Resolving Name-Clashes in Eiffel

Semester Thesis

Alan Fehr

July 15, 2005

Student: Alan Fehr
Student-No: 01-908-862
Supervising Assistant: Bernd Schoeller
Supervising Professor: Bertrand Meyer

Contents

1 Definition of Name-Clashes

1.1 Definition

A name-clash in software development is a situation where two constructs use the same name. For this thesis the construct will be restrained to the notion of class. Thus, a name clash is a situation where two classes use the same name.

Name-clashes appear mainly in large software projects. When libraries from different sources are combined, the probability that two classes bear the same name increases. Also, in projects with many developers - possibly internal and external – involved, there is potential for naming conflicts to arise.

Many languages provide mechanisms for preventing name clashes or reducing their likeliness. Some of these will be presented in the following as well as the solutions which Eiffel and its supporting tools provide.

1.2 Name-Clashes in Eiffel

1.2.1 Rename for Features

Due to the fact that Eiffel (the programming language) supports multiple inheritance it is prone to name-clashes at the level of features. However, the language provides a thorough mechanism for resolving such name-clashes. The user is required to disambiguate them using 'rename' clauses.

1.2.2 LACE

One of the ideas behind Eiffel is to maximize reusability. For this reason, the language doesn't provide a mechanism for assembling a system. Some external mechanism is therefore needed to specify the entry-point for a program. One of these mechanisms is a language called LACE (Language for Assembling Classes in Eiffel).

Since the Eiffel knows nothing of assembling classes into systems it doesn't provide a mechanism for resolving name-clashes.

1.2.3 Rename in LACE

LACE offers a rename clause similar to that of the Eiffel language. However, this mechanism is used at the class level rather than the feature level. Using a rename clause, the user can disambiguate naming conflicts manually. The following example illustrates the use of rename in ace files. The system contains two clusters which both contain a class BOX. Assuming the root cluster is a client of BOX (i.e. it contains a class referencing BOX) the compiler doesn't know which version of BOX to reference. Using the rename clause, the user can specify alternative names for the conflicting classes. It is thus possible for client code to refer to a class using a different name than the original author had devised.

```
…
cluster

      inventory:        ".\inventory"


      gui:              ".\gui"


      root_cluster:     "."
            adapt
                  inventory: rename BOX as INVENTORY_BOX;
                  gui: rename BOX as GUI_BOX;
            end
…
```

The inconvenience of this solution is that the user must manually resolve name-clashes after they happen. A convenient mechanism should prevent name-clashes *a priori* or at least assist the user in disambiguating them.

1.3 Component Software

The trend in software development is towards component-oriented systems [3], [4]. They are a highly desirable form of software development and therefore their implications on namespaces and vice-versa must be analyzed as well.

1.3.1 Definition

"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties." – Szyperski and Pfister [3]

The question arises if classes can be viewed as components. Meyer states in [1]: "As a decomposition unit, a class is a module, that is to say a group of related services packaged together under a single name." Classes and components share many characteristics a component may however rely on multiple classes. In such a case, a class would no longer be a unit of independent deployment since it would rely on other classes to provide its specified services.

How are components accessed? When composing components, the interface is the essential part for its reuse. A component will export its functionality (possibly classes) under certain names which may lead to name-clashes. It is not essential that a component have a name, if it does however, a component becomes very similar to the concept of a module as will be seen shortly.

2 Possible Solution: Namespaces

2.1 Namespaces

2.1.1 Modules and Namespaces

Many programming languages solve name-clashes with some form of namespaces. One of the first languages (if not the first) to introduce such a concept was *Modula-2* with modules [5]. A Module provides a list of services under a common name. They provide more than just a namespace facility, but it is this facility which helps resolve name clashes. Client code may disambiguate an imported name by using the 'qualified name', meaning the name preceded by the module name and a dot ('.'). The user must specify which modules he wishes to import in the same place he uses them.

The user isn't required to use the qualified name *a priori* however which means a name-clash can still occur forcing the user to disambiguate it *a posteriori*. This slight inconvenience is the price for the rather large convenience of allowing the user to use shorter names in his code (i.e. unqualified names). This situation occurs in most languages which provide some form of namespaces and cannot be avoided without the painstaking cost of always using fully qualified names. However, the instructions on which namespaces or modules to import are commonly in the same place they are used: in the source code. The user has a close view over which namespaces he is currently using. A good development environment may warn the user of naming conflicts due to multiple imports and require that he disambiguates them while developing.

2.2 Global Namespaces

2.2.1 Drawbacks of local namespaces

Namespaces as described in the previous section aren't fully adequate for preventing name-clashes. There is no restriction on the names chosen for the namespaces which may lead to a situation where two namespaces have the same name. Due to the lack of coordination between different namespaces, the namespaces as previously described shall be called local namespaces. Local namespaces just redirect the naming problem to the namespace level. Instead of name-clashes at the class level (in object-oriented programming languages) name-clashes can now occur at the namespace level. Although name-clashes at the namespace level occur less often (presumably), ambiguous namespace-names don't provide a definitive solution.

The motivation behind global namespaces is to prevent name-clashes at the namespace level. Two authors could devise the same name for their namespaces. If a client needs to use two conflicting namespaces he is left without a utility to disambiguate the name-clash. The only solution would be to manually replace the names of one of the suppliers. This may or may not be possible, depending on the form of distribution, and is surely a completely undesirable solution. The process of renaming would have to be repeated every time the supplier provides a new version of his software.

Both XML [6] and Java [7] provide a mechanism for globally unique namespaces. The XML Standard suggests using an URI but acknowledges that URLs can be managed in a way to provide unique namespaces. The Java solution relies on URLs as well and will be described in detail.

2.2.2 Global Namespaces in Java

In Java a *package* is a named collection of classes (and possibly sub-packages) defining a namespace for the contained classes. Similar to the solution in Modula-2, developers may import packages in order to use the unqualified names for the contained classes as seen in the following example.

```
import java.util.Hashtable;
import java.util.Vector;


…
Hashtable m_states; // can now use unqualified names

Vector m_nfa_states
```

When declaring a class, the developer may specify which package it belongs to by using the package keyword. Sun recommends preceding the package names by the internet domain of the author (with reversed elements) as illustrated in the following.

```
Package ch.alanfehr.Sorting; //assuming this is my domain

class QuickSort {
…
```

If everybody follows this rule and only uses domains they own, we are guaranteed globally unique namespaces. Java furthermore provides access modifiers to specify if classes (and parts of them) may be accessed from outside of packages. If classes can not be accessed from outside a package they can also not be seen and will not take up a name for the user of the package.

2.3 Drawbacks of Namespaces

Globally unique namespaces provide a solid solution to name-clashes. Unfortunately, namespaces also introduce new problems. If introduced manually, i.e. neither as part of the language nor as part of the supporting tools, they will result in clumsy and hard-to-manage code. If merely included in the supporting tools the use of namespaces in code will be inconvenient. Thus, the language must be extended by a new construct serving the sole purpose of disambiguating names. Including such 'artificial' constructs can be undesirable in a language striving for simplicity and self-containment.

In systems with hierarchical namespaces refactoring can also be a problem. An example would be moving a Java subpackage in the hierarchy. Any client code using that class would be broken. This situation can also occur in systems with flat (i.e. non-nested) namespaces, but changing the hierarchy of systems is a common refactoring task, whereas completely removing a class from a group and inserting it into another definitely should not be.

2.3.1 Components Revisited

This view is confirmed if we consider namespaces as components. Given two individually distributable components classes should never be moved from one to the other while refactoring. Doing so would break any clients relying on the component losing the class which don't use the receiving component.

The desirable situation would be for components to be non-hierarchical namespaces. Any namespaces depended on should be included, but hidden from clients. This may of course lead to multiple inclusions of a namespace, but this shouldn't pose a problem. A smart compiler will only include one instance of the identical code.

3 Possible Solutions for Eiffel

Two solutions for resolving name-clashes in Eiffel will be shown in this chapter. The first solution suggests extending the language to prevent name-clashes *a priori* while the second solution provides a tool for resolving name-clashes *a posteriori*.

3.1 Extending the Language

3.1.1 Extensions

As shown, hierarchical namespaces are undesirable because they impose problems on refactoring. It is also desirable to impose a naming mechanism which will guarantee globally unique names. A reasonable solution would therefore include flat namespaces with the naming scheme used in Java.

Eiffel lacks the possibility to restrict access to a class in its entirety (although LACE provides such a mechanism). It only provides access restriction at the feature level. A mechanism for completely hiding classes is also a desirable feature; hidden classes would not take up names and clients of a component wouldn't know about a hidden class at all.

Apart from specifying access restrictions when defining classes it should also be possible to specify which imported classes should be re-exported. Such an export would be necessary if an exported class has a feature which includes the imported class in the signature.

Feature	**Implementaion**
Global Namespaces	Add a keyword **`namespace`** and require a namespace clause at the beginning of a class file. A namespace clause is composed of the keyword and an identifier, specifying which package the class is in: **`namespace`** `domain.package_name`
Class Access Restriction	Use the same mechanism which is used for restriction at the feature level: a class may optionally specify who may access it (and thus to whom it is visible). **`class`** `MYCLASS {accessors}`

Namespace Access Restriction	Allow namespace names in access lists. Adding the keyword **namespace** to an access list denotes the current namespace, the following class is only visible from within the current namespace. `class MYCLASS {namespace}`
Importing Namespaces	Add a keyword **using** and an import clause specifying which namespaces to import. The clause must appear after the namespace clause but before the class declaration. `using ns1, ns2, ns3`

Re-exporting classes	Any imported classes appearing in exported feature signatures are exported with the same status as the feature. Optionally, it could be allowed to specify classes using the **export** keyword in a using clause. The exported classes would be enclosed by **export** and **end**. The keyword **all** would export all classes in the namespace.
	`using parser export all end,`
	`structure export TABLE end`

3.1.2 An Example

```
-- file: application.e

namespace ch.alanfehr.resolve_wizard

using com.gobosoft.et_tools   -- contains class LACE_PARSER

class APPLICATION {ALL} -- do not restrict access

create
…

feature

      parse_with_lace_parser(parser: LACE_PARSER, file: STRING) is
                  -- LACE_PARSER is now also exported
            do
                  …
            end

…
```

```
-- file constants.e

namespace ch.alanfehr.resolve_wizard

class CONSTANTS {namespace} -- restrict access to current namespace

…
```

3.2 Resolve Wizard: an External Tool

Resolving Eiffel name-clashes *a posteriori* using LACE and manual renaming can be cumbersome. Resolve Wizard is a tool which assists the user in resolving name-clashes in an Eiffel / LACE project.

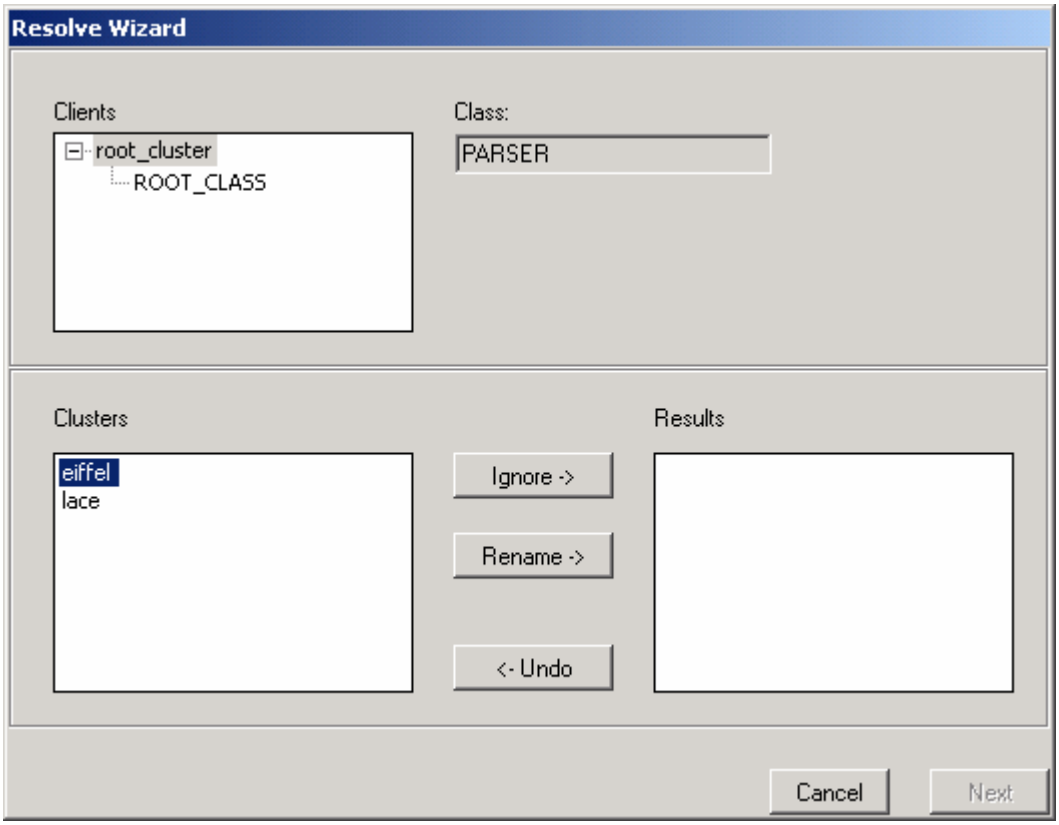
3.2.1 General Use

The tool provides a GUI which allows the user to load the ace file of his project. The tool then loads the entire project – which must all be provided as source code - and checks for name-clashes. It displays information about conflicting names and checks if any conflicts need

resolving. A naming conflict is found if two or more files (in different clusters) declare a class with the same name. A conflict may also arise, when an adapt clause (in the ace file) renames a class to an already existing class name. Conflicts only need to be resolved if there is a client of the conflicting class in a different cluster. Clients in the same cluster refer to the sibling class per default. The case where a user defines a conflicting class and a client, which should refer to an external provider, in the same cluster is not handled by this tool.

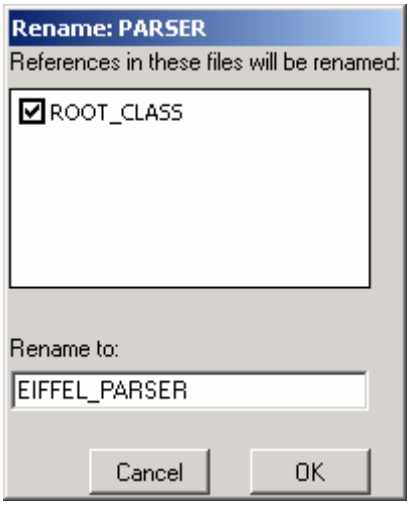
If there are conflicts in need of resolving, the tool displays options for each *resolve issue*. A resolve issue consists of a name-clash and all clients residing in a specific cluster. Clients in a different cluster form a separate resolve issue since LACE adapt clauses only apply to one cluster. The options for an issue include renaming a class (by using a rename clause in the ace file, the class itself is not changed) and the references in its clients, ignoring a cluster and removing a conflicting rename clause.

3.2.2 A Detailed View of Resolving Issues



This window displays options for resolving an issue. The "Clients" display shows the client cluster and all its classes which have references to the conflict class. The "Class" display specifies which class is the root of the conflict.

All clusters which define the conflicting class are listed under "Clusters". To resolve a conflict the user must select a cluster from the list and press either Ignore or Rename. Ignore is to be used with care and should only be selected if no listed clients have references to any classes in the selected cluster. The cluster will be completely ignored and since the tool does not check for further references to the cluster, ignoring a needed cluster leads to an invalid system. The safer alternative is to use Rename.



The tool will add a rename clause to the ace file, renaming the class (PARSER) to the specified name (EIFFEL_PARSER). It will also rename the references in all selected files.

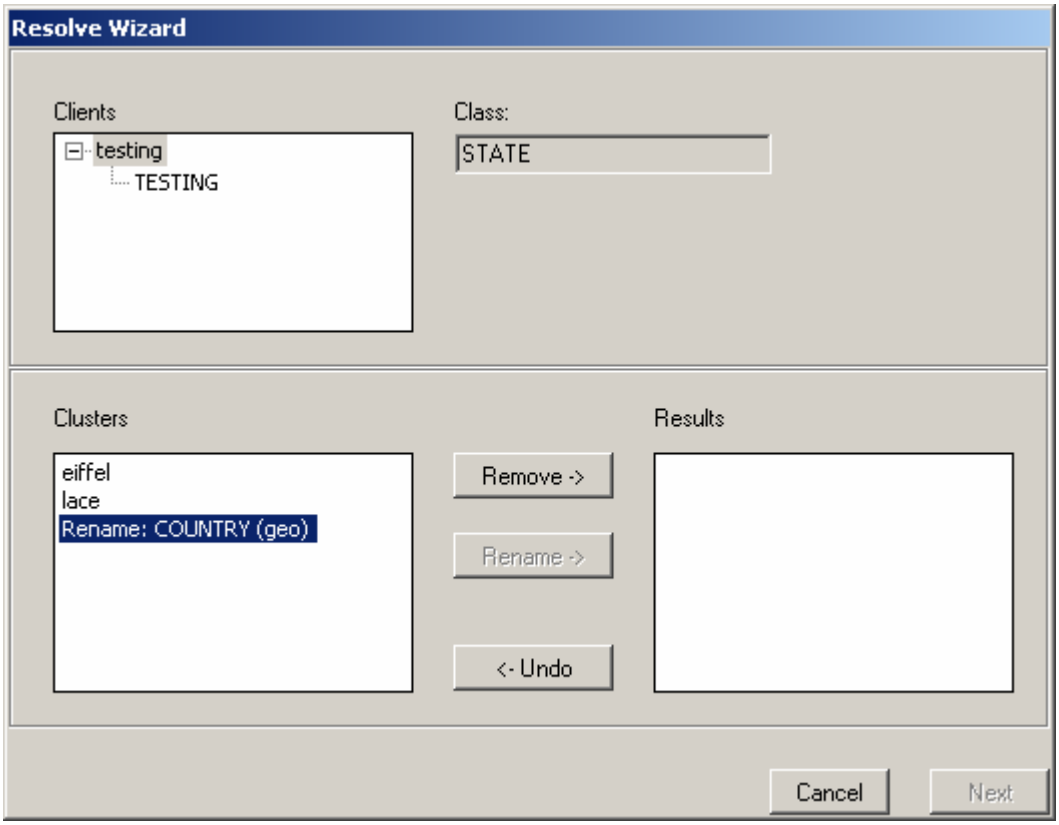
The "Next" button will become available if the issue is resolved. An issue is resolved in both of the following cases:

- all clients have been renamed
- the field "Clusters" lists less than two clusters

If "Next" is pressed, all class files which were selected for rename are processed and the references therein are renamed. The tool creates backup files with an additional ".bak" extension. If there are further issues to be resolved, the tool proceeds to the next issue. If the current issue was the last, the tool makes the selected changes to the ace file, also creating a backup file, and returns to the main window.

If "Cancel" is pressed, the tool returns to the main window, without modifying the ace file. Changes made to class files in previous steps are not removed, however.

3.2.3 Conflicting Renames



Rename clauses can also lead to name-clashes though it is an uncommon scenario. The tool only supports removing the rename clause, no references are renamed. This may lead to references pointing to the wrong class. Also, removing the rename-clause may lead to a new name-clash which will go unnoticed and unhandled in the current session. The recommended solution is to keep the rename clause and use Rename or Ignore on the other clusters.

4 References

[1] Bertrand Meyer: *Eiffel – The Language*, Prentice Hall 1992

[2] Bertrand Meyer: Object-Oriented Software Construction, 2nd edition, Prentice Hall, 1997.

[3] Clemens Szyperski: Component Software – Beyond Object-Oriented Programming, *2nd edition*, Addison-Wesley / ACM Press, 2002

[4] Karine Arnout: *From Patterns to Components,* ETH-HDB (Zurich) 2004

[5] Niklaus Wirth: Programmieren in Molula-2, 2nd edition, Springer 1991

[6] W3C: *Namespaces in XML,* http://www.w3.org/TR/REC-xml-names/, 1999

[7] David Flanagan, *Java in a Nutshell, 3rd edition*, O'Reilly 1999