environment. An example of such an environment is the Overture tool [42] which is a dedicated tool for an object-oriented variant of the Vienna Development Method called VDM++. Classic CASE tools, as described in section 4.3.3, are also often tailored to a specific method.

Third, some hybrid software development environments exist where the environment can be parameterized with different methods. Meta CASE environments as described in section 4.3.4 often fall into this category. These higher-order environments generate method specific environments. A tool such as MetaEdit+ [75] is exactly designed to support different methods. These environments are different from our first category as they are not agnostic of the method. A generated environment embodies a given method and has explicit support for this method, for instance by using a given notation.

6.3 Configuration Management

An important part of any software development environment is its ability to manage configuration items - the support for *configuration management*. All the various source artifacts of a project can be considered configuration items. These items are composed into configurations that capture valid relations between source artifacts. As a project progresses, the individual source artifacts can be revised several times and the different versions of the same source artifact may take part in different configurations. Configuration management, which is illustrated in figure 18, handles the problem of keeping track of these items and configurations.

Versioning of source artifacts can be handled manually but software development environments typically automate this process by using *version control tools*. Several versioning paradigms exists [87]. A versioning paradigm takes its outset in the conceptual definition of a configuration item. Source artifact-oriented systems such as CVS [17] store versions of single source

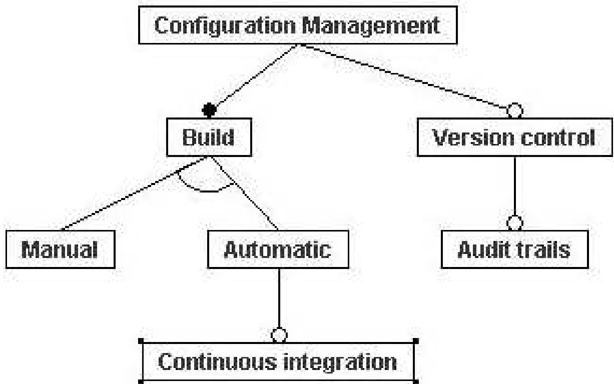


Figure 18: Different kinds of configuration management support

artifacts and use a layer of metadata to retrieve configurations. Change-oriented systems, such as Arch [88], store change-sets and apply these to retrieve configurations. Most version control systems are based on storage models such as repositories or central databases as described in section 3.2.1.

Apart from storing the different versions of source artifacts, version control systems are also important from a managerial perspective. The version histories provide information about the progress of a project and audit trails can be used to assign blame when the source repository is corrupted. Configurations and configuration items are typically annotated and this information enhances the possibility of offering traceability. A bug or a change request can for instance be traced back to a certain revision of a source artifact.

Other parts of configuration management are *build automation* and *continuous integration*. Build automation concerns the process of transforming source artifacts to the target product. Automation tools such as Make [89] and Ant [90] allow the developer to formalize the process of building a product. The build process can usually be parameterized such that different targets can be produced from the same set of sources. One can for instance produce a build either with or without test code.

Continuous integration, a term coined by Fowler and Foemmel [91], is a synthesis of version control, build automation and testing. The idea is to extend the software development environment with a separate server which automatically retrieves the latest code from the repository, builds it according to the formalized build process and runs tests on the target product. This practice ensures frequent integration tests and regular feedback to developers and managers of the state of the code base. Continuous integration is implemented in tools such as CruiseControl [92] and Draco.NET [93].

6.4 Decomposition Techniques

The understanding of a complex problem requires certain analytical skills. The classic way of understanding such a problem is to decompose it into smaller more comprehensible parts. Software development environments support this process by embodying different *decomposition concepts* as shown in figure 19. Decomposition is often prescribed in a certain manner by the employed method. If a software development environment is tied to a certain method as described in section 6.2 then this usually implies a special decomposition technique. Object-oriented methods as described by Wirfs-Brock and Johnson [94] are for instance centered around decomposition in objects and classes. Some of the most common techniques use the following concepts during the decomposition process:

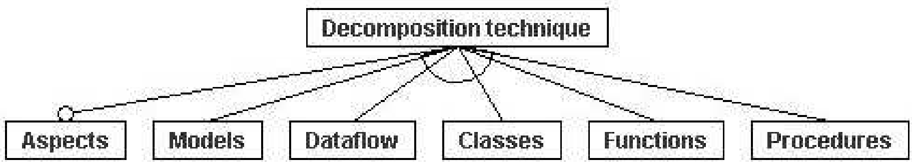


Figure 19: Different decomposition concepts

It is beyond the scope of this paper to describe the different decomposition techniques in detail. The key point in relation to software development environments is that decomposition techniques can, similarly to methods, be integrated to various degrees. Some environments are agnostic of decomposition whereas others are tied to a specific decomposition. A recent trend is to allow various different decomposition techniques simultanuously. The Concern Manipulation Environment [95] follows this idea discarding the tradition of having only one decomposition technique - a tradition which has been called *the tyranny of the dominant decomposition* [96, 97].

7 Applying the taxanomy

In this section, we will evaluate a couple of software development environments with respect to the previously introduced taxonomy. As there is an astronomical number of different environments on the market from industry, academia, and open source hackers, we will have to limit ourselves to a few. We will furthermore only evaluate these tools in relation to parts of the taxonomy. These evaluations should be seen as paradigmatic examples that the reader can use to perform his own evaluations using the taxonomy.

It is difficult to choose a representative selection of tools that sheds light on all the different facets of software development environments. We have chosen to base our evaluation on four environments to show how different parts of the taxonomy can be used. The evaluation should not be perceived as an ordinary comparison. The selected environments, to some extent, address different problems and our intention here is not find to the best one. The four environments are:

Visual Studio 2005 Team System [83]
: Microsoft's Visual Studio product has been through several versions. We have chosen the 2005 Team System edition which is currently the latest. This product is a general-purpose software development environment which has been extended with some interesting process-related features.

Rational Software Architect 6.0 [98]
: IBM's product Rational Software Architect is similar to Visual Studio in scope and purpose. This is also a general-purpose software development environment which has been extended beyond mere coding capabilities. The environment consist of a large set of plugins to the open source platform Eclipse [36].

Visual Works Smalltalk 7.0 [21]
: Different Smalltalk environments have played a significant role in the evolution of software development process. We have included Visual Works in our comparison both to acknowledge its significance but also because Visual Works is still a competitive and relevant general-purpose environment.

Emacs 21.4 [99]
: Similar to Smalltalk, Emacs have also played a major role historically and is still the tool of choice for a large number of programmers. Emacs is - especially on the Unix/Linux platforms - very well integrated with the commandline facilities of the operating system. We have chosen to ignore this in our comparison and view tools such as CVS, Make, FTP etc as external to Emacs. It is a classic general-purpose environment which has great appeal to people working according to our artist-metaphor in section 2.1.

Our evaluation is based on our own experiences with these tools as well as the official datasheets and documentation from the different vendors. The products are evaluated according to a subset of the features in our taxonomy.

The evaluation is therefore split in four parts. One for each dimension or for each P in our 4P taxonomy. Feature support can fall in three categories: 1) fully supported (+), 2) partially supported (-/+), or 3) not supported (-). The product names are abbreviated as explained in table 3.

Product	**Abbreviation**
Visual Studio 2005 Team System	VS2005TS
Rational Software Architect 6.0	RSA
Visual Works Smalltalk 7.0	VWS
Emacs 21.4	Emacs

Table 3: Abbreviations used in the next sections.

Since several of these environments can be extended beyond recognition, we have chosen to evaluate these products with their basic feature sets. Each environment contains only the plugins and settings that are part of the standard distribution. They have not been adapted in any way. We have chosen this restriction as extra plugins would allow these tools to support virtually any feature which we could come up with. It is simply not possible, for us, to properly map the plethora of plugins for a tool such as e.g., Rational Software Architect. When deciding on a tool, one should of course not ignore the plugin market for a prospective tool. With the advent of open source, the size, quality and rapid evolution of available plugins can easily become a deciding factor.

7.1 Program

We have chosen to evaluate the tool selection in terms of two facets along the Program dimension - the storage mechanism and the representation form. The different possible storage mechanisms are shown along the vertical axis in table 4. The traditional model of shared files storage is the most common. There may, as suggested in section 3.2.1, be several reasons why this model is preferred. We believe that the two main reasons are familiarity and tool extensibility.

Familiarity stems from the fact that most users of development environments are familiar and therefore more comfortable with the file system. Emacs is probably the best example of this. Emacs users typically have at least one buffer open with a shell, i.e., a direct interface to the file system. They consider the file system a focal point of development, and Emacs is therefore used to enhance their ability to interact with the filesystem.

Environments such as VS2005TS and RSA also use the shared file system, but with these tools the motivation is less obvious. All three environments

	VS2005TS	RSA	VWS	Emacs
Shared files storage	+	+	-	+
Repository storage	+	+	-	+
Database storage	-	-	-	-
In-memory storage	-	-	+	-

Table 4: *Storage mechanisms.*

provide a graphical interface that shields the user from direct interaction with the file system. Several of the files in a typical project are used to configure the project, workspace and graphical interface, so these files are not really meant for direct editing through a text editor. The combination of graphical user interface and configuration files are used to abstract away the details of file management to some degree. It might therefore be argued that these environments could benefit equally well from other storage mechanisms.

Tool extensibility is often richer in file based environments since it is easier to contribute for third party plugin developers. These third party developers can rely on their knowledge of established file formats, such as for instance the file structure of a Java file. It is often more difficult to motivate third party developers to learn idiosyncratic formats for database og in-memory storage. An environments such as RSA is based on the idea of extensibility and an open architecture, so this is a deciding factor in the choice of shared files as storage mechanism.

The main exception in Table 4 is VWS which is based on in-memory storage. The environment logs all editing actions and stores the complete state of the workspace as an image file when it is closed by the user. This approach has been very popular both in academia and industry, but today it has to a large degree been overtaken by environments such as VS2005TS and RSA. The main reason that VWS uses in-memory storage is the choice of representation form which is covered next.

	VS2005TS	RSA	VWS	Emacs
Uni-representation	-	-	+	-
Multi-representation	+	+	-	+

Table 5: *Representation form.*

VWS relies on a uni-representation as shown in table 5. VS2005TS, RSA and Emacs, on the other hand, relies on the more common scheme of multi-representation. The distinction between uni- and multi-representation was discussed in greater detail in section 3.2.2. Interestingly, some of these tools

actually show exceptions to the general categorization. VWS for instance stores the single representation in an image file when the environment is shut down. Nevertheless, the actual bodies of methods are not stored in this image file, but in a separate file which is referred to from the image file. This indicates that the distinction between uni- and multi-representation is not clear cut.

Similarly, Emacs is generally considered a multi-representational environment, but when it is used as a LISP editor, it is actually working directly on a single representation. Since Emacs is written in LISP, this is a common scenario when extending the environment. In ordinary Java, TeX or other programming scenarios, Emacs can be considered multi-representational.

7.2 Platform

Our four environments have different characteristics along the *platform* dimension. We have chosen to look at view features, such as notation, language support, synchronization and edit mode. The first of these, notation, is intimately linked to the last, edit modes. All four environments support textual notation when dealing with raw source code. This affects the edit modes as textual notation is often displayed in free text editors. As described in section 4.2.2, structured editing has not been very successful in relation to textual notations. This is probably why our selection of modern software development environments have all opted for free editing.

	VS2005TS	RSA	VWS	Emacs
Textual notation	+	+	+	+
Graphical notation	+	+	-	-
On demand synchronization	-	-	+	+
Incremental synchronization	+	+	-	-
Constant synchronization	-	-	-	-
Free editing	+	+	+	+
Structured editing	+	+	-	-

Table 6: Support for different *view* features.

As shown in Table 6, VS2005TS and RSA also offers graphical notations such as class diagrams. Both environments offer structured graphical editors for these notations. Structured editing is considered a useful feature in graphical environments as lines automatically connect with boxes in drag and drop scenarios. It would be a nuissance if the user had to use too much time on aiming when connecting different graphical entities. This is

in contrast to textual environments where structured editing has been less convincing.

	VS2005TS	RSA	VWS	Emacs
Syntax highlighting	+	+	+	+
Grammatical structure	+	+	+	+
Static semantics	+	+	-	-

Table 7: Language support

All four environments offer language support as Table 7 shows. VS2005TS, RSA and Emacs even offers an extensible set of languages using modes, plugins and various adaptions. Emacs offers syntax highlighting for various languages out of the box. VS2005TS and RSA offers support for .NET and Java respectively. Both can be extended with additional languages, but when working with the primary language these environments offer extra benefits, such as object browsers and incremental synchronization, as described in section 4.2. VWS on the other hand is tailored for Smalltalk and it does not make sense to consider VWS independently of that language. All four environments offers text editors with an understanding of the grammatical structure of programs such that for instance correct indentation can be provided. VS2005TS and RSA extends this with continuous analysis of the program's static semantics which enables *intellisense*, auto-completion and other advanced features.

Generally, we can conclude that language support is a significant requirement for any software development environment today. The recent interest in meta CASE environments, as described in section 4.3.4, is a good example of this requirement since the focal point of meta CASE tools is the automatic generation of language-oriented editors.

7.3 People

Our selection of environments can be split into two general categories along the *people* dimension. VS2005TS and RSA both aim to support the entire life cycle of a project. This means that they must be suitable for all different roles in a project. We believe, as shown in Table 8, that both environments succeed in supporting every role except the analyst. The analyst requires a graphical and preferably domain-specific notation for his work. This kind of support is only present in custom-made tools or environments generated from meta CASE tools. The central trade-off that both VS2005TS and RSA has made in their effort to cover the complete life cycle has been to focus on

expert users. Both environments support a lot of different functions and are therefore very complex. This excludes most novice and intermediate users from mastering these environments.

	VS2005TS	RSA	VWS	Emacs
Analyst	-	-	-	-
Designer	+	+	-	-
Developer	+	+	+	+
Tester	+	+	-	-
Deployment manager	+	+	-	-
Project manager	+	+	-	-

Table 8: Support for different *areas of expertise*.

VWS and Emacs represent the second category along the *people* dimension. These environments are focused on the coding phase and all efforts are directed towards supporting the programmer in his tasks. Both environments are consequently significantly easier to use as opposed to VS2005TS and RSA. The two categories, life cycle versus coding environments, can be seen as instantiations of the work metaphors described in section 2.1. Environments, such as VS2005TS and RSA, that cover the entire life cycle appeals to the engineer. Here the focus is on planning and coordination. Environments, such as VWS and Emacs, on the other hand appeals to the artist or craftsman. Here the idea of fast, manual and *ad hoc* production is the essence.

7.4 Process

Finally, our selection of environments can be evaluated along the *process* dimension. We have chosen to look at some configuration management features, i.e., build management and version control, here. As shown in Table 9, our environments fall into three categories when it comes to build management. VS2005TS and RSA rely on automatic builds. These builds are formalized in scripts and executed by specialized build engines. The user can generate these scripts automatically and tailor them manually if necessary. Emacs on the other hand relies on manual builds. This is part of the artist or craftsman attitude which dictates fine-grained control of every part of a project. On larger projects this attitude can become infeasible and that is when engineering tools such as VS2005TS and RSA take over. VWS does not really fall into either of these two categories. Since VWS uses a uni-representation behind the scenes, it is not necessary to build the program at

all. The user interacts directly with the internal representation and hence this representation does not need to be built.

	VS2005TS	RSA	VWS	Emacs
Manual builds	-	-	n/a	+
Automatic builds	+	+	n/a	-
Continuous integration	+	-	-	-
Version control	+	+	+	+

Table 9: Support for *configuration management*.

All four environments offer version control out of the box. VS2005TS adds continuous integration on top of this. Since version control is a large and very complicated area, all these environments focus on offering simple client functionality for the most common version control systems. More advanced version control schemes typically requires both server-side software as well as client-side adaptions in these environments.

Generally, we can conclude that all four environments are expected to offer some process support, such as build management and version control. This indicates an acceptance of the fact that software development is a collaborative effort and the software development environments should as a bare minimum offer some collaboration support.

8 Related work

Several researchers have previously organized software development environments in conceptual schemes. The key motivation is the lack of cognitive tools to support our understanding of the field. It is recognized that the characteristics and features of an environment, to a large extent, can determine the outcome of a project. Nevertheless, we claim that no one has yet succeeded in creating a comprehensive conceptual framework to actually understand these characteristics and features. In this section, we will describe some of the most interesting attempts in the past.

One of the most comprehensive conceptual studies of software development environments is Nørmark's report on programming environments [2]. This report introduces a notation called *programming form diagrams* to describe the relation between the *program* and the *platform* dimension in an environment. The report emphasizes program construction, examination and administration as three important dimensions of software development environments. The discussion of pros and cons of different editing modes is especially intriguing and has been a major influence in the related

discussion in this paper. There is furthermore a very detailed description of representation which has also inspired the discussion here. Nørmark's focus is on tools such as Emacs [11, 12] and Interlisp as well as language-based environments such as Cornell Program Synthesizer [23] and Gandalf [24]. These tools are also examined by Mancoridis [100] who place them in a multi-dimensional taxonomy based on previous surveys.

Hausen and Muellerburg's [101] survey describes 20 tools and is mainly drawn from a workshop on software engineering environments in 1980. The authors emphasize that environments should be evaluated on their ability to represent the life cycle of a project. This is important as it is an early realization of the fact that environments should be able to support more than merely the programming phase. Part of what the authors denote the *life cycle model* is also considerations on how environments may imply a certain method. These ideas are very similar to the process dimension of our taxonomy. The main difference is that the authors do not include decomposition techniques which, in our view (section 6.4), is also a major part of the process embedded in environments.

Other surveys take a more conceptual approach by describing generic software development environment models. Howden [1] proposes a taxonomy where environments are organized by the cost of their project. Environments can be put into four categories ranging from minor projects to very large projects. Each category has special tool and architecture requirements. The emphasis here is on the composition of an environment's toolkit. Perry and Kaiser [102] also propose four categories, but they base their categories on the number of developers involved instead. Their categories use the sociological metaphors of the individual, the family, the city, and the state. Each category can be described by what policies, mechanisms and structures it employs. We believe that these conceptual models provide powerful cognitive schemes but they ignore a lot of the nuances that are captured on the detailed level of our taxonomy. A saying has it that *the devil is in the detail* and hence a taxonomy must provide a more fine-grained description than these two surveys.

A different class of related surveys are the tool-centered ones which focus on the definition and description individual tools. Reiss [43] offers a set of definitions for tools, such as editors, compilers, debuggers etc. These definitions can be used to establish a common vocabulary when discussing especially programming tools. Reiss [43] and Meyers [16] offer detailed descriptions of the integration models of software development environments which they complement with lots of practical examples. We have based parts of our taxonomy on the their work with regard to our program and platform dimension. We extend their work by fitting tool integration and

individual programming tools into the larger context of software development environments.

Harrison et al.'s [97] survey provides a historical overview of the key trends in software development environments. They propose a historical time line with four phases. In the first phase, environments were loosely coordinated. In the second phase, coordination of work flow was introduced. In the third phase, programming support became a major topic. Finally in the fourth phase, life cycle models and process support entered the stage. We agree with their historical account although it should be noted that these four phases are not necessarily sequentially ordered as even the fourth phase can be traced back to the seventies. A major insight in their survey is the emphasis on decomposition techniques where they warn against environments that only support a single decomposition technique.

Finally, two other important surveys introduce compressive categorizations of existing tools. Grundy and Hoskings [44] survey from the Wiley Encyclopedia of Software Engineering [103] proposes 18 categories of tools, such as requirements tools, formal methods tools etc, which they relate to the life cycle of a project. Their survey is at least as comprehensive as our taxonomy. The main difference is that our taxonomy offers one generic way of describing environments whereas their survey offers 18 ways of categorizing tools. A similar and equally comprehensive survey by Kelleher and Pausch [85] evaluates a wealth of tools on the tool's ability to support novice programmers. Their survey is a paradigmatic example of conceptualization about software development environments but their focus is much more narrow than the focus in our taxonomy.

9 Conclusion

This paper proposes a taxonomy for software development environments. The taxonomy is structured along four dimensions: Platform, Program, People and Process. We have analyzed these four dimensions using cardinality-based feature modeling which has shown how different facets of each dimension relates to one another. It is our claim in this paper that the taxonomy provides an important starting point for a comprehensive study of software development environments. We have shown how to apply the taxonomy on a small selection of environments to prove that our ideas can serve as a useful cognitive tool. We believe that such cognitive tools are required in order to get an overview of the plethora of past and current software development environments.

The main contributions of this paper are:

1. A taxonomy that describes central distinctions and trade-offs in the area of software development environments. The full taxonomy can be found on page 62

2. An application of this taxonomy that shows how our distinctions and ideas can be used to evaluate concrete software development environments.

3. A survey of important parts of literature and tools in this area during the last 35 years.

We believe that the use of cardinality-based feature modeling has proven a valuable tool when trying to survey the wealth of information in this area. Interestingly, we have discovered that the final feature models to a large degree depend on an initial implicit decision, wiz., our decision to take literature about tools rather than tools as the starting point. This decision was motivated by our desire to include older tools that are unavailable today in the survey. We believe that our emphasis on literature has given the feature models a focus on concepts rather than concrete features. This is especially visible in section 7. The application of the taxonomy is focused on the underlying concepts rather than the concrete features from each environment's datasheet. It would be interesting to see if a feature model which was based on data sheets would resemble our proposal.

Another interesting idea for future work would, of course, be to perform a more comprehensive application of the taxonomy. The taxonomy would probably need several small-scale alterations if it was applied on a larger selection of software development environments. Nevertheless, we believe that the four central dimensions will stand. Furthermore, we also claim that the general approach of gradually exploring the features and their relations is the most fruitful way of understanding what software development environments really are.

Acknowledgements:

We would like to thank Peter Sestoft, Kasper Østerbye and Jacob Winther Jespersen for comments and reviews of earlier versions of this paper.

References

[1] Howden, W.E.: Contemporary software development environments. Commun. ACM **25**(5) (1982) 318–329

[2] Nørmark, K.: Programming Environments - Concepts, Architectures and Tools. Technical Report R-89-5, Aalborg Universitetscenter (1989)

[3] Knuth, D.E.: Computer programming as an art. Commun. ACM **17**(12) (1974) 667–673

[4] TeX Users Group: Just what is TeX. (`http://www.tug.org/whatis.html`) Last accessed April 1, 2006.

[5] Philipkoski, K., Philipkoski, K.: Kent: The Genome Superman. `http://www.wired.com/news/medtech/0,1286,46154,00.html` (2001) Last accessed April 1, 2006.

[6] Naur, P., Randell, B., eds.: Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968. (1969) Brussels, Scientific Affairs Division, NATO, 231pp.

[7] Kim, C.H.P., Czarnecki, K.: Synchronizing Cardinality-Based Feature Models and their Specializations. In: Proceedings of ECMDA'05. (2005)

[8] Czarnecki, K., Helsen, S., Eisenecker, U.W.: Formalizing cardinality-based feature models and their specialization. Software Process: Improvement and Practice **10**(1) (2005) 7–29

[9] Fuggetta, A.: A Classification of CASE Technology. Computer **26**(12) (1993) 25–38

[10] Dolotta, T.A., Mashey, J.R.: An introduction to the Programmer's Workbench. In: ICSE '76: Proceedings of the 2nd international conference on Software engineering, Los Alamitos, CA, USA, IEEE Computer Society Press (1976) 164–168

[11] Gosling, J.: A redisplay algorithm. In: Proceedings of the ACM SIGPLAN SIGOA symposium on Text manipulation, New York, NY, USA, ACM Press (1981) 123–129

[12] Stallman, R.M.: EMACS the extensible, customizable self-documenting display editor. In: Proceedings of the ACM SIGPLAN SIGOA symposium on Text manipulation, New York, NY, USA, ACM Press (1981) 147–156

[13] Neighbors, J.M.: Software Construction using Components. PhD thesis, Department of Information and Computer Science, University of California, Irvine (1980) published as Technical Report UCI-ICS-TR-160.

[14] Neighbors, J.M.: Chapter 12. Draco: a method for engineering reusable software systems. ACM Frontier Series. In: Software Reusability, Volume I: Concepts and Models. Addison-Wesley, New York, NY, USA (1989) 295–319

[15] Object Management Group: Model Driven Architecture. (`http://www.omg.org/mda/`) Last accessed March 5, 2006.

[16] Meyers, S.D.: Representing Software Systems in Multiple-View Development Environments. PhD thesis, Brown University (1993) published as Technical Report CS-93-18.

[17] Grune, D.: Concurrent Versions System, a method for independent cooperation. Vrije Universiteit, Amsterdam. (1986) published on `http://www.cs.vu.nl/~dick/CVS.html`.

[18] Collins-Sussman, B., Fitzpatrick, B.W., Pilato, C.M.: Version Control with Subversion, v.1.1. O'Reilly Media (2005)

[19] Garlan, D.B.: Views for tools in integrated environments. PhD thesis, Carnegie-Mellon University (1987)

[20] Teitelman, W., Masinter, L.: The Interlisp Programming Environment. In Barstow, D.R., Shrobe, H.E., Sandewall, E., eds.: Interactive Programming Environments. McGraw-Hill, New York (1984) 83–96

[21] Cincom: VisualWorks Smalltalk. (`http://smalltalk.cincom.com/index.ssp`) last accessed March 2, 2006.

[22] Squeak. (`http://www.squeak.org/`) last accessed March 2, 2006.

[23] Teitelbaum, T., Reps, T.: The Cornell Program Synthesizer: a syntax-directed programming environment. Commun. ACM **24**(9) (1981) 563–573

[24] Habermann, A.N., Notkin, D.: Gandalf: Software development environments. IEEE Trans. on Software Engineering **12** (1986) 1117–1127

[25] Reiss, S.P.: PECAN: Program development systems that support multiple views. In: ICSE '84: Proceedings of the 7th international conference on Software engineering, Piscataway, NJ, USA, IEEE Press (1984) 324–333

[26] Reiss, S.P.: Consistent Software Evolution. `http://www.cs.brown.edu/~spr/research/clime/whitepaper.pdf` (2001)

[27] Reiss, S.P.: Constraining Software Evolution. In: ICSM '02: Proceedings of the International Conference on Software Maintenance (ICSM'02), Washington, DC, USA, IEEE Computer Society (2002) 162

[28] Backus, J.W., Bauer, F.L., Green, J., Katz, C., McCarthy, J., Perlis, A.J., Rutishauser, H., Samelson, K., Vauquois, B., Wegstein, J.H., van Wijngaarden, A., Woodger, M.: Report on the algorithmic language algol 60. Commun. ACM **3**(5) (1960) 299–314

[29] Object Management Group: Meta-Object Facility. (`http://www.omg.org/technology/cwm/`) Last accessed March 4, 2006.

[30] Microsoft: fxcop. (`http://www.gotdotnet.com/Team/FxCop/`) Last accessed March 3, 2006.

[31] Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional (1999)

[32] Lanza, M.: CodeCrawler. (`http://www.iam.unibe.ch/~scg/Research/CodeCrawler/`) Last accessed March 3, 2006.

[33] Jones, C.B.: Systematic software development using VDM (2nd ed.). Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1990)

[34] Spivey, J.M.: The Z notation: a reference manual. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1989)

[35] The RAISE Language Group: The RAISE Specification Language. Prentice Hall (1992)

[36] The Eclipse Foundation: Eclipse Java Development Tools. (`http://www.eclipse.org/jdt/`) Last accessed March 3, 2006.

[37] Microsoft: Visual Studio .NET 2005. (`http://msdn.microsoft.com/vstudio/`) Last accessed March 3, 2006.

[38] Hovemeyer, D., Pugh, W.: Finding bugs is easy. SIGPLAN Not. **39**(12) (2004) 92–106

[39] The Software Composition Group: Framework-based Approach for Mastering Object-Oriented Software Evolution (FAMOOS). (`http://www.iam.unibe.ch/~famoos/`) Last accessed March 3, 2006.

[40] Nuseibeh, B., Easterbrook, S., Russo, A.: Making inconsistency respectable in software development. The Journal of Systems and Software **58**(2) (2001) 171–180

[41] Fraternali, P.: Tools and approaches for developing data-intensive web applications: a survey. ACM Comput. Surv. **31**(3) (1999) 227–263

[42] : The Overture Tool. (`http://www.overturetool.org/index.php`) Last accessed March 24, 2006.

[43] Reiss, S.P.: Software tools and environments. ACM Comput. Surv. **28**(1) (1996) 281–284

[44] Grundy, J.C., Hosking, J.G.: Software Tools. In: Software Tools. 2 edn. Wiley InterScience (2001) Appears in [103].

[45] Wasserman, A.I.: Tool integration in software engineering environments. In: Proceedings of the international workshop on environments on Software engineering environments, New York, NY, USA, Springer-Verlag New York, Inc. (1990) 137–149

[46] Reiss, S.P.: Connecting tools using message passing in the field environment. IEEE Softw. **7**(4) (1990) 57–66

[47] Harrison, W., Kavianpour, M., Ossher, H.: Integrating coarse-grained and finegrained tool integration. In: Proceedings of the Fifth International Workshop on Computer-Aided Software Engineering (CASE '92). (1992) 23–35

[48] Brown, A.W., Penedo, M.H.: An annotated bibliography on integration in software engineering environments. SIGSOFT Softw. Eng. Notes **17**(3) (1992) 47–55

[49] The Eclipse Foundation: Eclipse Plug-in Development Environment. (`http://www.eclipse.org/pde/`) Last accessed March 3, 2006.

[50] Gamma, E., Beck, K.: Contributing to Eclipse. The Eclipse Series. Addisson-Wesley (2003)

[51] Czarnecki, K., Eisenecker, U.W.: Generative Programming - Methods, Tools, and Applications. Addison-Wesley (2000)

[52] Kaiser, G.E., Feiler, P.H.: An architecture for intelligent assistance in software development. In: ICSE '87: Proceedings of the 9th international conference on Software Engineering, Los Alamitos, CA, USA, IEEE Computer Society Press (1987) 180–188

[53] Hansen, W.J.: Creation of Hierarchic Text with a Computer Display. PhD thesis, Dept. of Computer Science, Stanford University, California (1971)

[54] JetBrains: Meta Programming System. (`http://www.jetbrains.com/mps/`, 12. December 2005)

[55] Fowler, M.: Language Workbenches: The Killer-App for Domain Specific Languages? (`http://www.martinfowler.com/articles/languageWorkbench.html`, 12. June 2005)

[56] Roberts, D., Brant, J., Johnson, R.: A refactoring tool for smalltalk. In: Theory and Practice of Object Systems. Volume 3. (1997) 253 – 263

[57] Opdyke, W.F.: Refactoring Object-Oriented Frameworks. PhD thesis, University of Illinois at Urbana-Champaign, Dept. of Computer Science, Urbana-Champaign, IL, USA (1992) published as technical report UIUCDCS-R-92-1759.

[58] Edwards, J.: Example centric programming. SIGPLAN Not. **39**(12) (2004) 84–91

[59] Edwards, J.: Subtext: uncovering the simplicity of programming. In: OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications, New York, NY, USA, ACM Press (2005) 505–518

[60] Reiss, S.P.: A conceptual programming environment. In: ICSE '87: Proceedings of the 9th international conference on Software Engineering, Los Alamitos, CA, USA, IEEE Computer Society Press (1987) 225–235

[61] Joy, W.: An Introduction to Display Editing with Vi. Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley. (1981)

[62] JetBrains: IntelliJ IDEA. (http://www.jetbrains.com/idea/) Last accessed March 5, 2006.

[63] Martin, J.: Applications Development Without Programmers. Prentice Hall (1981)

[64] Horowitz, E., Kemper, A., Narasimhan, B.: A survey of application generators. IEEE Software **2**(1) (1985) 40–54

[65] Oracle: Oracle Reports. (http://www.oracle.com/technology/products/reports/index.html) Last accessed March 5, 2006.

[66] SAP: SAP R3 ABAP. (http://www.sapgenie.com/abap/) Last accessed March 5, 2006.

[67] Microsoft: Navision C/AL. (http://www.microsoft.com/danmark/mbs/losninger/navision.asp) Last accessed March 5, 2006.

[68] Microsoft: Microsoft Access. (http://office.microsoft.com/en-us/FX010857911033.aspx) Last accessed March 5, 2006.

[69] SAS Institute: SAS. (http://www.sas.com/) Last accessed March 5, 2006.

[70] Microsoft: Visual Basic. (http://en.wikipedia.org/wiki/Visual_Basic) Last accessed March 5, 2006.

[71] The Software Engineering Institute: What is a CASE Environment? http://www.sei.cmu.edu/legacy/case/case_whatis.html (2004)

[72] Pressman, R.S., Ince, D.: Software Engineering: A Practitioner's Approach (European Adaption). 5 edn. McGraw-Hill (2000)

[73] Iivari, J.: Why are CASE tools not used? Commun. ACM **39**(10) (1996) 94–103

[74] Jarzabek, S., Huang, R.: The case for user-centered case tools. Commun. ACM **41**(8) (1998) 93–99

[75] MetaCASE: MetaEdit+. (http://www.metacase.com/mep/) Last accessed March 5, 2006.

[76] Institute for Software Integrated Systems: The Generic Modeling Environment. (http://www.isis.vanderbilt.edu/projects/gme/) Last accessed March 5, 2006.

[77] Xactium: XMF-MOSAIC. (http://albini.xactium.com/web/) Last accessed March 5, 2006.

[78] Microsoft: Visual Studio DSL Tools. (http://msdn.microsoft.com/vstudio/DSLTools/) Last accessed March 5, 2006.

[79] Grundy, J., Hosking, J., Mugridge, W.B.: Inconsistency Management for Multiple-View Software Development Environments. IEEE Trans. Softw. Eng. **24**(11) (1998) 960–981

[80] Balzer, R.: Tolerating inconsistency. In: ICSE '91: Proceedings of the 13th international conference on Software engineering, Los Alamitos, CA, USA, IEEE Computer Society Press (1991) 158–165

[81] Nuseibeh, B.: To Be and Not to Be: On Managing Inconsistency in Software Development. In: IWSSD '96: Proceedings of the 8th International Workshop on Software Specification and Design, Washington, DC, USA, IEEE Computer Society (1996) 164

[82] Nygaard, K., Handlykken, P.: The System Development Process - Its setting, some problems and needs for methods. In Hünke, H., ed.: Software Engineering Environments Proceedings of a Symposium (S2E2). (1980) cited from [101].

[83] Microsoft: Visual Studio 2005 Team System. (http://msdn.microsoft.com/vstudio/products/vsts/default.aspx) Last accessed March 22, 2006.

[84] IBM: Rational Software. (http://www-306.ibm.com/software/rational/) Last accessed March 22, 2006.

[85] Kelleher, C., Pausch, R.: Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. ACM Comput. Surv. **37**(2) (2005) 83–137

[86] The Mozilla Organization: Bugzilla. (http://www.bugzilla.org/) Last accessed March 22, 2006.

[87] Conradi, R., Westfechtel, B.: Version models for software configuration management. ACM Comput. Surv. **30**(2) (1998) 232–282

[88] GNU: Arch. (http://gnuarch.org/index.html) Last accessed March 8, 2006.

[89] Feldman, S.I.: Make - A Program for Maintaining Computer Programs. Software - Practice and Experience **9**(4) (1979) 255–65

[90] The Apache Software Foundation: Apache Ant. (http://ant.apache.org/) Last accessed March 24, 2006.

[91] Fowler, M., Foemmel, M.: Continuous Integration. ({http://www.martinfowler.com/articles/continuousIntegration.html}) Last accessed March 24, 2006.

[92] : CruiseControl. (http://cruisecontrol.sourceforge.net/) Last accessed March 24, 2006.

[93] Chive Software Limited: Draco.NET. (http://draconet.sourceforge.net/) Last accessed March 24, 2006.

[94] Wirfs-Brock, R.J., Johnson, R.E.: Surveying current research in object-oriented design. Commun. ACM **33**(9) (1990) 104–124

[95] IBM Research: Concern Manipulation Environment (CME): A Flexible, Extensible, Interoperable Environment for AOSD. (http://www.research.ibm.com/cme/) Last accessed March 24, 2006.

[96] Tarr, P., Ossher, H., Harrison, W., Stanley M. Sutton, J.: N degrees of separation: multi-dimensional separation of concerns. In: ICSE '99: Proceedings of the 21st international conference on Software engineering, Los Alamitos, CA, USA, IEEE Computer Society Press (1999) 107–119

[97] Ossher, H., Harrison, W., Tarr, P.: Software engineering tools and environments: a roadmap. In: ICSE '00: Proceedings of the Conference on The Future of Software Engineering, New York, NY, USA, ACM Press (2000) 261–277

[98] IBM Rational Software: Rational Software Architect V6.0. ({http://www-306.ibm.com/software/awdtools/architect/swarchitect/}) Last accessed March 25, 2006.

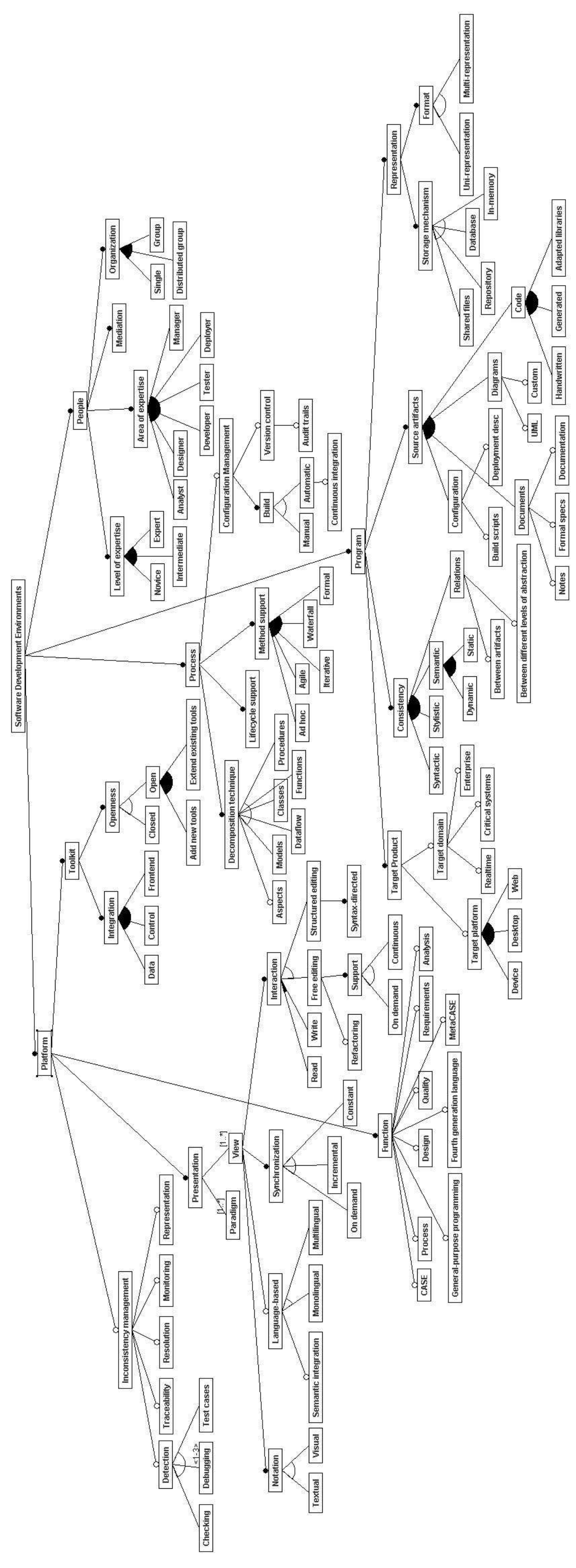
[99] Free Software Foundation / GNU: GNU Emacs V21.4. (http://www.gnu.org/software/emacs/) Last accessed March 25, 2006.

[100] Mancoridis, S.: A multi-dimensional taxonomy of software development environments. In: CASCON '93: Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research, IBM Press (1993) 581–594

[101] Hausen, H.L., Muellerburg, M.: Conspectus of software engineering environments. In: ICSE '81: Proceedings of the 5th international conference on Software Engineering, Piscataway, NJ, USA, IEEE Press (1981) 34–43

[102] Perry, D.E., Kaiser, G.E.: Models of software development environments. In: ICSE '88: Proceedings of the 10th international conference on Software engineering, Los Alamitos, CA, USA, IEEE Computer Society Press (1988) 60–68

[103] Marciniak, J.J.: Encyclopedia of Software Engineering. John Wiley & Sons, Inc., New York, NY, USA (2002)



Software Development Environments
Platform
Toolkit
Openness
Closed
Open
Add new tools
Extend existing tools
Integration
Data
Control
Frontend
People
Level of expertise
Novice
Intermediate
Expert
Area of expertise
Analyst
Designer
Developer
Tester
Deployer
Manager
Mediation
Organization
Single
Group
Distributed group
Process
Lifecycle support
Method support
Ad hoc
Agile
Iterative
Waterfall
Formal
Configuration Management
Version control
Audit trails
Build
Manual
Automatic
Continuous integration
Decomposition technique
Aspects
Models
Dataflow
Classes
Functions
Procedures
Program
Representation
Format
Uni-representation
Multi-representation
Storage mechanism
Shared files
Repository
Database
In-memory
Source artifacts
Configuration
Build scripts
Deployment desc
Documents
Notes
Formal specs
Documentation
Diagrams
UML
Custom
Code
Handwritten
Generated
Adapted libraries
Consistency
Syntactic
Stylistic
Semantic
Dynamic
Static
Relations
Between artifacts
Between different levels of abstraction
Target Product
Target domain
Realtime
Critical systems
Enterprise
Target platform
Device
Desktop
Web
Inconsistency management
Detection
<1-3>
Checking
Debugging
Test cases
Traceability
Resolution
Monitoring
Representation
Presentation
[1..*]
Paradigm
[1..*]
View
Notation
Textual
Visual
Language-based
Semantic integration
Monolingual
Multilingual
Synchronization
On demand
Incremental
Constant
Interaction
Read
Write
Free editing
Structured editing
Syntax-directed
Refactoring
Support
On demand
Continuous
Function
CASE
Process
General-purpose programming
Design
Fourth generation language
Quality
MetaCASE
Requirements
Analysis