Feature-Oriented Classification and Reuse in IPSEN

Jürgen Börstler
Lehrstuhl für Informatik III
Aachen University of Technology
Ahornstraße 55, W-5100 Aachen, Germany
jubo@rwthi3.informatik.rwth–aachen.de

Abstract

A successful approach to aid development with reuse is the usage of libraries to store software documents. To support the organization of such libraries specialized classification schemes evolved. Most of the approaches developed so far are not generally applicable for two reasons. They are based on more or less static classification schemes and/or only consider functional properties of the stored documents. To overcome both deficiencies we propose a new approach, called feature-oriented classification.

To use such libraries successfully it is important to be able to understand and to adapt the recovered documents. To support development for reuse we propose the usage of a sophisticated design language to provide developers with a single language in which they can adequately document their design decisions.

Keywords: Reusability, Classification, Libraries, Design, Programming-in-the-Large, Tools, Software Development Environments, Integration, Architectural Frameworks

1 Introduction

During software development reuse can be seen from two points of view.

Development for reuse addresses the fact that documents[1] are not reusable per se. Documents must have been developed for later reuse. Therefore, tools are needed to increase the reusability of documents. A document itself supports reuse, if it encapsulates a single, well-defined abstraction, and hides the details of its realization. It should clearly list its dependencies on other documents, and/or the environment, but there should be as few as possible such dependencies. To say it in other words, the application of the well known (design) principles, like abstraction, information hiding, encapsulation, structuring, parameterization, coupling, and cohesion is critical for reuse. The importance of these principles is well-known (/Parn 72/), and though a lot them found their way into programming language concepts. In /Börs 91b/ we discussed how languages and tools for Programming-in-the-Large (PiL) can help in the development of reusable documents (see section 3.3).

Development with reuse deals with the usage of such predefined documents to build new applications. Therefore, tools are needed to store and recover these documents. In section 3.2 we will describe how feature-oriented classification can be used to describe and structure the properties of documents, support efficient storage and retrieval of reusable documents.

In addition the (re-)users want some help in adapting the documents to their special needs, and to incorporate them into their own environment. To support this task the tolls for PiL are again very useful. Thus the key question is "is this the right document, and if so, how can I reuse it?" For SDEs (software development environments) to be successful, it is therefore important, to give support for development for reuse and development with reuse.

2 Levels of Reuse in IPSEN

IPSEN (**i**ntegrated and incremental software **p**rojects **s**upport **en**vironment) is an integrated SDE, i.e. a comprehensive set of cooperating tools, which supports the development and maintenance of software. The documents produced during software development represent the software project at different levels of abstraction, and in different degrees of formality. The contents of these documents are therefore highly interrelated. These relationships can be established between increments of one document, as well as different documents. We even support the definition of relationships between increments of different document types. Such relationships are used for ex-

1. Documents can be all kinds of (partial) results produced during the life-cycle of (software) systems, like requirements definitions, designs, code, documentation, test cases, and so on.

ample to connect increments in a requirements definition to their corresponding counterparts in design documents (/BöJa 92/). There exist tools to analyze and manipulate such relationships (/Lef 93/). Compared to usual hypertext systems it should be noted that our links are typed, and may have different semantics dependent on the (type of) linked increments. We also support course-grained links, which can relate complex objects (which may themselves include other links). In this way a software project can be described as an integrated set of (consistent) documents (/LNW 87/).

To enable such an interaction of documents tools one needs sophisticated internal integration mechanisms. In our project IPSEN we proved attributed graphs to be an appropriate formal basis for this purpose. Therefore, all documents in IPSEN are internally described by graph schemes, and the tools are described by graph transformation rules (/Nagl 86/).

IPSEN is designed as an **integration framework** (/Lef 93/). Following some special guidelines new tools can be created, and integrated into IPSEN without (major) modifications to already existing ones. This was achieved through a unified model of documents and the tools that manipulate these documents. This leads us to three different views of reuse in IPSEN:

(a) The models-builders view:

Based on the unified model we have developed generators for syntax-, and semantics directed editors, as well as general purpose components to support complex analyses, command handling, etc. The models-builders are mainly concerned with development-for-reuse. They make available the integration framework to be used by the tools-builders. In this sense the models-builders create a model of a software development environment, which can be filled with life by the tools-builders.

(b) The tools-builders (models-users) view:

The tools-builders (re-)use the facilities created by the models-builders to develop and integrate new tools into the framework. From the tools-builders point of view, development-for-reuse is merely a matter of language design. After identifying a domain of interest, syntax, semantics, and representation of an appropriate language are defined. With the help of the tools developed by the models-builders an editor for this language can be generated and integrated into the framework.

(c) The tools-users view:

Most of the integrated tools are not especially tuned to support reuse on the tools-users level, but there are some notable exceptions. Our editor for programming-in-the-large (PiL) supports a language which offers most of the facilities, which have shown up to be important for the design-for-reuse. We also developed an editor for the development and maintenance of classification schemes to sup-

port the organization of document libraries. Both aspects are discussed in more detail in the sequel of this paper.

3 Reuse in IPSEN from the Tools-Users Point of View

All IPSEN tools share the same characteristics. They operate on a document through a unified user interface. Commands are filtered, so that only useful commands can be executed. After execution of each command all affected documents are updated incrementally. Links between different documents are also kept consistent. A snapshot of a set of interrelated documents from the tools-users point of view is shown in figure 1. The documents reflecting the support for version control and project management are omitted for simplicity.

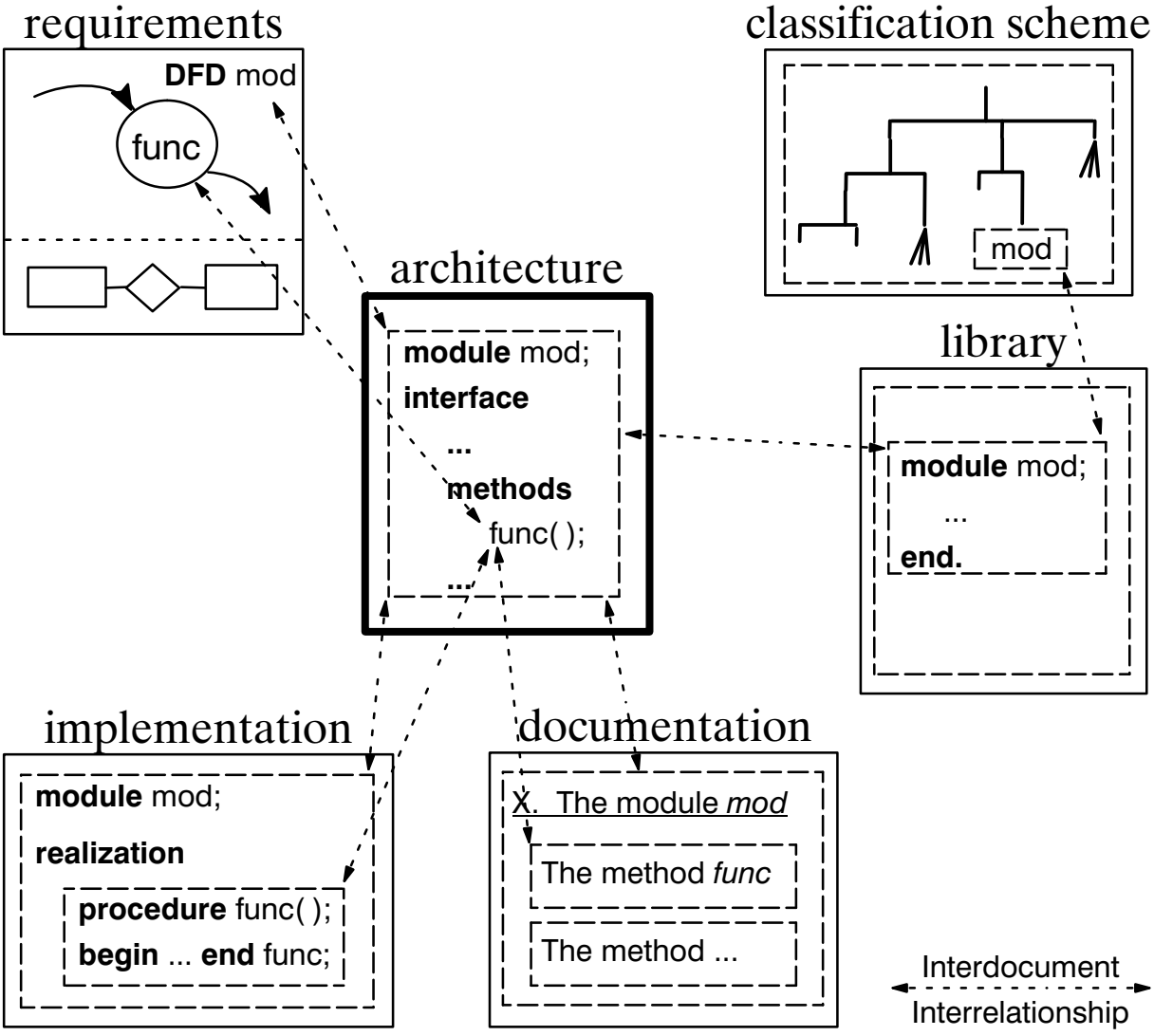

Figure 1: IPSEN documents from the tools-users point of view.

3.1 Development with Reuse

One promising approach to aid development with reuse is the usage of libraries to store the various documents which may be reused. But most of these libraries did not succeed as expected. The designers of such libraries noticed that it is not enough to add great bunches of code to the libraries, and leaving the (re-)users alone with it. Therefore, additional tools were developed to describe, store, and recover the various kinds of documents.

For the organization of a complex domain controlled vocabularies (idexes) have a long and successful tradition (/Linné 53/) for classification. Since documents usually have orthogonal aspects describing their properties multiple classification, i.e. the classification of documents according to more than one aspect (facet) in parallel, seems appropriate. There exist several recent approaches in this direction (/PrFr 87/, /Fra 89/, /HeFr 92/). Most of these approaches rely on static classification schemes and/or only consider functional properties of the stored documents. Our approach allows for changes in the classification schemes, since classification schemes tend to change as knowledge on the domains covered increases. We also support the classification of structural properties, as necessary for data-oriented documents.

3.2 Feature-Oriented Classification

In feature-oriented classification documents are classified by sets of features. We call such a set of features the descriptors of the document. To support the understanding and construction of descriptors the features are organized in a classification scheme. In this classification scheme the semantics of a feature is given through its successive refinements, until well-known notions are reached. These well-known notions correspond to the attributes or terms known from other classification schemes. The refinements span trees. Each of these trees contains features corresponding to the same aspect, similar to a facet, as used in faceted classification (/PrFr 87/). Therefore, a feature can be interpreted as a path in this tree. Figures 2 to 4 show an example classification scheme for the usual collection types/objects.

We do not require a complete classification of documents. We allow descriptors to contain non-leaf features, and also to completely omit the classification along some facet trees. We did this for two reasons:

(a) Some facet trees or views may only make sense for special kinds of documents.

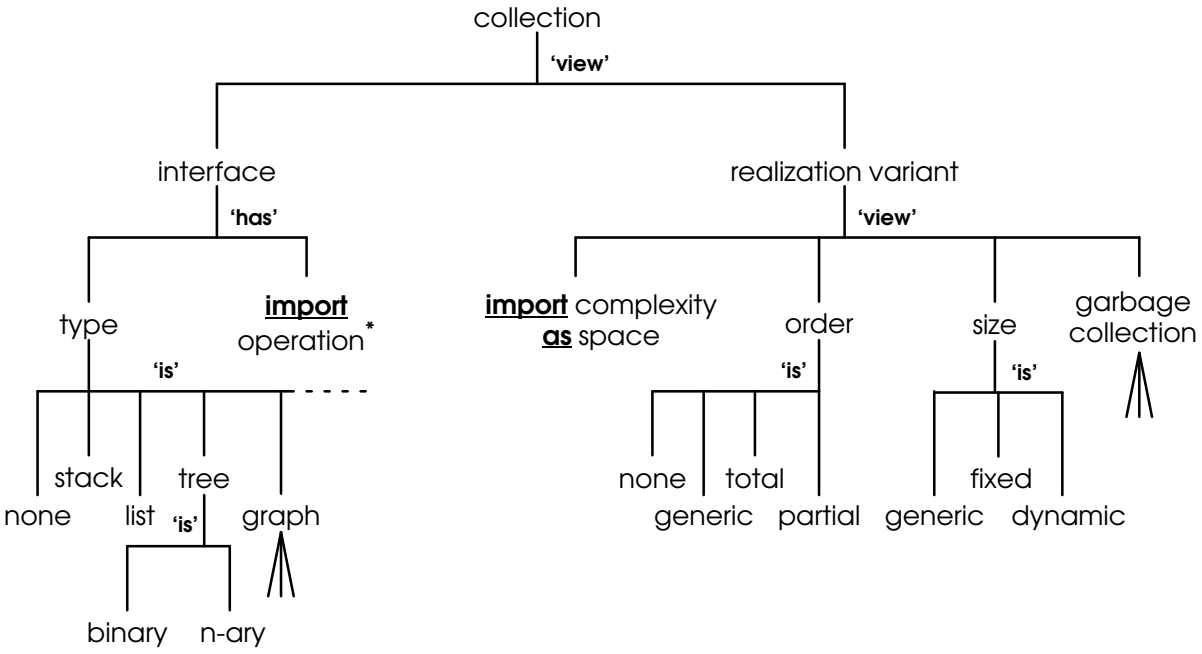

Figure 2: An example classification scheme (facet tree collection).

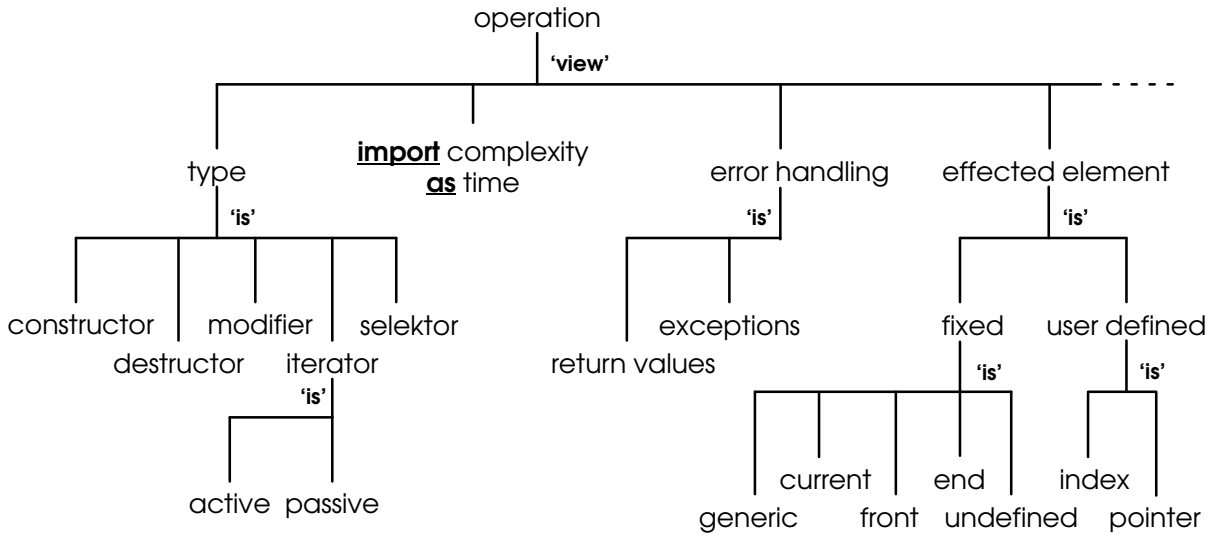

Figure 3: An example classification scheme (facet tree operation).

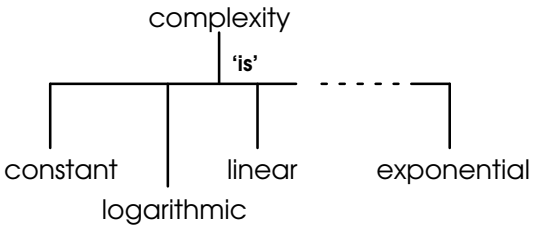

Figure 4: An example classification scheme (facet tree complexity).

(b) We want to support a step by step evolution of the descriptors. Therefore, we only require correct descriptors, and have no reason for rejecting a descriptor, because it is not detailed enough.

We support three different relationships, representing different semantics of refinement:

(a) A **view**-refinement represents a distinguishable aspect of a feature. A single document should be classified by at least one refinement of a feature. In our example a 'collection' can be classified according to its 'interface', and 'restrictions'.

(b) A **has**-refinement represents the decomposition of a feature. Therefore a single document should be classified by all of the refinements of a feature. In our example the 'interface'-view of a 'collection' is composed of 'type' and 'operation'.

The '*' denotes repetition. Since an interface usually has several operations, the descriptor of a collection should contain (sub-)descriptors for all of them. A descriptor may therefore contain more than one refinement of 'operations'.

(c) An **is**-refinement represents the specialization of a feature. Therefore a single document can only be classified by exactly one refinement of a feature. In our example the 'type' of an 'interface' can either be a 'stack', a 'list', a 'tree', or a '...'.

In using this scheme it turned out, that is-refined features can usefully be refined only by using further is-refinements. Other refinements are used to structure the name space, and therefore to support the understanding of features. In other words, only is-refinements give information to restrict the search space. Descriptors should therefore contain only features from is-refined subtrees. Following these rules we can define how correct and/or useful descriptors should look like. The definitions can then be embedded into tools to support the construction and analysis of descriptors. A document can then be classified step by step starting with an empty descriptor.

To avoid the duplication of work we support the **import**ation of facet trees into other facet trees. To assign the imported subtree an appropriate name importation is combined with renaming.

The nodes of the facet trees contain pointers to those documents satisfying the corresponding feature, i. e. one can interpret the set of all nodes pointing to a given document as the descriptor of this document. Since, each node in the classification scheme corresponds to a set of documents, the classification scheme can also be used as a means to browse through the document library.

Since is-refinement represents specialization, navigation along is-refinements of a feature will reduce the number of corresponding documents (matches). Navigation to a view- or has-refinement will focus on a special aspect or part of a document set. In a similar way this structure can also be used to find similar documents. Given a des-

criptor of a document all nodes corresponding to its features can be visited, and hence all documents with the same classification can be found. Navigation to children of the same refinement, or against the refinement will lead to documents representing very close matches with the classification of the current document.

In this way a similarity metric between documents can be computed from the similarity of their descriptors. We therefore perform a pairwise comparison of all features of the descriptors. Two descriptors are equal, if they contain identical sets of features. To compute the similarity of features we take into account the corresponding classification scheme. The closer two features are placed in the classification scheme the more similar they are. To improve the usefulness of this metric we introduced a special **subsumes**-relationship. This relationship is valid only between children of the same is-refinement. It indicates that one feature subsumes another one. In our example classification scheme the subsumes-relationship could be used, for e.g. to relate the children of the feature 'complexity' in the complexity facet tree (see figure 5). A retrieval can then be parameterized to take into account the subsumes-relationship. By doing so the retrieval algorithm will also return documents corresponding to descriptors, which contain features subsumed by features in the current descriptor. In our example classification scheme the descriptor '(x, collection →realization variant→space→logarithmic[2], y)' will therefore also return all documents corresponding to the descriptor '(x, collection→realization variant→space→ constant, y)'. Note that this is not true vice versa, since the subsumes-relationship is directed.

To improve the browsing capabilities, we also provide user-definable non-hierarchical relationships, e.g. to introduce hypertext like cross-references. These relationships do not contribute to our similarity metric.

In addition to the set of features, which classify a document, each document provides a set of basic attributes, to support qualification (certification). Some of them are simple metrics, which can be computed automatically (e. g. LOC), others are provided by developers (e. g. language used) and/or (re-)users (e. g. level of satisfaction) of the documents.

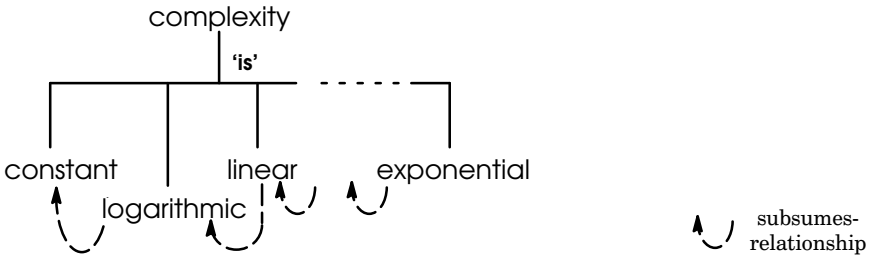

Figure 5: Revised facet tree complexity.

2. Note that features correspond to paths in the facet trees. Here we must actually indicate the whole path to remove ambiguities.

Retrieval can be supported in three ways:

(a) By **browsing**:

If one has only a very vague idea of the features of a document, one can start navigating through the classification scheme beginning at an appropriate (sub)tree (facet). Since all matching documents can be reached, from any node in the classification scheme, detailed information (basic attributes, documentation, interface) on each document can be displayed.

(b) By searching for matches **with a given descriptor**:

If one knows most of the features a document requires, one can construct an appropriate descriptor. The system will then return all documents matching this descriptor. The construction of a descriptor can be done manually, or by selecting a set of features in the classification scheme.

The matching algorithm takes the classification scheme into account. For each feature in the descriptor all documents corresponding to its is-refinements in the classification scheme will be returned. In our example a descriptor containing the feature 'tree' will yield all documents referred to from node 'tree', as well as those referred to from 'binary', and 'n-ary'. This set of documents will then be merged with the result built so far.

Merging can be done in several modes. The 'exact-match' mode will simply perform a set intersection. Other modes may relax descriptors, e. g. by replacing a feature by its is-generalization. In addition, features can be given weights to indicate their relative importance in a descriptor. These weights can also be considered by the matching algorithm.

(c) By searching for matches **with given basic attributes**:

This kind of retrieval will usually be used as a back end process to assess the documents found through (a), or (b).

3.2.1 Maintaining Classification Scheme and Descriptors

Since classification schemes tend to change, we built a syntax-directed editor for classification schemes. Like other IPSEN-tools, this editor provides support for analyzing classification schemes. In some cases, changes to the classification scheme may invalidate existing descriptors. Since our classification scheme contains references to the documents as described before, we have immediate access to all documents corresponding to descriptors possibly affected by a change to the classification scheme. We will now discuss how we can react to such changes. To simplify the discussion we distinguish between several kinds of such changes:

(a) **Non-structural changes** to the classification scheme (e.g. renaming a feature, or introducing non-hierarchical relationships).

Since our editor guarantees, that new names do not conflict with already existing ones, an update can be performed automatically. Changes to non-hierarchical relationships do only effect the browsing capabilities, and therefore do not effect descriptor construction.

(b) **Simple structural changes** to the classification scheme (insertion/deletion of leaf nodes, e.g. refining a feature).

Insertion of a new refinement offers the opportunity of more detailed classifications. Since our editor guarantees, that extensions do not conflict with the classification scheme built so far, all descriptors remain correct. Although, such modifications do not cause problems, it is important to propagate the updates regularly.

Deletion of a leaf node invalidates all descriptors containing the feature corresponding to this node. To correct this we can automatically replace this feature by its generalization. These corrections may produce descriptors containing useless features, i.e. features, which do not narrow the search space.

(c) **Complex structural changes** to the classification scheme (structural changes to non-leaf nodes, e.g. deleting a refined feature).

This category contains the most critical changes. Deleting a refined feature may not only invalidate all descriptors containing this feature, but also those, which contain refinements of this feature. The simplest way to correct this is to remove the whole subtree rooting at this feature and proceed as above. But in most cases we want to preserve this subtree and the corresponding classifications. How this can be done depends on the kinds of the two refinements surrounding the deleted feature.

Although we allow for changes in the classification scheme, complex changes should not happen very often. These changes do not only affect descriptors of classified documents, but also the descriptors used to search for documents. The same descriptor may yield a completely different result after a change in the classification scheme occurred. This will confuse users of the classification scheme. Changes should therefore not only syntactically save as described above, but also semantically save, i.e. only slightly effect the result yielded by a user query.

3.3 Development for Reuse

To benefit from libraries it is not only important to find an appropriate document, but also to understand it, and to adapt it to the problem at hand. In IPSEN PiL plays an important role to overcome these problems. This is due to the fact that most docu-

ments depend (directly or indirectly) on the software architecture document (see figure 2), which is created during PiL (/Börs 91b/).

Using a special language for PiL provides developers with a single language in which they can adequately document their design decisions in using appropriate concepts (/PrNe 86/). In the IPSEN project we use the language EMIL (**e**xtended **m**odule **i**nterconnection **l**anguage), which fulfils most of the needs of a modern language for PiL (/Börs 93/). EMIL supports different kinds of modules to cover functional, data, and architectural abstraction.

A **data type module** encapsulates an abstract data type similar to the class concept known from object−oriented programming languages (/Meyer 88/). It is used as a pattern for the dynamic creation of objects of the same type, which can only be created and manipulated by the operations offered at its (export) interface.

A **data object module** encapsulates a single, static data structure, which can not be created or deleted from outside the module. It can be seen as a named instance of an abstract data type. It is useful to make important objects visible in the architecture, for example collections of objects, which are used by several different modules.

A **function module** encapsulates one or more functions, which do not manage an own memory. Its purpose is to designate modules which transform input data to output data independent of an internal state.

A **simple module** encapsulates constants and basic (open) types. Its purpose is to encapsulate basic declarations. Simple modules must not export operations.

A **subsystem** encapsulates a set of interrelated modules, hiding the complexity of their actual interrelationships. Subsystems behave like modules in providing their own interfaces, and hiding their realizations. Subsystems realizations are described by separate architecture documents. In this way architectural abstraction is a consequent continuation of abstraction and information hiding on architecture level.

Modules can have formal **generic parameters**. The actual (type) parameters can be constrained by defining operations, which must be supported by instantiation.

Subtype relationships can be defined between the data abstraction modules, by means of an inheritance hierarchy. Data type modules can be abstract, that means they may export resources, which defer their implementations to subtypes. Usability relationships define import/export relationships between modules.

Version control is performed separately (/West 89/, /West 91/, see figure 1).

In addition to a sophisticated editor, additional tools are necessary to support the usage of this language, and to handle documents written in this language. These tools should not only have knowledge of the syntax and static semantics of the language—like all editors in IPSEN do—but also of rules which lead to ”good” architectures. This knowledge may be an implementation of published guidelines. Unfortunately most of these guidelines depend on special programming languages (e. g. /SPC 91/, /HeNy 92/), and/or focus on implementation details, which are not relevant for PiL (/OmCo

91/). Language independent rules can be derived from accepted design principles. The 'Law of Demeter' (/LiHo 89/, /Sakk 88/) for example is a set of rules, which restrict the coupling between classes in object-oriented designs. Another source for the definition of such rules are design metrics, which estimate the quality of a software architecture by evaluating its structure (/GiCo 84/).

The editor for EMIL gives some support in this direction. We have developed and implemented a set of guidelines for the construction of interfaces of data abstraction modules (see /Börs 91b/).

4 Conclusion and Future Work

In this paper we described the approaches taken in IPSEN to make reuse an integral part of the software development process. We identified several levels of reuse comprising four working areas, which are critical for reuse to be successful:

(a) Sophisticated tools for PiL, to support development-for-reuse.

(b) Flexible classification, to support development-with-reuse.

(c) Development of integration frameworks, to support the integration of new tools.

(d) Advanced internal integration mechanisms, to support all of them.

Although we only focused on the areas listed above, there are more working areas, where reuse can help to increase productivity. In IPSEN there are two such areas, where reuse will be examined in future. The first one is requirements specification. Besides the specification of functional and data-oriented aspects of requirements, we have integrated real-time specifications into our requirements specification language (/Beeck 92/). These real-time specifications can occur at different levels of abstraction. The lower levels serve as abstract behavioral specifications of families of requirements, which can be reused to define detailed requirements (/GaFr 90/). The other point of interest is the incorporation and reuse of knowledge into the transformation process from requirements to design as described in /Lef 93/ and /BöJa 92/.

References

/Beeck 92/ M. von der Beeck, "Integration of Structured Analysis and Timed Statecharts for Real-Time and Concurrency Specification," Technical Report, AIB Nr. 92−26, University of Technology Aachen, FRG, 1992.

/BöJa 92/ J. Börstler and T. Janning, "Traceability Between Requirements and Design: A Transformational Approach," Proc. COMPSAC '92, Chicago, Ill, Sep 21-25, 1992, 362−368.

/Börs 91a/ J. Börstler, "Integrating Reuse into a Software Development Environment," Proc. of the 1st Int. Workshop on Software Reusability, Dortmund, FRG, Jul 3−5, 1991, 234−238.

/Börs 91b/ J. Börstler, "An Editor for Programming-in-the-Large as a Central CASE-Tool," in german, Proc. TOOL '91, Karlsruhe, FRG, Nov 26−28, 1991, 341−350.

/Börs 93/ J. Börstler, "Programming-in-the-Large, A Language, Tools, and Reuseability," PhD thesis (in german), Aachen University of Technology, FRG, forthcoming.

/Fra 89/ W.B. Frakes, "Panel Session: Information Retrieval and Software Reuse," Proc. SIGIR '89 Int. Conf., Cambridge, MA, Jun (25−28) 1989, 251−256.

/GaFr 90/ A. Gabrielian and M.K. Franklin, "Multi-Level Specification and Verification of Real-Time Software," Proceedings of the 12th International Conference on Software Engineering, Sep 1990.

/GiCo 84/ N. Giddings and T. Colburn, "An Automated Software Design Evaluator," Proceedings of the ACM '84 Annual Conference, Oct 1984.

/HeFr 92/ B. Henderson-Sellers, P. Freeman, "Cataloguing and Classification for Object Libraries," ACM SIGSoft, Vol.7 (1), Jan 1992, 62−64.

/HeNy 92/ M. Henricson, E. Nyquist, "Programming in C++, Rules and Recommendations," Technical Report, M 90 0118 Uen, Ellemtel Telecommunication Systems Laboratories, 1992.

/Lef 93/ M. Lefering, "An Incremental Integration Tool Between Requirements Engineering and Programming-in-the-Large," Proc. RE '93, San Diego, CA, Jan 1993.

/LNW 87/ C. Lewerentz, M. Nagl, and B. Westfechtel, "On Integration Mechanisms within a Graph-based Software Development Environment, Proceedings of the Workshop on Graph-Theoretic Concepts in Computer Science (LNCS 314), Jun/Jul 1987.

/LiHo 89/ K.J. Lieberherr and I.M. Holland, "Assuring Good Style for Object-Oriented Programs," IEEE Software, Vol.6 (5), Sep 1989.

/Linné 53/ K. von Linné, "Caroli Linnaei systema naturae, per regna tria naturae, secundum classes, ordines, genera, species," Thoma Nob. de Trattnern, Vindobonae, 1753.

/Meyer 88/ B. Meyer, "Object−oriented Software Construction," Prentice Hall, 1988.

/Nagl 86/ M. Nagl, "A Software Development Environment based on Graph Technology," Proceedings of the Workshop on Graph Grammars and their Applications to Computer Science (LNCS 291), Dec 1986.

/OmCo 91/ P. W. Oman, C. R. Cook, "A Programming Style Taxonomy," Journal of Systems and Software, Vol. 15 (3), Jul 1991, 287−301.

/Parn 72/ D. L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," CACM, Vol. 15 (12), Dec 1972, 1053−1058.

/PrNe 86/ R. Prieto-Diaz, J. M. Neighbours, "Module Interconnection Languages," Journal of Systems and Software, Vol. 6 (4), Nov 1986, 307−334.

/PrFr 87/ P. Freeman, "Classifying Software for Reusability," IEEE Software, Vol.4 (1), Jan 1987, 6–16.

/Sakk 88/ M. Sakkinen, "Comments on 'the Law of Demeter' and C++," SIGPLAN Notices Vol.3 (12), Dec 1988.

/SPC 91/ Software Productivity Consortium, "Ada Quality and Style: Guidelines for Professional Programmers," Technical Report, SPC–91061–N, Software Productivity Consortium, 1991.

/West 89/ B. Westfechtel, "Revision Control in an Integrated Software Development Environment," Proc. of the 2nd Int. Workshop on Software Configuration Management, Nov 1989, 96–105.

/West 91/ B. Westfechtel, "Structure-Oriented Merging of Revisions of Software Documents," Proc. of the 3rd Int. Workshop on Software Configuration Management, Trondheim, Norway, Jun 1991.