A Basic Formalism for Systematic Software Evolution

Tom Mens
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2 – 1050 Brussel
(+32) 2 629 3474
tommens@vub.ac.be

ABSTRACT

In this extended abstract, we present reuse contracts as a simple but powerful formalism for dealing with software evolution in a systematic way.

Keywords

Reuse contracts, software evolution, conflict detection.

1. MOTIVATION

Evolution is omnipresent in software engineering due to constantly changing requirements, technological advances, bug fixes, software maintenance, new insights in the problem domain, and so on. Moreover, changes occur at every level of detail, and in every phase of the software life-cycle, ranging from requirements specification over analysis and design to low-level code.

Many problems can arise during evolution. When a software component evolves, other components that depend on it may give rise to evolution conflicts when they rely on assumptions about the evolved component that are not valid anymore. This problem frequently occurs when upgrading to a new version of the software system, and is directly related to the problem of *change propagation*. In order to maintain the consistency of the system, evolution of one system part often requires the dependent parts to be changed as well. In this way, a simple change may propagate through the entire system. This problem is also know as the so-called *ripple effect*.

In order to solve the problems above, we need to find out which components are affected when a certain software component is changed, and also how they are affected. This is not a trivial task, due to the lack of a systematic way to deal with evolution. Indeed, probably the most widespread way to deal with evolution of components is by directly editing the component. With this kind of evolution the developer changes a component to satisfy new requirements without maintaining or requiring any form of link with the previous version. Consequently, it is very difficult to track how changes in a component affect dependent components. For this reason, we propose a formalism to deal with evolution in a more systematic way. To achieve this, we keep an *explicit link* between the original component and the evolved component, and formally document *how* each component evolves. In this way, we can find out precisely which modifications to a component give rise to evolution conflicts in dependent components.

In order to provide such a formalism for systematic evolution, we need to make some important design considerations. For example, a balance needs to be found between formality and ease of use. On the one hand, the approach should be formal to allow reasoning about evolution conflicts in an unambiguous way, and so that it can form the basis of automated tool support. On the other hand, it should be simple enough, so that it will be adopted by software engineers. These design considerations lead us to the formalism of *reuse contracts* [3, 6] with which we tried to satisfy both demands. Reuse contracts focus on the essential aspects of software components only. In this way, they provide insight in the core behaviour of the system without burdening the developer with unnecessary details.

Much of the inspiration for developing reuse contracts was drawn from practical experience in developing object-oriented frameworks [2]. These and other experiments suggest that documentation of how components depend on other components, and how components evolve, allows an accurate estimation of which components are affected by a change, and also how they are affected. While this approach is clearly related to the research on *software change impact analysis* [1], the main contribution of reuse contracts is that they also document *how* components are changed, and to which conflicts in oher components this can lead. As a result, it becomes easier to solve evolution conflicts, in a semi-automatic way.

2. REUSE CONTRACTS TERMINOLOGY

In the context of evolution, the essential idea of reuse contracts is that component evolution takes place on the basis of an explicit *contract* between the *provider* of a component and an *evolver* that modifies this component.

To introduce more reuse-contract specific terminology, let us take a look at an illustrative example (Figure 1). Suppose that during the design phase we want to express the fact that an existing `Document` interface specification (the provider) evolves into a `BrowsableDoc` interface by adding a `resolveLink` operation and refining the `mouseClick` operation. From an implementation point of view, the only way to do this is by editing a `Document` class and making the changes directly in the code, or by creating a copy of the `Document` class, called `BrowsableDoc`, and making the changes in the copy. Both approaches have the disadvantage that there is no explicit link between the original component and the evolved component. As a result, when the original component is changed afterwards (which is even impossible with direct code editing), the evolved component will never know about this, and cannot be upgraded to cope with these changes. This often leads to evolution problems such as *version proliferation* and unanticipated evolution conflicts.

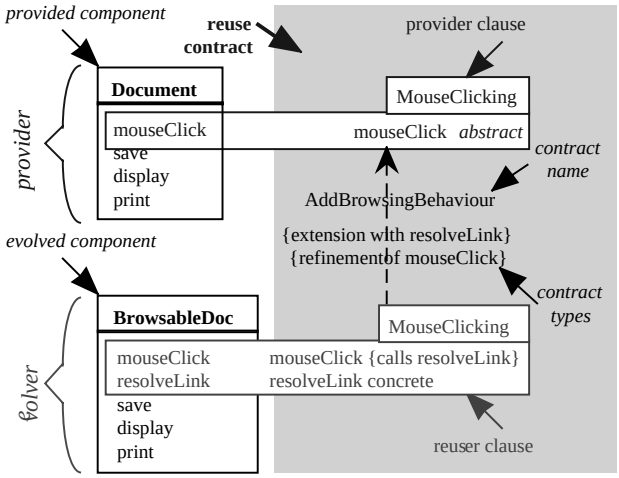

Figure 1: Documenting evolution of classes

To avoid such problems, we need to make evolution more *systematic*. Both the provider and the evolver have to specify *contractual obligations*. The primary obligation of the provider is to document what properties of the component can be relied on by other components. This is specified in a so-called *provider clause*. For example, in Figure 1 the provider clause specifies that the `mouseClick` operation in the `Document` component is an abstract operation. The evolver or reuser is obliged to document how the provided component actually evolves. This is specified by means of a *reuser clause* and a *contract type*. In the figure, the contract type specifies that `Document` is extended with a new operation `resolveLink`, and that the `mouseClick` operation is refined with an extra operation invocation. In the reuser clause we can see that this is an invocation of `resolveLink`.

The *contract type* expresses the specific way in which a component evolves. Many different kinds of contract types can be identified. The most important ones are extension, cancellation, refinement and coarsening. The contract type imposes obligations, permissions and prohibitions onto the evolver. For example, the extension contract type obliges evolvers to add new operations, but prohibits overriding of existing operations. It permits adding multiple operations at once. Contract types and the obligations, permissions and prohibitions they impose are fundamental to systematic evolution, as they are the basis for detecting conflicts when provided components evolve: *evolution conflicts correspond to breaches of contractual obligations and prohibitions.* Depending on the kind of contract type, different kinds of evolution conflicts are possible.

In a reuse contract, both parties provide only a certain view on the component. Thus a component can participate in different reuse contracts. Different contracts address different concerns of the provided component. In the example, different concerns might be navigation, input/output and displaying. Each contract expresses the properties of the component regarding the concern it addresses: which services a component should at least provide for that concern, the dependencies between services, and so on.

3. CLASS COLLABORATIONS

Instead of only specifying the evolution of single classes with reuse contracts, it is also possible to express evolution of more complicated components such as collections of collaborating class interfaces (or any other kind of software component used during the software development process). As an extension of the previous example, Figure 2 shows how the `Document` interface collaborates with a `Browser` interface to express the navigation concern. An UML-like notation [5] is used. We make use of a package annotated with a user-defined stereotype «provider clause» to represent a provider clause. This provider clause contains a class collaboration to represent the static structure of the participating classes, and an interation diagram to show the dynamic structure (i.e. the message interactions) of participant instances. [4] discusses how reuse contracts for collaborating classes can be incorporated in UML.

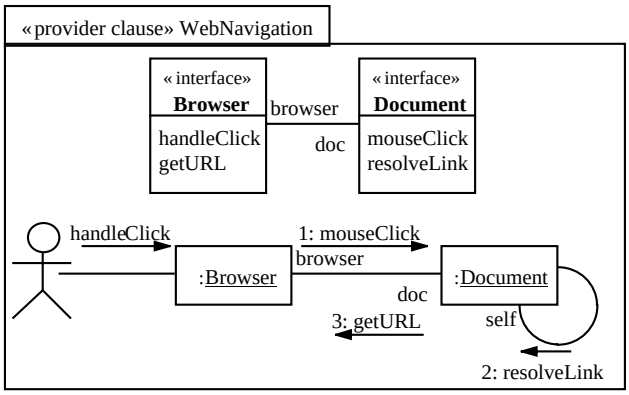

Figure 2: Provider clauses for collaborating classes

Figure 2 expresses the essential design for navigation in a web browser. There are only two participating interfaces in the collaboration: `Browser` and `Document`. These communicate with each other through an association with two roles: `browser` and `doc`. `Document` contains two operations: `mouseClick` and `resolveLink`. `Browser` also contains two operations that are important for navigation: `handleClick` and `getURL`. When a mouse click is detected by the browser, the `handleClick` operation is invoked. This operation detects whether the click occurs inside a document. If this is the case, the browser sends a `mouseClick` message to `Document`, which determines if this mouse click causes a link to be followed. If this is the case, the `resolveLink` self send is issued. `resolveLink` specifies what happens when a hyperlink is followed in the document, and sends a message `getURL` back to `Browser` to fetch the contents of the web page pointed to by the hyperlink.

4. DETECTING EVOLUTION CONFLICTS

Figure 3 shows a user-defined customisation of the WebNavigation component, where `Document` is specialised to a new kind of document (a PDF document) that only contains hyperlinks that point to places within the document itself. For this reason, the targets of these links can be retrieved by the document itself. This is achieved by removing the `getURL` invocation and replacing it by a `gotoPage` self send. In object-oriented languages this kind of customisation (or specialisation) can easily be achieved by creating a subclass `PDFDocument` that inherits from the original `Document` class. This subclass adds a new `gotoPage` operation, and overrides the `resolveLink` operation.

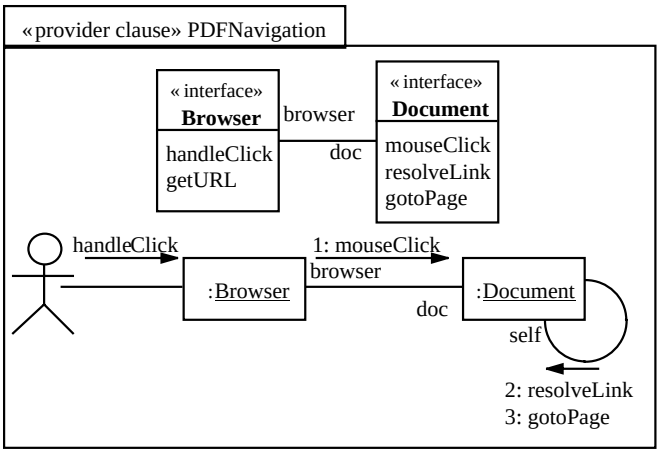

Figure 3: Customisation of WebNavigation provider clause

Suppose now that the original `WebNavigation` provider clause evolves by adding history behaviour (Figure 4). As a result of this, each time a hyperlink is followed through `getURL`, the URL of this link is stored somewhere through an extra invocation of `addURL`. This allows us to return to this location at a later time.

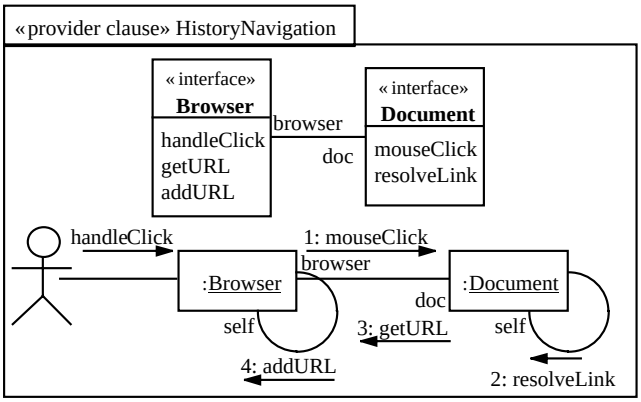

Figure 4: Evolution of WebNavigation provider clause

Both modifications work fine separately, but an evolution conflict arises when we try to combine the customised PDF document behaviour with the evolved history functionality. Since link resolving is dealt with by the PDF document itself, `resolveLink` does not invoke `getURL` anymore. As a result, the `addURL` operation in `Browser` will never be invoked, so the history will not be updated when a link is followed within the `Document`. This conflict is called an *inconsistent operations* conflict, since the `resolveLink` operation becomes inconsistent with `getURL`.

In order to detect this conflict automatically, we need to document the changes that were made to the original `WebNavigation` provider clause, in the customisation step as well as in the evolution step. Schematically, all these changes are illustrated in Figure 5.

- To obtain an evolved `HistoryNavigation`, two changes were made. First, an operation `addURL` was added to the `Browser` class, and then a self send was added from `getURL` to `addURL`. The first step can be expressed by a reuse contract with contract type «extension», the second step by a contract with type «refinement».
- To obtain a webbrowser for dealing with `PDFNavigation`, we needed to make three changes. First, an operation `gotoPage` was added to `Document` by a reuse contract with type «extension». Then, the invocation from `resolveLink` to `getURL` was removed by means of a «coarsening» reuse contract. Finally, a self send was added from `resolveLink` to `gotoPage` by means of a «refinement».

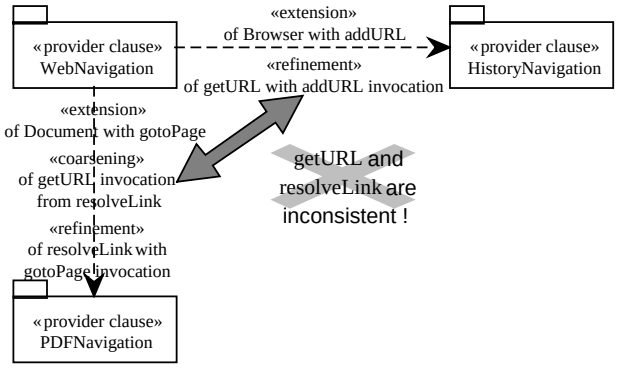

Figure 5: Detecting Evolution Conflicts

To enable automatic detection of evolution conflicts, a whole range of rules can be defined formally [3]. As the conflicts that can possibly occur are dependent of the contract type, tables can be set up where both the rows and columns represent contract types and the fields specify what conflicts can arise for a certain combination of types. This table can be filled in by simply comparing all contract types two by two, and determining whether they can interact in an undesired way. In the example above, we will find an inconsistent operations conflict between a «coarsening» (that removes the invocation of `getURL` from `resolveLink`) and a «refinement» (that adds an invocation from `getURL` to `addURL`).

5. CONCLUSION

In this extended abstract we proposed reuse contracts as a basic formalism for systematic software evolution. Reuse contracts provide an intuitive yet formal approach for reasoning about evolution of components. By explicitly documenting how components are customised and how components evolve, we are able to inventorise some of the conflicts that can occur during evolution, and to detect these evolution conflicts automatically. Moreover, the type of conflict that occurs already gives an indication of how it can be solved, possible even in a semi-automatic way. In this way, the problem of change propagation and ripple effects is addressed. For a detailed discussion of the underlying ideas we refer to [3].

An obvious virtue of our formalism is its simplicity. However, at the same time this is also its major shortcoming. Due to the simplicity, we are not able to detect all evolution conflicts. The more information that can be expressed in the contract clauses, the more conflicts we will be able to detect, but the more complex our model will become. The art lies in finding the right balance between simplicity and expressiveness, so that we can detect as much conflicts as possible without sacrificing the ease of use of the model. This is clearly an area of further research.

Another important contribution of reuse contracts is that the underlying ideas are independent of the kinds of components that are considered, and applicable in every phase of the software life-cycle. To validate this claim, we are currently developing a formalism in which reusable components are represented by *graphs*, while component evolution is expressed by means of *graph rewriting*.

6. ACKNOWLEDGEMENTS

This extended abstract reports on research performed by the Reuse Contracts Team, consisting of Prof. Theo D'Hondt, Dr. Carine Lucas, Dr. Patrick Steyaert, Dr. Koen De Hondt, Kim Mens, Roel Wuyts, and our industry partners at MediaGeniX.

7. REFERENCES

[1] Bohner, S. A. and Arnold, R. S. Software Change Impact Analysis. IEEE Computer Society Press, 1996.

[2] Codenie, W., De Hondt, K., Patrick, S. and Vercammen, A. From Custom Applictions to Domain-Specific Frameworks. Communications of the ACM, Special issue on application frameworks, 40(10), pp. 70-77, October, 1997.

[3] Lucas, C. Documenting Reuse and Evolution with Reuse Contracts. PhD Dissertation, Vrije Universiteit Brussel, September 1997.

[4] Mens, T., Lucas, C. and Steyaert, P. Supporting Disciplined Reuse and Evolution of UML Models. Proceedings of <<UML>>'98: Beyond the Notation, LNCS, Springer-Verlag, 1998.

[5] Object Management Group: *UML 1.1 Document Set.* OMG Documents ad/97-08-01 to ad/97-08-07, 1 September 1997.

[6] Steyaert, P., Lucas, C., Mens, K. and D'Hondt, T. Reuse Contracts: Managing the Evolution of Reusable Assets. Proceedings of OOPSLA '96, ACM SIGPLAN Notices, 31(10), pp. 268-286, ACM Press, 1996.