Technical Report TUD-CS-2006-5

Difference Detection and Visualization in UML Class Diagrams

Martin Girschick
TU Darmstadt
Department of Computer Science
Metamodelling and its Applications
girschick@informatik.tu-darmstadt.de
http://www.mm.informatik.tu-darmstadt.de/staff/girschick/

Abstract: Analyzing typical diagram life cycles results in the observation that they usually undergo several modifications during the development process. The better the tracking of such changes the more information can be obtained from them. As class diagrams are among the most widely used and most important UML diagram types, this paper investigates the concept of *class diagram diffing* and applies it to several application areas such as software design, implementation documentation, roundtrip modeling, and model driven development. I present the design of a change-detection implementation, which automatically detects differences between class diagrams and visualizes them by the use of color.

1 Introduction

Modern modeling tools already support the creation of source code templates from class diagrams. With the advent of MDA, model transformation and code generation become important research topics. Both technologies bridge the gap from model to source.

Looking at current software development processes shows that after designing the model and starting with the implementation the model is seldom updated to match the current implementation state. There are several reasons for that: Sometimes the design is done by one team and implementation by another. Maintaining the model requires knowledge of the whole system, only simple changes in the source code can be automatically applied to the model. An often practiced technique is to reengineer the class diagram after the implementation phase to match the final version and then check, if it still complies with the modeled version. This comparison may need to be repeated several times during the development process. However, currently no tool for showing the differences between the designed model and the reengineered counterpart exists.

The required model comparison is similar to the process of comparing two versions of source code. Current versioning tools (CVS, SCCS, Subversion, etc.) use a purely text

based approach to find the differences. This generic approach leads to several problems, which are detailed in section 4.1.

This paper presents an algorithm, which compares two class diagrams and visualizes the differences. The advantage of this specialized algorithm is the ability to find similar and not only identical elements and to describe the differences more accurately. It can be used with both manually designed models and reengineered diagrams created from source code.

The next section gives an example for *class diagram diffing*. After that the applicability of a difference analysis is discussed by looking at the lifecycle of a class diagram. This section also details, how differences can be described and visualized. Subsequently the *diffing algorithm* and a prototype implementing it are described. A case study and a list of possible applications of *class diagram diffing* follow. Before concluding, related and future work is described.

2 An example

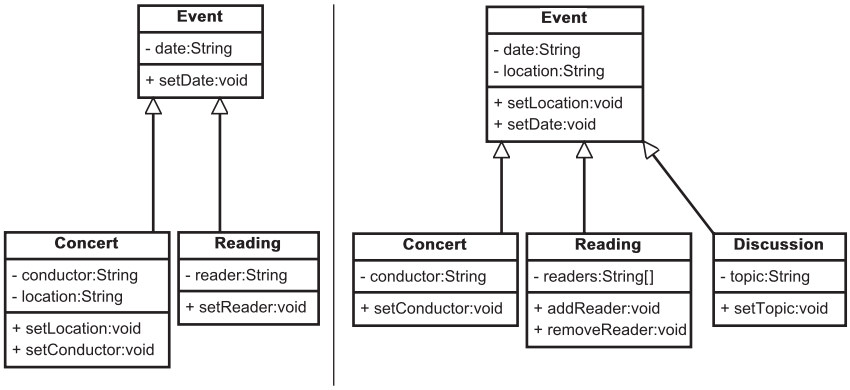

Figure 1: Two sample UML class diagrams

A short example illustrates the application of the algorithm. In figure 1 two class diagrams are shown, the diagram on the right side is a modified version of the diagram on the left. The observed differences are:

- The class `Discussion` with the attribute `topic` and the operation `setTopic` has been added. It inherits from the class `Event`.
- The attribute `location` and the operation `setLocation` have been moved from `Concert` to `Event`.
- The attribute `reader` of the class `Reading` has been renamed to `readers` and converted to an array.

- `setReader` has been removed, `addReader` and `removeReader` are added.

The aim of the algorithm is to automate the analysis process and to generate a list of transformations, which converts the left to the right diagram. Merging both diagrams and coloring the modified elements results in the diagram shown in figure 2. The HTML report, which is also automatically produced by UMLDiff$_{cld}$, includes a list of the transformations. A green background shows moved elements, blue denotes new elements, and orange removed elements. As can be easily verified, all observed differences in this example have been correctly identified.

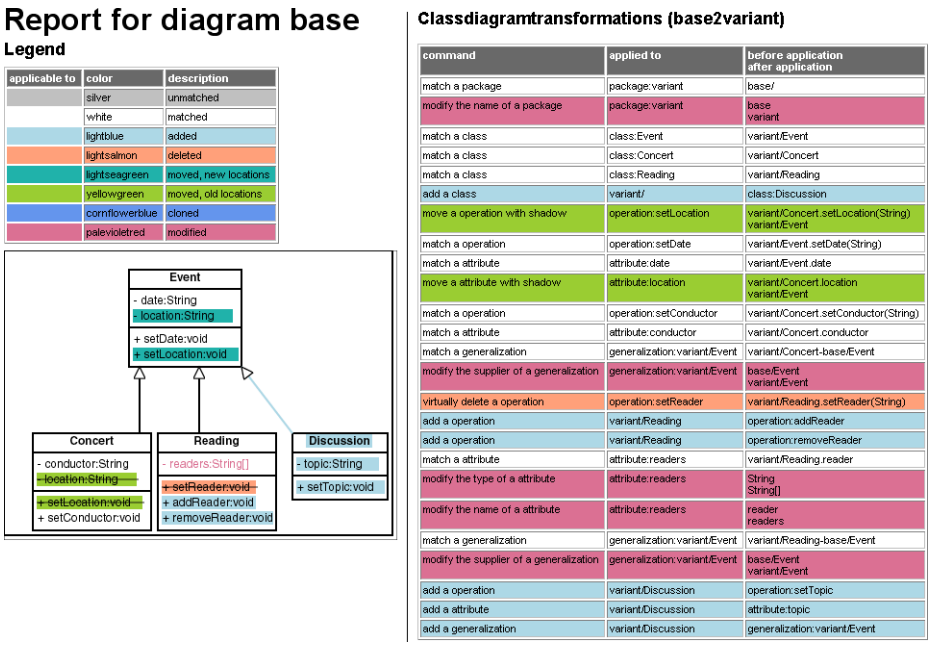

Figure 2: HTML report including the combined and colored diagrams

3 Analysis

3.1 The lifecycle of a class diagram

Class diagrams are used throughout the software development process. During the design, class diagrams are created from scratch or taken from previous releases. After that, the diagrams are incrementally refined. Usually diagrams do not only exist in several revisions but also in different variants. A difference analysis can compare the branches and visualize the differences. This is useful, when the branches have to be merged again.

In iterative processes the phases are repeated several times. Often between iterations the

software is released for testing. By analyzing the differences to previous releases both developers and clients can check which parts have been changed and whether the system still complies with the design.

Finally, class diagrams serve as a good starting point for documenting a system. By using color to visualize differences between a previous and the current release the changes on the model are compactly documented. For instance "unstable" parts of the system, which change extensively, can be identified.

3.2 Describing the changes

The modifications applied to a class diagram can be described using basic transformation operations. By reapplying a list of such transformations to an old version of a diagram the new version can be recreated. The following transformation operations have been identified:

add To add new modeling elements to the diagram

delete To remove elements

rename To rename elements (where applicable)

move Moves an element (and optionally its parts) to another container (A container is a diagram element, which contains other elements, called parts.)

clone Copies an element to another container

modify property Changes a property (e.g. type, visibility, stereotype, multiplicity, etc.)

3.3 Visualization

Several visualization techniques can be applied to provide a better presentation of the differences between class diagrams. Color-coding the different transformation operations (e.g. newly added elements are colored green, removed elements are red) and presenting them in a tabular report is one of them. Another technique is the usage of animation to present the recorded changes. This is useful to show the source and destination of moved elements. The color coding can also be transferred to class diagrams as has been shown in section 2.

4 Calculation of differences

Two different approaches can be used to detect differences in data structures.

Online By adding functionality to a modeling tool it is possible to record the changes while they occur. The advantage of this approach is that not only the change is detected but also how the change is produced. This helps to alleviate some of the analysis problems, which can occur when comparing class diagrams (see subsection 4.2.1). There are a few minor disadvantages: By recording all changes on a model a plethora of data is collected. This results from the fact that some changes are only temporary until the model reaches the next "savepoint". In addition tracking the changes "online" is only applicable to one model. This prevents analysis of models, which come from different sources (e.g. reengineered models).

Offline During the construction of the class diagram "snapshots" are taken (similar to the procedure of text based versioning tools). Later a separate tool compares them. With this technique it is possible to compare class diagrams, which have been created not only from snapshots but also from different branches or automatic diagram generation tools.

The prototype, named UMLDiff$_{cld}$[1], uses the *offline approach* to compare two class diagrams. It creates a list of transformation operations which are then displayed in a colored report table and a colored UML class diagram. The numerous applications of this approach are detailed in section 6. The prototype uses XML files as input. The chosen format contains only the information necessary for model analysis and is not yet compatible with XMI, although support for XMI or other model interchange formats is planned. A description of the used XML schema is available in [Gir02].

4.1 Comparing general data structures

By choosing the *offline approach* and defining an appropriate data structure the difference analysis has many similarities to traditional ways of comparing two data structures. Several algorithms and tools have already been developed for this purpose. Widely known in the Unix world is `diff`[2], which uses a line based algorithm to find the difference between two text files. As this approach has no knowledge of the used data structure it cannot identify specific differences and is therefore unusable to compare structured data.

Several algorithms have also been designed to work with XML files. As an XML document can be seen as a tree structure, graph algorithms have been considered as well. The *Tree-to-Tree-Problem* described in [Sel77] is a well known graph problem, which determines the differences between two trees. In [HO82] five algorithms are described and compared. These algorithms have several restrictions: The limited alphabet of the nodes makes them unusable for the described problem. Additionally those algorithms only try to match identical and not "similar" parts, which is also a requirement for the difference analysis of class diagrams. A now retired project at IBM/Alphaworks created a tool named XMLTreeDiff[3], which adapted the functionality of Unix-diff for XML files. Changed and

[1] *cld* stands for class diagram.

[2] `http://www.gnu.org/software/diffutils/diffutils.html`

[3] `http://www.alphaworks.ibm.com/tech/xmltreediff`

removed XML attributes and elements are colored to emphasize the differences. Unfortunately, this algorithm cannot deal with moved elements either.

The change-detection algorithm X-Diff presented in [YW] deals with another XML issue: Siblings within an XML structure are usually not ordered. This algorithm uses an unordered model for comparing siblings and is therefore more accurate. A disadvantage is that this algorithm does not detect and describe elements, which have been moved to a different location in the XML tree.

Finding elements, which have been moved to a different location within the XML tree, is possible with the method used in [CTZ01]. However, this works only with identical elements and not similar items.

Finally the best suited algorithm is the one presented by Sudarshan S. Chawathe und Hector Garcia-Molina in [CGM97]. In their approach, the matching of the elements is reduced to a graph problem, which tries to find the minimal cost for matching a bipartite graph[4]. A weighting function is used to determine the probability of a correct match. This approach is very good for generic data structures but if the data structure is known, better results can be accomplished by writing specific algorithms.

4.2 Comparing class diagrams

The previous subsection showed that generic *diffing* algorithms cannot effectively be used to compare class diagrams. The algorithm described in the remaining section is a combination of ideas from the generic algorithms and some self developed techniques.

The following data structure details have to be taken into account and can be exploited when designing the algorithm:

- The type of a class diagram element doesn't change (A class can't change to an attribute).
- The order of classes, and the order of attributes and operations within a class is irrelevant.
- The order of the parameters of an operation is relevant.
- The hierarchy of elements is well defined (e.g. packages are containers, their parts are classes, parameters are attached to operations).
- Classes, attributes, operations and parameters have a name.
- The name (for operations this includes the signature) is unique within that container (A container is the enclosing data structure element.).
- Some modeling tools attach a unique ID to every diagram element. If the ID remains constant even across multiple versions and when the element is moved to a different

[4] A bipartite graph consists of two node sets, edges only go from one set to the other.

container it can be used for the matching process. This ID-check is restricted to diagrams, which originate from the same source.

- Stereotypes cannot be changed (they are deleted and added instead, which is semantically more sensible).
- The supplier of a generalization can only be changed to another superclass (otherwise the generalization has to be deleted and - with the new supplier - added again).
- Associations can only be moved along the inheritance hierarchy.

Exploiting such information for the design of the algorithm allows more semantic flavor to be captured in diffing, as opposed to a plain syntactic approach not taking any of the element properties into account. In contrast to the generic algorithms from the previous section, which only detect operations like *delete* or *insert*, this algorithm also tries to find *moved* and *renamed* elements correctly.

Usual diffing algorithms only try to match identical parts. The class diagram diffing algorithm also matches *similar* elements. Often an element within a class diagram is only changed slightly from one release to another (e.g. the type of a parameter is changed to a superclass). This prevents generic algorithms to identify the changed parts correctly.

Some modeling tools attach additional (non UML) information to diagram elements, for instance visualisation hints like color or position. These additional properties are also analyzed by UMLDiff$_{cld}$ and produce additional `modify` transformation operations.

The algorithm takes two class diagrams as input. From now on the first diagram is called the *base diagram* (short d_b) and the second *variant diagram* (short d_v). The algorithm produces a list of transformation operations, which transform d_b to d_v. An element within the base or variant diagram is called e_b or e_v respectively.

The following section gives a detailed description of the algorithm. In addition figure 3 gives a short overview of the algorithm using pseudo code.

4.2.1 The UMLDiff$_{cld}$ algorithm

The XML data structure is compared level by level. First all packages, followed by all classes, followed by the four element types within all classes. Each level produces a set of transformation operations. After all levels have been compared, the collected transformation operations are optionally passed on to a supplementary analysis phase. See section 7 for the different applications of this phase.

On each level a comparison function called `findMatch` helps to find the best match for each element. For that *evaluation functions* are used, which measure the "quality" of a match. Some functions are common to all element types, others are specific to a subset of them. A "breadth first" approach was chosen because elements are often moved to other containers (classes to other packages, operations to sub- or superclasses). Another advantage is that the effect of wrong matches is minimized, because only after all classes have been matched the elements within those classes are considered. With a depth first

approach each class and its parts would be handled separately. This degrades the quality of matches towards the end of the matching process, because better matching items might have been already matched before. Breadth first makes the matching more precise in the following levels.

Each element from the variant diagram is visited and the most appropriate match in the base diagram is determined. The reason for starting in the variant diagram is that it usually contains more information to be analyzed. Another reason is the `clone` operation, which is used to match one element in the base diagram with two or more elements in the variant. From each match a list of transformation operations is generated. This contains both structural changes (e.g. an element is moved to a different location) and modifications (e.g. changed name or visibility). If no match is found, the element is considered to be *new* and an `add` operation is generated. Unmatched elements from the base diagram generate a `delete` transformation operation.

The operations are then applied to the base diagram. Deleted elements are only marked as deleted, because the parts might have been moved to other elements (e.g. a package has been deleted, but the classes have been moved to different packages). The parts of newly added elements are not created yet, because they may have been moved there from other locations. The updated diagram is now used on the following levels for comparison. The specifics of the different levels are detailed further in the following list:

1. **packages** Packages are matched by their name and their parts. If a match is found the name of both packages are compared. A changed name results in a `modify` transformation operation.

2. **classes** If for one class in the variant diagram several classes exist in the base diagram, the `clone` operation is used to create the newly found classes. This can be useful when a class is split into two or a class implementing an interface is created. Classes, which have been moved to a different package, are represented by a `move` operation. Other class specific changes (e.g. name or type) are described by `modify` operations. When applying the `move` operation the parts of the class are moved as well.

3. **generalizations** An evaluation function takes into account that a generalization is usually only changed to another sub- or superclass within the same inheritance hierarchy.

4. **attributes** Matching attributes correctly is difficult as an attribute has only a few properties. Even when an online approach is used to detect changes it is usually not known, whether the programmer renamed an attribute or simply reused an existing attribute for something else by changing its name.

5. **associations** Associations normally correspond to attributes within the class. As those had been examined before and any moved attributes should have been detected correctly, finding the moved associations is simple.

6. **operations** Finally, the operations and their parameters are examined. Here we want to honor the fact that quite often new operations are created, which have the same

name as existing ones. A `clone` operation is used to show that the newly added operation is similar to an existing one. The same type of operation is used to create new operations, which have the same name as an operation in a superclass. This is especially useful when an interface is implemented and all its operations are cloned into the implementing class. To match an operation, its signature is included by the evaluation functions. The parameters of the matched operation are checked and - if changed - appropriate transformation operations are created.

4.2.2 The comparison function

`findMatch` is called for every element e_v in d_v. It first finds the corresponding container in d_b. For a class this means, that the search for a matching class is started in the package, which has been matched on the previous level. All elements within that corresponding container are *weighted* against e_v. This is done by calling the appropriate evaluation functions for that element type. Each functions returns a value between -16 and $+16$ representing the likelihood of the match[5]. Negative values show an unlikely match and high positive numbers a good match. 0 is returned when the function cannot decide whether a match exists or not. In future versions additional evaluation functions can be added to improve the matching process or a manual process can match elements by asking the user. All evaluation functions are summed up and the highest value represents the best match. If the sum exceeds a given limit, the match is taken. If the sum is lower than the limit, the search is broadened to all elements in d_b which are on the same level. If still no match can be found, it is considered to be *new*.

4.2.3 The evaluation functions

The characteristics of class diagram elements are compared by the evaluation functions. The following list describes some of those characteristics. The first items are "good" characteristics and towards the end of the list the "weaker" criteria are described.

location In most cases the matching element will be in the corresponding container, so this place is searched first. Looking at well known refactoring steps items are often moved to sub- or superclasses, so "nearby" locations are preferred.

name Most elements have a name, if this is identical, the evaluation function returns a high value, similar names are considered as well.

stereotype Often stereotypes are used to describe global aspects, which are in many cases only available once and therefore are a good matching criteria.

type The type is also a good matching characteristic. Again, if the changed type is "closer" in the inheritance hierarchy it leads to a higher value.

[5] 16 was chosen arbitrarily to define the range of the values to be returned by the evaluation functions.

```
void umldiff(Diagram db, Diagram dv)
{
  foreach (package pv in dv) analyse(pv,db);
  // mark unmatched packages as deleted
  foreach (unmatched package pb in db) markDeleted(pb);

  foreach (class cb in dv) analyse(cb,db);
  foreach (unmatched class cb in db) markDeleted(cb);

  foreach (generalization ...
  foreach (attribute ...
  foreach (association ...
  foreach (operation ...
}


// first find matching element, then
// find and apply transformations for this element
void analyse(Element ev, Diagram db)
{
  matchingElement = findMatch(ev,db);
  transform(matchingElement,ev);
}


// find appropriate match for ev
// first search previously matched container
// then search other containers
element findMatch(Element ev, Diagram db)
{
  variantContainer = getContainer(ev);
  baseContainer = getMatch(variantContainer);
  maxElement = null;

  // search in matching container
  foreach (element eb in baseContainer)
  {
    if (weightAgainst(eb,ev) > value(maxElement))
      maxElement = eb;
  }
  // check, if match exceeds minimum limit
  if (value(maxElement) < limit)
  {
    // broaden search by searching other containers
    // which have the same type and therefore are
    // on the same level (siblings)
    otherContainers = siblings(baseContainer);
    foreach (element eb in otherContainers);
    {
      if (weightAgainst(eb,ev) > val(maxElement))
        maxElement = eb;
    }
    // check against limit again
    if (value(maxElement) < limit) return null;
  }
  return maxElement;
}


// find transformation operations
// and apply them to the base diagram
void transform(Element eb, Element ev)
{
  if (eb equals null) // no match found
  {
    addOperation(ev);
    return;
  }
  else if (eb already matched) cloneOperation(eb,ev);
  if (eb differs from ev) modifyOperations(eb,ev);
  if (container(eb) != container(ev))
    moveOperation(eb,ev);
}
```

Figure 3: Pseudo code for the change analysis algorithm

signature The signature is a very important criteria for operations. If the operation has many parameters, it is more suitable than the name of the operation.

parts In both container types, packages and classes it is quite useful to also look at their parts to identify matching elements.

predecessor Usually the order of attributes and operations is not considered, but it might be helpful to take it into account to differentiate between very similar elements.

4.2.4 Analyzing the Example

Given the example from section 2 the algorithm performs the following steps when analyzing the attribute $\texttt{location}_v$ in the class $\texttt{Event}_v$:

1. The previously matched container is determined. In this case, the class `Event` of the variant diagram ($\texttt{Event}_v$) was matched to the class `Event` in the base diagram ($\texttt{Event}_b$).
2. All attributes of $\texttt{Event}_b$ are weighted against $\texttt{location}_v$.
3. No match exceeding the limit was found (all attributes differ too much).

4. The search is broadened by considering all classes in the base diagram.
5. The attribute `location`$_b$ in the class `Concert`$_b$ yields the maximum weight because the name is identical and the class is close in the inheritance hierarchy.
6. Both attributes are matched, the appropriate transformation operation (in this case "move") is generated and applied to the base diagram.

4.2.5 Visualization

By applying the transformations to the class diagram d_b and coloring the changed elements appropriately the differences between two diagrams can be visualized effectively. A sample diagram was already shown in section 2. The following background colors had been used to represent the different transformations: **Blue** is used for added or cloned elements, **green** presents moved elements, and deleted elements have an **orange** background.

The text of modified elements (or the line of associations and generalizations) is colored **violet**. The generated diagram can be viewed using an SVG[6] viewer or a browser plugin[7]. Additionally an HTML table report is generated, which lists the transformation operations (again with a color coded background). See section 2 for a sample report.

4.2.6 Complexity

An element in the variant diagram is only analyzed once and compared to all elements of the same type in the base diagram. If p is the number of packages and c the number of classes, then the worst case is $p_v \times p_b + c_v \times c_b$ comparisons for the packages and classes. The same calculation holds for the elements within the classes. As the search is first limited to a certain container and elements which have already been matched are not taken into account anymore the average case is much better.

4.2.7 Evaluation of the prototype

The chosen *offline approach* is only practical when the compared snapshots do not differ too much. For instance when both the name and the signature of an operation is changed the evaluation function cannot easily match the operation. If this matching already fails on the package or class level, it will fail completely because the elements within these containers cannot be matched. Also, elements that contain only a few properties may potentially confuse the matching process if too many similar elements exist. Furthermore, in most cases renamed elements cannot be matched, because the new name may differ immensely from the previous one. This means that enough clues in the model have to exist, to make matching possible. One suggested solution is to make snapshots more often to reduce the differences. Still, the approach returns better results than generic algorithms and by adding more evaluation functions the recognition rate can be enhanced further.

[6] Scalable Vector Graphics. An XML based vector oriented graphic language.
[7] `http://www.adobe.com/svg/`

5 Case study

To show the practical value of UMLDiff$_{cld}$ it has been applied to a software system which has been developed at our university during a practical for Software Engineering. The client-server architecture is used for two player board games which can be played across the network using RMI. The source code has been stored in a CVS repository from which snapshots can be extracted.

Two snapshots, which are approximately three months apart, have been selected. The class diagrams have been reengineered from the source code using Borland Together Control Center. During the three months some packages have been remodeled from the ground up, others underwent only minor changes. The whole system consists of 50 java classes in ten (partly nested) packages. As the current prototype of UMLDiff$_{cld}$ only analyzes one package at a time, three java packages have been selected for further (separate) analysis.

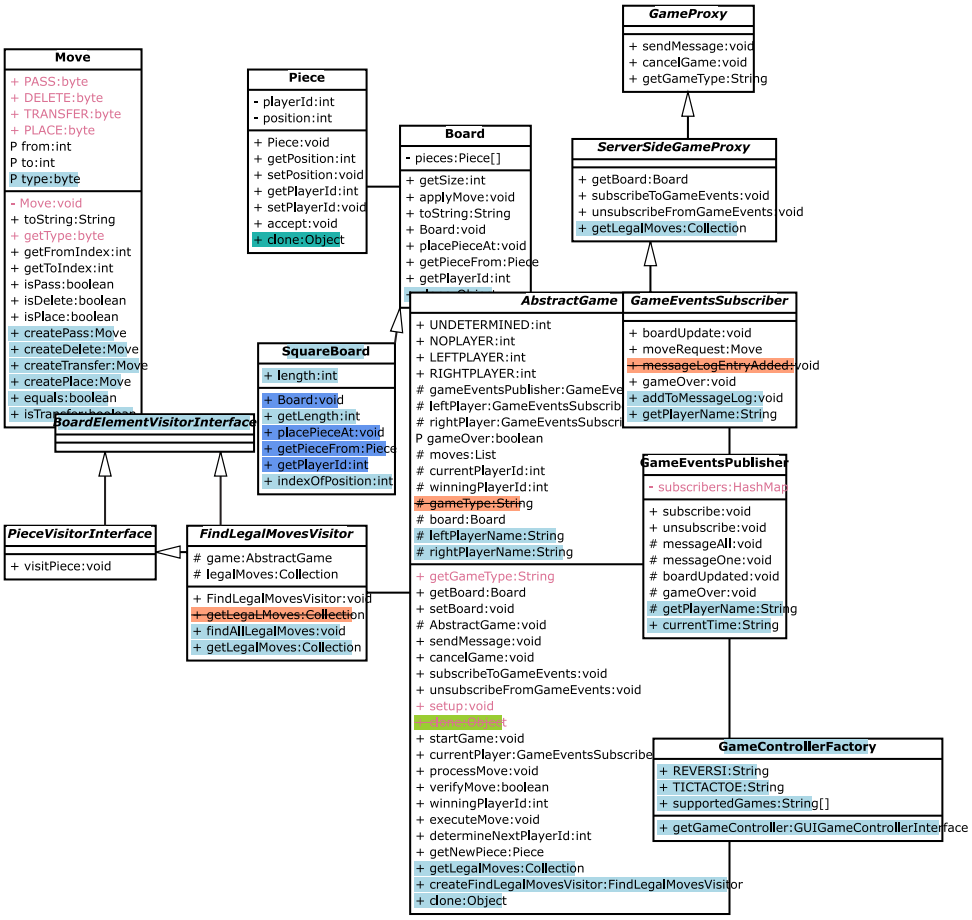

Figure 4: Colored class diagram showing difference of the `games` package of `platform2play`

The changes applied to the `client` package where only minor. Mainly classes and operations have been added and the differences where successfully detected by UMLDiff$_{cld}$.

Figure 4 shows the merged class diagram of the `games` package. The elements with a light blue background[8] (bright gray in black and white reproductions) have been added to the model, elements with pink (or gray) text have been modified (mostly changes of the visibility, e.g. from `public` to `protected`). The colored diagrams gives a good overview of the changes, which have been applied to the package.

The third package `gui` underwent heavy refactoring. The resulting difference diagram shows many erroneously detected move operations. This clearly shows the limitations of the algorithm if the number of differences is too high, only a different approach which uses online analysis (see section 4) would be able to reveal the differences correctly.

6 Applications

The previously described algorithm can be used for several purposes. The following scenarios show a few areas, where class diagram diffing can be applied:

- The different revisions of class diagram during the design phase can be compared to monitor the progress. Stable areas can be seen, repeatedly changed elements may require additional attention. Both designers and clients can benefit from this by easily finding even small changes.
- Developers compare a new revision with a previous one to detect changed programming interfaces or newly added functionality. This is both useful for checking provided frameworks and self developed systems.
- The specification of a system (the design model) can be checked against a generated diagram, which was reengineered from the implementation. Currently this is only done "by hand" if at all.
- Different variants of models can be compared to detect differences. For instance when a product is developed for several platforms and similarities or differences have to be analyzed. By automating this process different branches can be simplified or combined which lowers maintenance cost and complexity.

7 Related and future work

The work presented in this paper is based on the diploma thesis titled "Erkennung und Analyse von Unterschieden in Klassendiagrammen und Sequenzdiagrammen" [Gir02] which additionally analyses differences between sequence diagrams.

[8] The description of the used colors can be found in figure 2.

Some existing algorithms are limited to a specific language. For instance, JDiff[9] compares two Java packages and creates an HTML report containing the API differences between these two releases. The approach relies on syntactic differences and cannot identify moved or renamed elements.

Recently Zhenchang Xing and Eleni Stroulia investigated the evolution of specific classes during their lifecycle [XS04]. The algorithm used to detect the changes is – coincidentally – also named UMLDiff. Unfortunately, the paper does not go into detail how the algorithm works, making it impossible to compare it to the work presented here.

In [OWK03] Dirk Ohst, Michael Welle, and Udo Kelter also describe an algorithm which detects and visualizes the differences between versions of UML diagrams, but they rely on using unique identifiers to match similar elements. Algorithms which work without those identifiers are not within the scope of their paper. As has been mentioned in section 4.2 unique identifiers are not always present and limit the application of diagram diffing. For instance, comparing diagrams from different sources cannot be compared.

A recently published paper by Udo Kelter, Jürgen Wehren, and Jörg Niere [KWN05] refrains from using unique identifiers and tries to match elements by comparing subtrees. This approach shows similarities to UMLDiff$_{cld}$ but is not especially tailored to class diagrams. Without the additional information available in class diagrams the results might be less accurate but the more generic approach is easier adaptable to other diagrams which can be modelled with trees.

Some of the discussed algorithms are not freely available and adapting the generic algorithms to a class diagram structure is not easily accomplishable. All approaches show similarities therefore their applicability to other diagram types and the possibility to combine them should be further investigated.

Another concept, which is applied to source code, is *refactoring*. By applying well-defined operations the source code is transformed to an easier maintainable variant which is semantically equivalent. By extending UMLDiff$_{cld}$ refactoring steps (e.g. push up, pull down, etc.) can be automatically detected and visually presented. In addition, the continuous analysis of model differences could reveal recurring change patterns, i.e., candidates for new refactoring steps.

One of the main principles of model driven software development (MDSD) is to replace the code by the model as the primary artifact. Model analysis and model to model transformation are important research topics within this area. Concepts like *class diagram diffing* can help to visualize model differences, which is useful for model transformation debugging and version controlling. Looking at the versioning process reveals that with MDSD code diffing is replaced by model diffing.

[9] `http://sourceforge.net/projects/javadiff/`

8 Conclusion

Although the UML still contains many ambiguous constructs, the defined syntax for class diagrams is sufficient for detecting differences between models. The presented algorithm is highly tailored towards UML class diagrams, which has several advantages: The knowledge of the data structure makes the matching more precise and the found differences are described using high level transformation operations.

Compared to generic change-detection algorithms the presented approach promises to return better results. Adapting the techniques for other hierarchical data structures can easily be accomplished by designing evaluation function which model the restrictions of a given data structure.

The concept of *class diagram diffing* extends naturally the textual change detection to model information. Integrating the concepts of UMLDiff$_{cld}$ into modeling tools can help to visualize model differences, and improves the software development process.

References

[CGM97] Sudarshan S. Chawathe and Hector Garcia-Molina. Meaningful Change Detection in Structured Data. In *ACM SIGMOD International Conference on Management of Data*, pages 26–37, Tucson, Arizona, May 1997.

[CTZ01] Shu-Yao Chien, Vassilis J. Tsotras, and Carlo Zaniolo. XML Document Versioning. *SIGMOD Records*, 30(3), 2001.

[Gir02] Martin Girschick. Erkennung und Analyse von Unterschieden in Klassendiagrammen und Sequenzdiagrammen. Diploma thesis TU-Darmstadt, 4 2002.

[HO82] Christoph M. Hoffmann and Michael J. O'Donnell. Pattern Matching in Trees. *Journal of the ACM*, 29(1):68–95, 1982.

[KWN05] Udo Kelter, Jürgen Wehren, and Jörg Niere. A Generic Difference Algorithm for UML Models. In *Proceedings of the Software Engineering Conference 2005*, Essen, Germany, 2005.

[OWK03] Dirk Ohst, Michael Welle, and Udo Kelter. Differences between versions of UML diagrams. In *Proceedings of the 9th European software engineering conference*, pages 227–236, New York, NY, USA, 2003. ACM Press. isbn = 1-58113-743-5,.

[Sel77] S. M. Selkow. The tree-to-tree editing problem. *Information Processing Letters*, 6(6):184–186, December 1977.

[XS04] Zhenchang Xing and Eleni Stroulia. Understanding Class Evolution in Object-Oriented Software. In *12th International Workshop on Program Comprehension (IWPC 2004), 24-26 June 2004, Bari, Italy*, pages 34–45. IEEE Computer Society, 2004.

[YW] Jin-Yi Cai Yuan Wang, David J. DeWitt. X-Diff: A Fast Change Detection Algorithm for XMLDocuments. `http://www.cs.wisc.edu/niagara/papers/xdiff.pdf`.