Difference Tools for Analysis and Design Documents

Dirk Ohst, Michael Welle, Udo Kelter
Praktische Informatik, Fachbereich Elektrotechnik und Informatik,
Universität Siegen, D-57076 Siegen,
{ohst|welle|kelter}@informatik.uni-siegen.de

Abstract

This paper presents a concept and tools for the detection and visualisation of differences between versions of graphical software documents such as ER, class or object diagrams, state charts, etc. We first analyse the problems which occur when comparing graphical documents and displaying their similarities and differences. Our basic approach is to use a unified document which contains the common and specific parts of both base documents with the specific parts being highlighted. The central problem is how to reduce the amount of highlighted elements and enable the developer to have a certain amount of control over the changes be selectively highlighted. With regard to tool construction, we assume that software documents are modelled in a fine-grained way, that they are stored as syntax trees in XML files or a repository system and that a version management system is used. By using the features of the data model and the version model we are able to detect and visualise differences between diagram versions, including structural changes (e.g. shifting of a method from one class to another). We further exploit information about the version history delivered by the underlying version management system by highlighting only differences based on structural or logical changes.

Index Terms – fine-grained data model, versions, configuration, design transaction, meta CASE, software engineering environments, differences, UML diagrams

1. Introduction

Software configuration management (SCM) is an indispensable part of high-quality software development processes. SCM is a well established and common practice in the later phases of software development, notably during programming and integration. "Good" maintenance of existing software products is only possible through the usage of such tools. Maintenance depends on the ability to retrieve old versions which are to be corrected or extended. Most of existing SCM systems work only on textual files.

In contrast to that, SCM is less commonly practiced in the early phases, i.e. analysis and design. Modelling a software architecture with CASE-tools leads to a lot of diagrams, notably class diagrams. Their evolution and maintenance require repetitive and tedious intervention. The software architect must ensure that changes for each instance in which a design decision or an architecture pattern is used are reflected by the developed model. Only a few SCM systems provide functions that create versions of analysis or design documents and can visualise differences between the documents. One example is Rational ClearCase in combination with Rational Rose and the Model-Integrator. The creation of the diagrams and the detection of differences between them are distributed over Rational Rose and the Model-Integrator. Rational Rose cannot detect and visualise differences between versions of diagrams. This is only possible by using the Model-Integrator, which utilises a tree representation of the documents to visualise the differences. Besides the document data a lot of meta data are also presented to the developers. The additionally shown meta data reduce the overview of the differences.

Usual algorithms for textual data are not applicable to the detection of differences between versions of diagrams because they do not take the logical structure of such diagrams into account.

Besides the detection of differences, the visualisation is another point which has to be considered. Textual data is only a sequence of lines of text where corresponding blocks are displayed side-by-side. Diagrams cannot be displayed in such a way. One has to distinguish between changes to the layout and changes within the diagram elements. Hence usual methods for displaying differences (displaying the two versions side-by-side) cannot be used. The method used in the Model-Integrator (using a tree representation of the documents) has the disadvantage that the layout is ignored, therefore it is difficult for the developers to bring the original diagrams into their mind and to visually discern the differences.

Another aspect is the number of differences between the

documents. Usually a lot of perhaps small modifications to the documents belongs together. These modifications lead to a large number of differences between versions of this document. The result is that the developer is not able to identify the context of single differences.

This paper presents concepts and an SCM system with a tool for detecting and visualising differences between versions of analysis or design documents and methods to reduce the amount of highlighted differences. One basic assumption is that tools use an internal storage system which applies the concept of fine-grained data modelling. The concepts presented here are applicable independently of whether XML files, a proprietary file format, a relational or an object-oriented database is used. The SCM system presented here is integrated in a repository system known as H-PCTE [8], which is a structurally object-oriented DBMS.

The rest of this paper is organised as follows. Section 2 gives a short overview about other concepts of detecting differences between documents. In section 3 we present the used data model. Our concept to visualise the differences is presented in section 4. The concept is based on the detection of differences, which is described in section 5. Because of the complex visualisation we propose a concept to reduce the complexity in section 6. Furthermore we describe the benefits from the SCM system and fine-grained data model. In section 7 we introduce our version model and discuss how the concept is integrated in our tools and how it could be integrated in other tools. Concluding remarks are given in section 8.

2. Related Work

Usual SCM systems and their difference tools are not well adapted to the needs and circumstances of document management in the early phases. A large number of SCM systems and concepts are available [6], however, virtually all of them only work on files, to be specific, files containing lines of text in pretty-printed format. The difference tools are based on the algorithm of Myers [13] and extensions [19, 7] thereof.

In contrast to this software documents of the early phases are not text, but diagrams (e.g. the different types of UML diagrams). Diagrams are often stored in files, either in proprietary (printable) formats or in XML formats. In the case of a class diagram, each class might be represented by a few lines of text in the file. The order of these sections of text is irrelevant! The position where a class symbol appears in the diagram is explicitly stored in layout data. Therefore, diagram drawing tools can store the sections representing classes or other diagram elements in arbitrary order. As a consequence of this, even small changes in a diagram can cause an editor to a completely reshuffle the file contents and can lead to large number of significant textual differences. Conventional SCM systems are not aware of the logical structure of the document contained in a file.

Structured data can often be represented as trees. Algorithms for finding differences between special kinds of structured documents like LaTeX-files, HTML-files or data in CAD data bases [4, 3] are often based on algorithms solving the tree-to-tree correction problem [2, 21, 18, 17]. The difference algorithms for trees try to find an edit-script which is a sequence of atomic operations that transforms one tree into the other. Only a few of these algorithms can detect shifts across the trees.

The tree-to-tree correction algorithms have been adapted to XML documents [11, 5, 20]. These algorithms try to find similar subtrees of the document and calculate the differences using matched subtrees.

Algorithms for detecting differences between documents exist also in SCM systems which store structured data [10, 1]. These algorithms use persistent node identifiers of the versioned objects to find corresponding objects in both documents.

There are also proposals which describe a version management system for software documents like class diagrams [16] including merge functionality or algorithms for detecting the differences between two class diagrams [22] which are stored in an object structure. Commercial CASE-tools like Rational Rose use individual tools (i. e. ModelIntegrator) to detect and to merge differences between versions of a diagrams. But these tools do not provide a graphical representation of the diagrams and differences.

We are not aware of algorithms which visualise differences between graphical software documents like class diagrams and relate the differences to the corresponding changes. Furthermore it is not possible to exclude differences from being visualised based on versions or to modify the documents containing the visualised differences.

3. Data Model

We use a fine-grained data model to store the documents. Figure 1 shows a class diagram which is used as a running example. In a fine-grained data model all elements of a diagram are modelled as separate objects, the document itself, all classes, methods, attributes and parameters of methods. An example of a meta model of a fine-grained data model for class diagrams is presented in figure 2. Between the object types there exist component relationships, e.g. a document contains classes, a class contains methods and attributes and so on. Further kinds of relationships between object types which represent classes can express inheritance or association relationships between classes. The object attributes contain data which describes the classes, methods, etc. Examples are the name or the type of an element, or layout information belonging to one diagram.

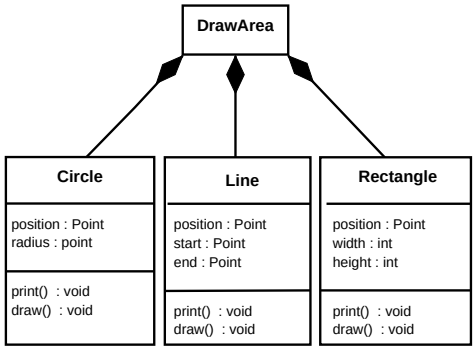


Figure 1. Example of a class diagram

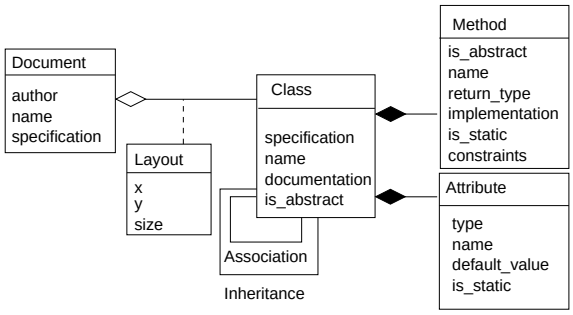


Figure 2. Meta model of a fine-grained data model

Let us look at how to store the class structure given in figure 1. The object structure shown in figure 3 uses the meta model from figure 2. Every component of a class is represented by an object of the appropriate type. Please note that the object structure shown in figure 3 does not contain all objects needed to model the given class diagram. The objects modelling the class `Circle` and the methods and the attributes of the class `Line` have been omitted.

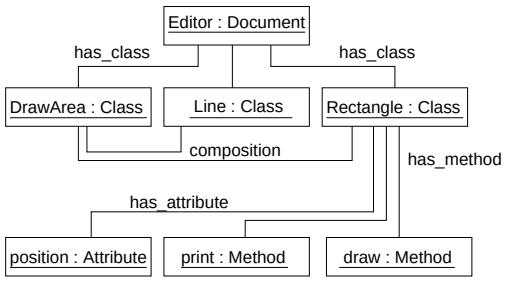


Figure 3. Example of an object structure

4. Visualisation of Differences between graphical Diagrams

In this section we discuss our solution for the presentation of differences between versions of diagrams, e.g. class diagrams, ER diagrams or Petri nets. The documents between which the differences shall be shown will be called *base documents*.

4.1. Presentation of Differences

The basic paradigm of all approaches to show differences between textual documents is to use two columns; each document is shown completely in one of the columns, identical parts in both documents are arranged side-by-side, differing parts are somehow highlighted.

The concept of using two columns works well with most textual documents; it fails if the lines of text are long (more than about 200 characters). It does not work reasonably at all in the case of graphical documents such as state charts, class diagrams, etc.: these documents (more specifically their usual graphical representation) are both "wide" and "tall" and cannot easily be split into sections which are equal or differing, but corresponding.

We have therefore adopted a completely different paradigm: the two documents are shown one *over* the other, they are so to say two transparencies arranged in two layers. The paradigm is intuitively simple, but leads to a number of non-trivial conceptual and technical problems, which we shall address in the following.

4.1.1. The "Unified" Document

Obviously, "common parts" of both documents should be shown only once. The complete picture thus consists of (a) the common parts and (b) the specific parts of each document.

Different colours can be used to distinguish these parts. If we disregard the colours, we see another document; it can be considered the representation of a "unified" (or "mixed") document[1].

The individual differences (or deltas) can be classified as follows:

- differences within a node of the graph, for example, the name of a state in a state chart might differ, or a class may have an additional attribute
- differences in the structure of the graph, for example, nodes or edges are created or removed.

[1] We avoid the notion of a merged document here since merging is often associated with removing conflicts; in that sense, a unified document is not a merged document.

Our unified document will in general show a new graph structure, which is not identical to the graph structure of either base document.

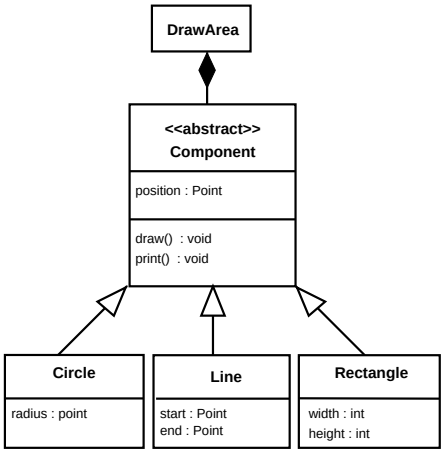
"Common" Parts. The common parts in the unified document should, of course, be as large as possible. Here it is important to distinguish between two sorts of data occurring in graphical documents: layout data and model data. *Layout data* are the positions of nodes and of corner-points of edges, the size of boxes etc., essentially everything which would be irrelevant in a textual representation of the graph[2]. The remaining data are *model data*, they constitute the "logical" document.

It turns out in most cases that the notion of "common parts" should only refer to model data; differences of the layout data are usually not considered relevant[3]. Thus a more sophisticated algorithm for computing the delta between two documents is needed.

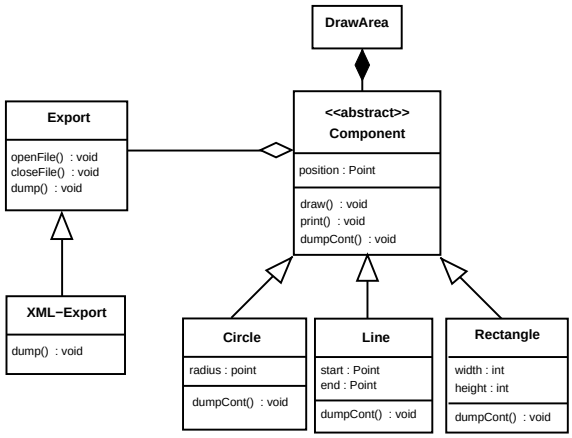
Since the "common parts" of both documents can have different layouts, a new common layout must be found into which both documents can be mapped.

An Example. Figure 4 shows two class diagrams which could result from editing the class diagram shown in figure 1. In figure 4(a), a common super class `Component` has been added and the methods `draw` and `print` have been deleted in the classes `Circle, Line` and `Rectangle`. In figure 4(b), two new classes `Export` and `XML-Export` have been added, and a method `dumpCont` has been added to the class `Component` and in the subclasses.

Figure 5 shows the unified diagram and the differences between the base documents. Different line styles have been used in this diagram to represent different colours (colours would not be visible in black-and-white copies of this paper).



(a) Extended with inheritance



(b) Extended with export functionality

Figure 4. Extending the class diagram of figure 1

4.1.2. Layout of the Common Parts

It is desirable to make the layout of the common parts identical or similar to the layout of one of the base documents, because a developer must trace back the differences represented in the unified document to the base documents. Since modifications to the layout are mostly not substantial, this goal appears to be achievable.

However, even seemingly minor modifications can necessitate significant differences in the layout: assume for example two class diagrams; document 1 contains a symbol for class C; the symbol is positioned between other class symbols and connected via component relationships with some of the other class symbols; in document 2, one of the component classes has been deleted and two new component classes have been created instead. If the layout of document 1 is to be retained, there may not enough space to position the two new component class symbols near class C. Because of this we use the layout of the diagram with the higher number of diagram elements to position the common parts. The specific parts of the other diagram are placed at the border.

Usual algorithms for automatic graph layout do not lead to satisfactory results when applied to class diagrams or similar document types, and they do not preserve the lay-

[2]Leading blanks which are used to visualise the block structure of source programs are a rare example of layout data occurring in textual documents.

[3]If they are considered relevant, the paradigm of transparencies will hardly work any more.

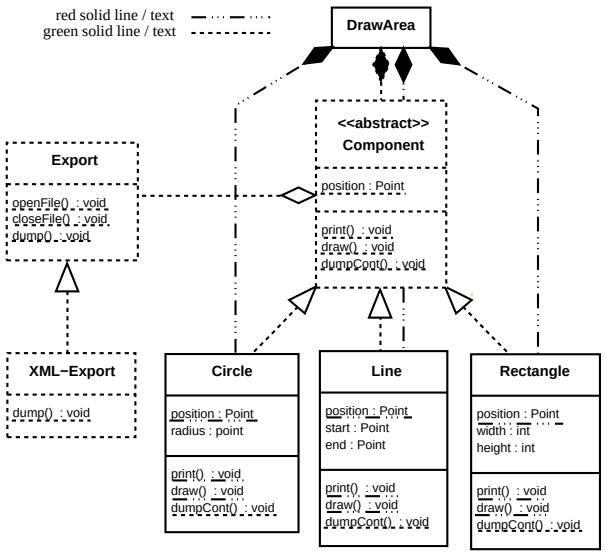


Figure 5. Unified document showing the differences between the base documents

out of one of the base documents. The latter is mostly very desirable since a clear layout may be the result of careful consideration and may have cost significant working time.

The development of suitable layout algorithms is not subject of this paper; in any case, one should not expect to get a perfect solution. We note that it would be desirable to be able to manually improve the initial layout produced automatically.

4.1.3. Intra-Node Differences

Intra-node differences are differences between attributes of two corresponding nodes. We can distinguish two sorts of attributes of nodes:

- single-valued attributes, e.g. the name or properties like {`abstract`} of a class, the visibility of an attribute or operation, etc.
- multi-valued attributes, e.g. the list of attributes or operations of a class, the list of parameters of an operation, etc.

The details depend on the type of diagram and node. Intra-node differences are not considered to be a difference between the node as a whole (then the unified document would contain both versions of the complete node); instead the node as such is considered a “common” part and the internal differences are shown inside the node symbol.

In the case of single-valued attributes, both versions are shown in different colours. In the case of multi-valued attributes, the two lists are compared, a common part is identified and parts appearing only in one version are coloured appropriately.

Examples in figure 5 are the classes `Circle, Line` and `Rectangle` where two methods and an attribute have been deleted and a method has been added. When a class is deleted or added the entire class is coloured (in this example the classes `Export` and `XML-Export`). The same is true for relationships between the classes.

5. Computing the Differences

5.1. Basic Algorithm

As mentioned in section 3 the documents are modelled in a fine-grained way, thus each document is represented as a directed acyclic object graph with a root object. The composition relationships form a spanning tree of this graph. We assume that the root object of the document is unchanged except for its attributes and components.

Algorithm 1 (see below) computes the differences between two documents. It traverses the spanning trees of both documents and tries to find matching objects in both trees. The algorithm works on both documents simultaneously. First the document root objects are put as initial tuple into a set *q*; the root objects are always the first pair of corresponding objects. In general, the set *q* contains the matching objects found so far. In the next step a tuple from this set is fetched and the differences between the attributes of both objects are computed. Then a new object describing the similarities and differences between the currently processed objects is put into the set *result*. The child objects of the current objects are also collected and put into two sets, *set1* and *set2*. Now the matching objects (s. algorithm 2) in these two sets have to be found and inserted as new tuples into the set *q*.

The remaining objects in *set1* and *set2* are candidates for shifted objects inside the document. Moving objects inside a document implies moving their corresponding subtrees. All objects in the sets *set1* and *set2* are put into the sets *shifted1* and *shifted2* respectively. After processing the entire document the two sets *shifted1* and *shifted2* contain all possibly shifted objects including the corresponding subtrees. To detect if an object is shifted the matching objects in both sets have to be found. The matching objects are put as new initial tuples into set *q* and the algorithm is started again. This is performed until set *q* is empty.

After processing both documents the set *result* contains the set union of all objects of the base documents. A persistent unified document is then created based on this set. The unified document conforms to the meta model of figure 2 with some modifications. The unified document must contain additional information about differences between both

documents, e.g. which objects are specific for one document or which attribute values have changed. Therefore the meta model has to be extended by one attribute (difference attribute) at each object type.

Algorithm 1 Computing the Difference between two Documents

```
function diff_objects( Object root1, Object root2 ) : Set
Set q, set1, set2;
Set result;
Set shifted1, shifted2; // possibly shifted objects
Tuple x;
Object obj1, obj2;
// insert document root
q.add ( new Tuple( root1, root2 ) );

// traverse object structure
while  q != ∅ do
   for all x in q do
      q.delete( x );
      // find difference of attributes of current object
      result.add( new Object ( diff_attrs( x ) ) );
      // collect child objects
      set1 = x[1].get_child_objects();
      set2 = x[2].get_child_objects();
      q.add( find_matching_objects( set1, set2 ) );
      // Sets of objects only contained in the first or
      // second document or possibly shifted
      shifted1.addAll( set1 );
      shifted2.addAll( set2 );
   end for
   // Notice: q == ∅
   q = find_matching_objects( shifted1, shifted2 )
end while
// Sets of objects only contained in one base document
result.addAll( shifted1 );
result.addAll( shifted2 );
return result;
```

5.2. Assumptions about the Architecture of Tools

There are two basic approaches for defining that two objects match:

- they have the same object identifier
- they have the same (or similar) contents

The first approach is significantly more efficient, more precise, easier to implement and thus more attractive, but only viable if the architecture of diagram editors fulfils an important condition: the editors must associate a unique identifier with each diagram element, these identifiers are stored in the persistent representation of the document, and they are not changed by the editor when it creates a new revision of the object.

Algorithm 2 Finding matching Objects

```
function find_matching_objects( Set set1, Set set2 ) : Set
Object obj1, obj2;
Set q;
for all obj1 in set1 do
   obj2 = set2.search_obj( obj1 );
   if  obj2 and not q.contains( obj2 )  then
      q.add( new Tuple( obj1, obj2 ) );
      set1.delete( obj1 );
      set2.delete( obj2 );
   end if
end for
return q;
```

These conditions are trivially fulfilled by our own tools because they use an underlying repository which provides unique object identifiers (surrogates) and because they operate directly upon these persistent objects (rather than on transient copies). These conditions can also be fulfilled by tools which work on files in an XML or proprietary format if they implement themselves something which is equivalent to unique object identifiers.

Of course, if a developer copies a diagram element and deletes the original later then the two versions of the diagram element will have different object identifiers. However, these cases should rather be an exception. The same observation applies if we compare versions of a diagram which are not in a predecessor-successor-relationship. In these cases, we have to fall back to techniques which are based on comparing the contents of the objects which represent diagram elements. One has to use heuristics for finding matching objects; it is clear that the heuristic must be document-type specific to produce “good” results. For example in class diagrams two classes can be considered equal if their names and the majority of the operations and attributes are equal or similar.

5.3. Extensions of the Basic Algorithm

When computing the difference between two documents certain local differences should not be considered relevant.

A prime example are layout data (see section 4.1.2); as already mentioned they must be treated in a special way. Layout data are typically stored in specific attributes; there can even be specific objects or relationships to which these attributes are attached. Further examples are author names, certain time stamps, version numbers, etc.; the details depend on the document type and even on the preferences of a developer.

Since we assume a fine-grained data model it is possible to specify the parts of the documents to be ignored by the difference operation as user-defined sets of object, relationship or attribute types. These sets are parameters of the difference operation. In this way, we can use the same difference algorithm for different document types. If data of certain types have been excluded in the computation of the differences, they must often be handled separately in a post-processing step.

6. Filtering highlighted Differences

If a document has evolved over a long period of time and has many revisions, or if the base documents belong to different branches, then the total number of differences between two "distant" versions can be very large. If all differences are coloured in the unified document nearly the entire unified document will be coloured in the worst case, e.g. as shown in figure 5. In this and similar cases, the unified document loses its value for the developer. The number of highlighted diagram elements must be made smaller somehow.

Developers are not always interested in all the differences between documents, sometimes they are only interested in modifications done in context of a specific logical change or in structural changes (e.g. all new classes but not new methods or attributes). Additionally differences due to repeated changes in the same diagram element (e.g. at a class) cannot be distinguished because they are equally coloured. An example therefor is presented in figure 4. The class `Component` is extended in both parts of this figure by some new methods. At first the class is extended by the methods `draw` and `print` due to the introduction of inheritance and secondly the method `dumpCont` is added to realise the export functionality. The problem is that both changes at the class `Component` are highlighted with the same colour so that the developers cannot distinguish between them.

Our proposed solution for this problem is not to colour all differences in the unified document. The differences in which the developer is not interested are painted grey. These grey painted differences are not so eye-catching as the coloured ones. A smaller number of coloured differences improve the overview. The developer can choose which differences of the document should be coloured in the following ways:

1. Highlight logical changes: changes done during an editing session

2. Highlight structural changes: changes belonging to specific diagram elements: e.g. classes (without operations and attributes) or operations or attributes

A combination of this methods is possible.

The information necessary to distinguish single logical changes or single structural changes can be obtained from the version model (explained below) and the data model respectively.

6.1. Version Model

A fine-grained data model makes it necessary to choose a sophisticated version model which is aware of the data structure. File based version management systems do not work properly [14] when they are used to store data modelled in a fine-grained way. Therefore we use a version model based on *design transactions* and *tool transactions* [14]. A design transaction represents a task during the development and combines the results of changes done during several tool sessions, possibly by using different tools. These tools use tool transactions (TTAs) to synchronise concurrent editing sessions. TTAs differ significantly from transactions in conventional data bases. The TTAs have a longer run time and slightly modified characteristics [15]. All modifications made in a TTA lead to the automatic creation of versions of the modified objects and relationships, with the restriction that only one version of each object and each relationship is created in a TTA. Each object and each relationship is versioned independently, thus it has its own version tree. All versions of objects and relationships created inside a TTA are combined into a *configuration*. Each configuration gets a unique identifier. The identifiers of the versioned objects are not changed but the versions can be distinguished by means of the configuration identifier.
We call the configuration of an active TTA a *working configuration*. Every configuration can be used as a baseline for a TTA (*baseline configuration* for short). The working configuration is a direct successor of the baseline configuration.

Three configurations are involved in the example presented in section 4.1.1. One configuration [4] represents the first version of the class diagram (s. figure 1). The other two configurations represent the modifications to this diagram. The first of them combines the versions of objects and relationships due to extending the class diagram with inheritance. The objects representing the export functionality are combined in the second newly created configuration.

6.2. Exploiting Change Histories

The information offered by the version management system makes it possible to distinguish between changes of succeeding editing sessions (configurations). The developer

[4] Usually, several configurations are created during the development of a diagram.

can limit the coloured differences to changes combined into single configurations.

For example the coloured differences presented in figure 4 consist of the changes introduced by adding the class `Component` as super class of the classes `Circle, Line` and `Rectangle` (s. figure 6.2) and the changes introduced by adding the export functionality (extension of the diagram with the classes `Export` and `XML-Export` and the extension of several classes with the operation `dumpCont`). According to the corresponding configuration the highlighted differences of the unified document can separately painted grey.

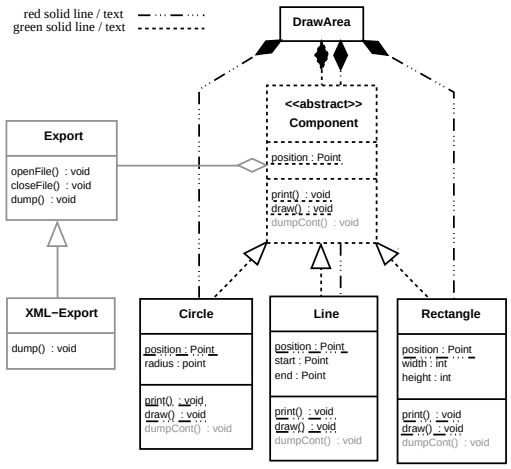


Figure 6. Unified document with unmarked parts of figure 5

6.3. Exploiting a fine-grained Data Model

In some cases a developer is not interested in all differences between the documents. Structural changes of the same kind can be identified separately by using the meta model. For example, when a developer is only interested in changes in the class structure, coloured differences at operations or attributes make the unified document unnecessarily complicated. In our approach it is possible to restrict the coloured differences to parts of the document based in their type, e.g. classes, relationships, methods and attributes or combinations of them. It is also possible to detect shifts between different classes. Only differences at the structural parts chosen by the developer are coloured, all other differences are painted grey.

7. Tool Support

7.1. Constructing Tools

Differences between pairs of documents are normally shown on screen, thus a tool is required for this purpose. For the sake of brevity, we will call such tools *difference tools*.

Most difference tools for text files are at the same time merge tools, i.e. they allow a developer to choose, for each alternative section of the text, between the alternatives and produce another document which reflects these choices. Some difference tools provide additional functions which enable a developer to edit the merged document.

The same approach should be adopted for difference tools for graphical documents. As we have seen above, such a tool should at least offer functions for editing the layout of the unified document, ideally arbitrary editing should be possible. A difference tool would be similar, but not identical to a normal editor for the base documents.

The construction of difference tools can cause a considerable effort, in particular because a dedicated difference tool is needed for each document type. The effort can be reduced by using meta-CASE approaches. This technology further exploits the similarity between normal editors and difference tools. We have adopted this idea; we are using a generic framework for the construction of editors for a range of document types [12, 9].

Persistent Difference Document. One consequence of this approach is that the unified document must be a normal, persistent document. The unified document and the base documents differ only slightly in their meta models (s. figure 2):

- In general, the unified document has weaker consistency constraints. A typical example are relationships with cardinality one. Assume a relationship modelled with cardinality of one (e.g. an exclusive component may be contained in at most one container). If different objects participate in such a relationship in the base documents this relationship must appear twice in the unified document. This would violate the cardinality constraint.

- The meta model for the unified document has to be extended by an additional attribute at each object type. This attribute describes the differences between both versions of one object and the involved configurations. The difference tools must be able to handle the data in this additional attribute.

The basic functionality of displaying a diagram is the same for difference tools and normal graphical editors.

However, difference tools have to colour the diagrams elements appropriately; this feature is new in comparison with normal graphical editors.

7.2. A Prototypical Difference Tool

Our own tools enable a developer to restrict the highlighting of differences on the basis of configurations. Our tools display a list of the configurations which are involved in the difference between the two base documents (s. figure 7). If the versions of the document belong to the same branch, the list contains all configurations between the configurations related to the base documents. If the versions belong to two branches with a common ancestor we use two lists, each one with the configurations between the common ancestor and the configurations relating to one of the base documents. These lists are displayed at the right side of the diagram. The developers can explicitly choose which configuration should be considered. The visualisation algorithm of the tool framework evaluates this information. Furthermore the visualisation algorithm takes the configuration (`confID`) and the contents of the difference attribute (`diffAttr`) into account. Elements which are contained in the first document (diffAttr := "doc1") and were last changed in configuration one (confID := 1) are unchanged in our version history. They are drawn in black (e. g. Class `Circle`). Elements which are contained in the first (second) document and have been changed in another configuration than the first are shown in red (green) if they are not unselected by the developer. In our example this applies to the attribute `position` and the operations `print` and `draw`. The user has unselected configuration 3. So, the operation `dumpCont` is drawn in grey colour, because it is changed in the unselected configuration 3.

7.3. Third Party Tools

The integration of our approach into existing third party tools like ArgoUML or Rational Rose can be divided into two tasks: calculating the differences and displaying the difference document.

Calculating the Difference. CASE-tools often use an XML-like tree structure with node identifiers to store their models. Since we are using a normal but slightly extended persistent document our approach can adopted easily. The data model of the tools has to be extended by one attribute per node to store the additional difference information. The difference operation has to be extended by functionality to traverse and manipulate XML-trees. This can be done without much effort.

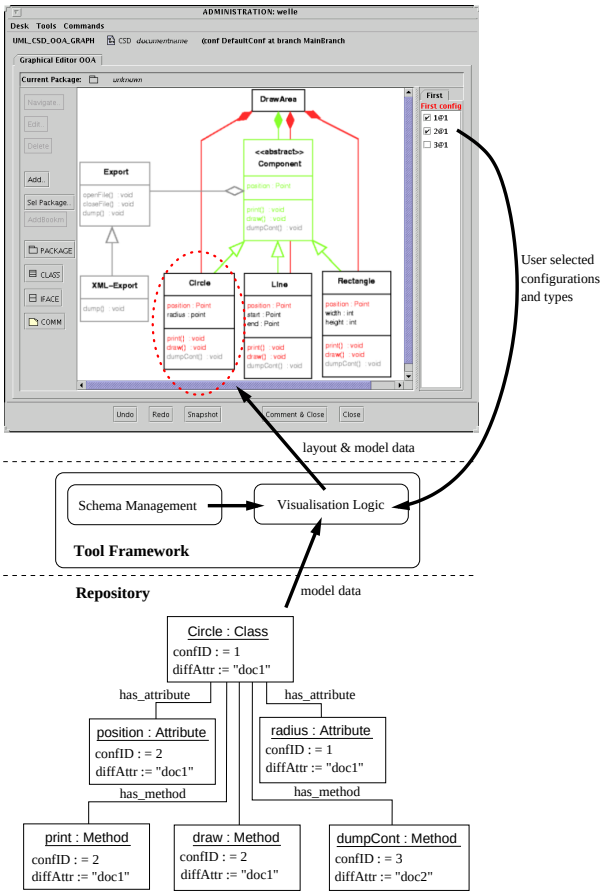


Figure 7. Difference view inside the class diagram editor

Displaying the Difference Document. The tools have to consider the extended data model. Therefore the operations to load and store the data and the transient data structure must be adapted. The graphical editors must support the manipulation (changing colours etc.) of the graphical representation of a node at runtime. Implementing this functionality in an existing tool seems a bit expensive.

8. Conclusion and Future Work

We presented concepts to compute and display changes between graphical documents which are frequently used in the early phases of software development. The differences between two documents are displayed in one unified document. All parts of the unified document which are common to both base documents are painted black. The differences between both base documents are coloured so that they can easily be distinguished.

The unified document is difficult to understand if too

many parts of the document are coloured. We proposed a concept to reduce the number of coloured parts in the unified document. Fine-grained data modelling and a specialised version model allow us to colour only the differences based on structural changes or on changes which occurred during one or several former sessions. The other differences are painted grey.

Future work will address the merging of documents. We are convinced that one can take advantage of configurations to reduce the merge conflicts.

The functionality to create and display differences between UML class diagrams is implemented and working. It has been integrated into the tool environment PISET [9]. Because of creating a new document which contains all information of the common parts and of the different parts it was easy to extend the existing tools to display unified documents.

Acknowledgements

We would like to thank our colleagues with whom we discussed the ideas that led to the present paper and, in particular, M. Monecke.

References

[1] U. Asklund. Identifying conflicts during structural merge. In B. Magnusson, editor, *Proceedings of NWPER'94, Nordic Workshop on Programming Environment Research*, June 1994.

[2] D. T. Barnard, G. Clarke, and N. Duncan. Tree-to-tree correction for document trees. Technical report, Departement of Computing andf Information Science Queen's University Kingston Ontario, Canada, January 1995.

[3] S. Chawathe and H. Garcia-Molina. Meaningful change detection in structured data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 26–37, Tuscon, Arizona, May 1997.

[4] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 493–504, 1996.

[5] G. Cobéna, S. Abiteboul, and A. Marian. Detecting changes in XML documents. In *18. International Conference on Data Engineering (ICDE) San Jose, California, USA, February 26-March 1, 2002*, 2002.

[6] R. Conradi and B. Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, 30(2):232–282, June 1998.

[7] J. J. Hunt, K.-P. Vo, and W. F. Tichy. An empirical study of delta algorithms. In I. Sommerville, editor, *Software configuration management: ICSE 96 SCM-6 Workshop*, pages 49–66. Springer, 1996.

[8] U. Kelter. H-PCTE – a high-performance object management system for system development environments. In *Proceedings COMPSAC Illinois, September 23-25*, pages 45–50. IEEE Press, 1992.

[9] U. Kelter, M. Monecke, and D. Platz. Constructing distributed SDEs using an active repository. In *Proc. 1st Intl. Symposium on Constructing Software Engineering Tools (COSET '99); 17.-18.05.1999, Los Angeles, CA*, pages 149–158, 1999.

[10] B. Magnusson, U. Asklund, and S. Minör. Fine-Grained Revision Control for Collaborative Software Development. In *Proceedings of the ACM SIGSOFT '93 Symposium on the Foundations of Software Engineering, Los Angeles, California*, pages 33–41, December 1993.

[11] A. Marian, S. Abiteboul, G. Cobena, and L. Mignet. Change-centric management of versions in an XML warehouse. In P. M. G. Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao, and R. T. Snodgrass, editors, *Proceedings of the Twenty-seventh International Conference on Very Large Data Bases: Roma, Italy, 11–14th September, 2001*, pages 581–590, Los Altos, CA 94022, USA, 2001. Morgan Kaufmann Publishers.

[12] M. Monecke. *Adaptierbare CASE-Werkzeuge in prozessorientierten Software-Entwicklungsumgebungen*. PhD thesis, Universität Siegen, 2003.

[13] E. W. Myers. An $O(ND)$ difference algorithm and its variations. *Algorithmica*, 1:251–256, 1986.

[14] D. Ohst and U. Kelter. A fine-grained version and configuration model in analysis and design. In *Proc. of the IEEE International Conference on Software Maintenance 2002 (ICSM 2002), 3-6 October 2002, Montreal, Canada*, 2002.

[15] D. Platz. *Ein Werkzeugtransaktionskonzept für Objekt-Managementsysteme als Basis von Software-Entwicklungsumgebungen*. Shaker Verlag, Juni 1999. Dissertationsschrift, Praktische Informatik, Universität-Gesamthochschule Siegen.

[16] J. Rho and C. Wu. An efficient version model of software diagrams. In *Proc. 5th Asia-Pacific Software Engineering Conf., 2-4 December 1998 in Taipei, Taiwan, ROC*. IEEE Computer Society, Dec. 1998.

[17] S. M. Selkow. The tree-to-tree editing problem. *Information Processing Letters*, 6(6):184–186, December 1977.

[18] K.-C. Tai. The tree-to-tree correction problem. *Journal of the ACM*, 26(3):422–433, July 1979.

[19] W. F. Tichy. The string-to-string correction problem with block moves. *ACM Transactions on Computer Systems*, 2(4):309–321, Nov. 1984.

[20] Y. Wang, D. J. DeWitt, and J.-Y. Cai. X-Diff: An effective change detection algorithm for XML documents. In *19th International Conference on Data Engineering, March 5 - March 8, 2003 - Bangalore, India*, 2003.

[21] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal of Computing*, 18:1245–1262, 1989.

[22] A. Zündorf, J. Wadsack, and I. Rockel. Merging graph-like object structures. In *Tenth International Workshop on Software Configuration Management (SCM-10) New Practices, New Challenges, and New Boundaries May 14-15, 2001 Toronto, Canada (a workshop of 23th ICSE 2001)*. http://www.ics.uci.edu/ andre/scm10/, 2001.