Customization of Links between Classes

Robert Chignoli, Pierre Crescenzo, Philippe Lahire[1]

Abstract This paper describes the main aspects of a model dedicated to the customization of relationships between classes in object-oriented languages. The aims of this model is to generalize and control the kinds of relationships between classes, such as inheritance and composition, and thus to offer the ability to specify new relationships like generalization, code reuse, versioning, ... The three main entities of the model are presented: *link*, *description* and *language*, and the customization aspects are described. Each concept includes a set of *parameters* used by the algorithms (*actions*) of the model. An example of link implementation is proposed: the extension of the *Java* language with a link for code reuse, first by using two well-known methods, then by defining an *ad hoc* link specified thanks to the model.

Keywords Object-Oriented Programming, Links between Classes, Customization, Software Engineering, Inheritance, Composition, Aggregation, Code Reuse, *Java*

[1] *For the three authors:* Laboratory I3S (UNSA/CNRS), Team OCL, Sophia Antipolis, France – E-Mails: {Robert.Chignoli | Pierre.Crescenzo | Philippe.Lahire}@unice.fr

1 Introduction

The problematics of our study is set in the framework of object-oriented languages [MAS 89, DUC 98] providing notions of *class* and *inheritance*, especially in the perspective of software engineering. In these languages, it is frequent to use the inheritance mechanism that is provided in order to implement different kinds of links. For instance, people do not hesitate to use inheritance to implement a strict sub-typing as well as a basic source code reuse operation [MEY 97]. This range in the use of inheritance demonstrates the very wide interest of this mechanism but also shows a straightforward default : it is very difficult for a programmer to specify what is the use he wishes to make of this mechanism, including all the consequences that it may imply according to control, readability, documentation, maintainability and the evolution of programs. Our approach will allow us to look into an improvement of inheritance expressiveness by providing customization parameters for this mechanism to programmers, and by proposing combinations of these parameters for the most common cases of uses.

This motivation goes beyond the framework of programming and reaches the one of design. Actually, there is a gap, and it sometimes tends to increase itself, with the accuracy and the semantics richness described in the last steps of the design and the capability provided by the programming languages to their expressiveness. Such reconciliation and bringing closer process are also what we promote, keeping in mind that this may be possible only through an open system. It allows us to specify the wished behavior. It also allows to remain compatible with what already exists and with the other extensions.

In the forthcoming part of the paper, section 2 presents the three basic entities of the core model (*links*, *descriptions* and *languages*). Section 3 mentions an essential aspect (*parameters* and *actions*) and tackles topics such as the customization level and link composition. And then, section 4 describes an example of new import link additions (reuse of code) in *Java* [CLA 97, GOL 98, ARN 98]. Finally, the last section concludes and proposes some perspectives for the evolution of this work.

2 Basic entities of the model

In order to reach the objectives described in the introduction, we propose a model [CHI 96, CHI 97, CHI 99a, CHI 99b, CHI 99c] which describes the main concepts and semantics of object-oriented languages based on classes. This proposal relies on three basic foundations: links (such as aggregation[2] and inheritance), descriptions and languages[3]. Our ambition is two-fold, on one hand, to be general enough to be able to describe the concepts of description and link of industrial object-oriented languages based on classes (*Java* [CLA 97, GOL 98, ARN 98], *C++* [COP 92, CHI 95, STR 97], *Eiffel* [MEY 94], ...); on the other hand, to allow peo-

[2]We name *aggregation* the client-supplier relationships which most of the time are defined through an attribute (with referencing). This link is also called *client relationship* in the *Eiffel* terminology and *composition by reference* in *UML* [LAI 97, BOO 98, RUM 98, JAC 99].

[3]The model describes also the attributes, methods, messages, basic types, statements, control structures ... but this goes beyond the scope of this paper.

ple to experiment new concepts (example: version-description link) and test their validity in order to make programming languages evolve. The three basic entities of the model are described in this section. They provide a meta-programming layer and a customization interface presented in section 3.

2.1 Links

The first fundamental concept of the model is dealing with link-concepts. We use the following terminology: a link-concept is an abstraction of a kind of link in the object-oriented languages (an example of link-concept: the *Java*-like inheritance link `implements`; an example of link: the `implements` relationship between an interface and a *Java* class). In the model, one may find the most common links of object-oriented languages described under the form of a specialization tree (cf. figure 1 page 3 which shows some of these links). In order to be more precise, each link-concept could be, for instance, described by one class in an implementation of the model, and the occurrences of this class could be links. In figure 1, the specialization tree could be described as an inheritance tree; the leaves, adapted to the described language, would only be concrete classes; for instance, the `INHERITANCE` leave would disappear, for *Java*, and would be replaced by `IMPLEMENTS` and `EXTENDS`. It is the same for the description-concept and the language-concept presented later. We have special interest in the link between classes, in particular in the inheritance link which we would like to express better according to its use. We defined, for each link, the notions of *source-description* (the one which describes the link) and *target-description*. For instance, according to inheritance, the source-description is the heir and the target-description is the ancestor; for aggregation, the source-description is the one which declares an attribute (or a method parameter or . . .) and the target-description, the one which describes the type of this attribute. Aggregation and composition[4] are use links. Inheritance and code reuse are considered as import links.

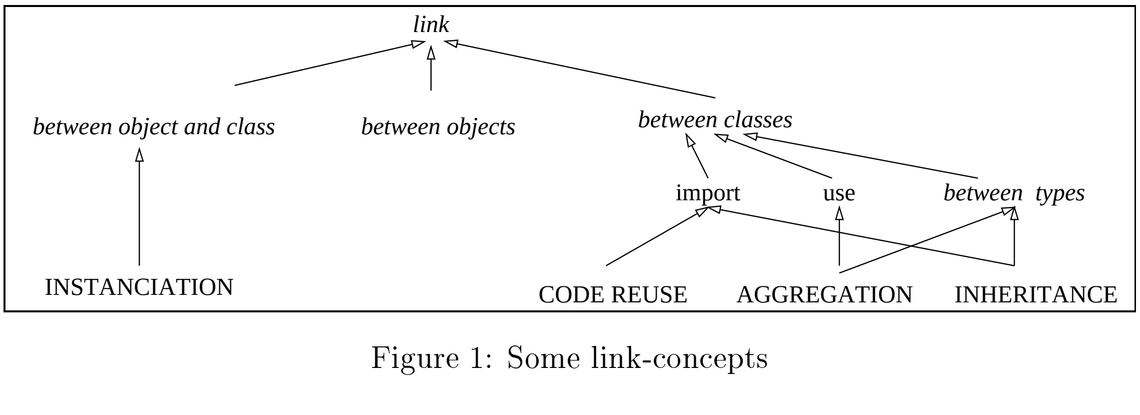

Figure 1: Some link-concepts

Also, the model describes links between descriptions and objects but our study is limited to the *instanciation* (with its inverse: the *extension*). On the other hand,

[4]The *composition* link should be understood as the *composition by value* in *UML* and *expansion* in *Eiffel*. It describes a specific *use* within which source-description declares that its occurrences *contain* an occurrence of the target-description (but not *reference* as we say for aggregation).

the links between objects do not belong to the scope of our research, though they could be described.

2.2 Descriptions

The modeling of the notion of description is more traditional [GOL 89, KIC 91]. Our objective is to make several description-concepts coexist with different semantics. That means to handle some kind of interoperability (let us imagine a language which provides description-concepts of both *Eiffel* and *C++*) as well as to take into account languages such as *Java* (in which we describe the notions of interface and class as two description-concepts).

Each description-concept defines a set of link-concepts it is compatible with, and handles their interactions. It is at this level that are handled possible problems related to the composition of links coming from different link-concepts. The most well-known example dealing with the problem of composition of links highlights the famous difficulties of the multiple and repeated inheritance uses. For instance: *Eiffel* would be described as one description-concept and three link-concepts corresponding to client relationships, expansion and (multiple) inheritance.

Finally, let us note the description-concepts which integrate the notion of genericity. The notion of type is also modelized. Each description describes several or only one type, depending on whether it is generic or not. Some links, such as inheritance, interact between types (through the intermediacy of descriptions which describe these types), whereas others, such as code reuse, only take effect in descriptions.

2.3 Languages

The language-concept is an important and simple notion. It describes, as is meant implicitly by its name, a modelized language. Each language includes two components:

1. a set of description-concepts
2. a set of link-concepts, each being compatible with at least one of the selected description-concepts.

The set of link-concepts can either be deduced from the set of description-concepts, or be defined explicitly; this possibly allows to reduce, at this level, the compatibility rules described within each description-concept. For instance, in a language, one may wish to use the *Eiffel* description-concept but without the ability to use the expansion link. Of course, only a restriction of the description specifications is allowed, but no extension.

In the end, language modeling will allow to implement and control the interoperability between objects created by different language-concepts.

3 Parameters and actions

We have just described the basic entities of the model. Because our goal is to provide a customization of these concepts in such a way that people may adapt them to specific contexts or software engineering experiences, it is necessary to describe their semantics. Our approach consists in the development of operational semantics that may be adapted, to each concept, thanks to a set of parameters guiding the execution of a system of actions. Each concept (link-concept, description-concept, language-concept) contains parameters. Only the link-concepts and the description-concepts have actions and a protocol to meta-program them. The meta-programmer's job may take two forms: he either gives a value to the parameters of each concept[5], or he wishes to modify the semantics associated to at least one of its parameters. As far as the first case is concerned, his job is finished and this is one of the interests of our approach. As regards the second case, he has to redefine actions whose semantics do not fit him. First, we will present the meta-programming protocol related to actions, and then, some of the parameters and actions attached to descriptions or links and some examples of actions. Finally, we will examine the levels of customization and the composition of links.

3.1 Meta-programming actions

The handling of actions follows a systematic protocol instaured to make their use and handling easier. Any action *AC* (examples: `look_up`, `assign`) is defined by the following elements :

- One boolean feature `IS_AC_VALID` comes `true` if *AC* corresponds to some semantics in the current language-concept, otherwise it is `false`.
- One feature `AC`[6] describes the implementation of the semantics related to *AC*.
- As far as the actions dealing with links only are concerned, we have a set of features `AC_`*`L`* (where *L* is a letter) which defines the steps corresponding to each *sub-behaviors* for `AC`.
- One set of features `DEFAULT_AC_`*`X`* (where *`X`* is an integer). It provides standard behaviors for *AC*, `DEFAULT_AC_0` describing, most of the time, the behavior of `AC` by default. This system may be extended to link action sub-behaviors.

3.2 Parameters and actions of links

We call *action of link-concept* any feature describing an essential behavior of link-concepts. At present, we distinguish more than forty customizable actions for the link-concept. From now on, as has been done in the title of this section, we will lighten the explanation by talking of *link action* instead of action of link-concept. The seven categories of parameters and actions dealing with links are feature look-up, semantics control, feature-call handling, management of description occurrences (creation, removal and assignment), management of description extensions, basic operations (copy, equality, . . .) and adaptation clauses (renaming, redefinition, removal, . . . detailed in the example, section 4). The algorithm of this action takes

[5] A parameter is a value, examples: an integer, a boolean, a table, . . .

[6] This feature is a routine (function or procedure) with or without parameters.

into account the value of link parameters in order to perform an action which is semantically valid : for instance, the `look_up` action, which searches for the feature to be performed according to the feature-call checks, among other things, if the link allows polymorphism or not and in which way(s).

Link parameters provide the essential information in order to allow a proper execution of the actions; here are some of them:

`cardinality` It describes the link cardinality under the format $1 - n$ which means that the link can be created between 1 source-description and n target-descriptions (n is the infinite value or a constant integer greater or equal to 1). For instance, a simple inheritance link will have the cardinality $1 - 1$ (each description can inherit from only one description), whereas a multiple inheritance link will have the cardinality $1 - \infty$ (each description can inherit from several descriptions). So that one will be able to limit the cardinality of an inheritance link $(1 - 3)$ that is to say to normalize the programming tasks within a development team or an entire company[7].

`circularity` It describes the ability to create cycles with the link: `true` means that cycles are allowed, `false` means that they are prohibited. For instance, inheritance and composition prohibit cycles whereas aggregation often allows it.

`polymorphism` It indicates if polymorphism is allowed by the link and if so, in which way it is possible. Let us take several examples: aggregation prohibits polymorphism, specialization allows it in the source-to-target way, generalization allows it in the reverse way and we may imagine a version link which allows it both way.

`repetition` It says if a repetition (direct or indirect) of a description is allowed in the source-description or in the target-description.

`symmetry` It says if it is a symmetrical link.

`opposite` It describes the opposite link if it exists. As far as we are concerned, there is a difference between symmetry and an opposite link. A link is symmetrical if it provides a symmetrical semantic relationship. One may imagine a link like *is-a-kind-of* for which the target-description and the source-description are the terms of such a symmetrical link. Besides, an opposite link defines *another* link describing some opposite semantics. For instance, a specialization link is the opposite of a generalization link. These definitions lead us to say that a symmetrical link is its own opposite (link).

The link-concepts may be described within two different contexts: a closed world where they know about the description-concepts which are implemented (this is the case when a language is modelized); an open world where the description-concepts are not *a priori* known (for instance, when a library of link-concepts is implemented). Several parameters take this distinction of context into account.

[7]We design our links with an $1 - n$ cardinality but the model allows to generalize them into $m - n$.

3.3 Parameters and actions of a description

The parameters and actions of description-concept are, for the descriptions, the counterpart of the parameters and actions of links. We call a *description action* (in fact the action of the description-concept) a feature which defines an essential behavior of a description. There are by now more than forty customizable actions for a description. The main situations that are taken into account are, of course, closely related to those handled by link actions, except for the adaptation clauses that do not exist any longer. The ability of a description to create occurrences, to handle its extension and the set of link-concepts that are compatible with it, represents three significant examples of description parameters.

3.4 Examples of actions

To illustrate our remarks, here is first the code (given here as a pseudo-code) of the *execute* description action describing the execution of a message `m`. It is the schematic algorithm of a by-default behaviour. It is provided here to give an idea of the meta-programmation task which is necessary when a by-default action does not fit. The control of errors (of each internal procedural or functional call) is only introduced for certain calls, i.e in this case for the `look_up` and `is_parameters_compatible` codes, to lighten the algorithm.

We also have to add that all the routines called by `execute` are description actions themselves and that *they* also have one or several standard behaviours (including one by default). In order to develop a meaningful example without any elements of no use for the understanding, the context of these actions is the following :

- The import graph is not cyclical.
- For the imported features, we only deal with renaming.
- We only consider the parameters of polymorphism.
- The behaviour described below for the search of a feature (`look_up action`) in the import graph (dynamic linking) is the following : we first search in the current feature, then in the current description (these two operations are carried out by the `match` action according to compatibility rules for polymorphism, overloading, . . .). If no feature happens to be found, we search in the target descriptions (for which ascendant polymorphism is allowed). If no feature is found, we search in the description sources (for which descendant polymorphism is allowed). The first feature which is found compatible with the message is considered as the one.

```
procedure {action of description} execute(m :  MESSAGE)
 local variables
  p :  FEATURE
 begin
   p ← NULL
   // Looking for the p feature which corresponds to m
   // in the type hierarchy (taking into account polymorphism parameters)
   p ← look_up(m)
   if p is not NULL
   then
     // Evaluation of the effective parameters
     parameters_evaluate(m)
     // Control of validity of the effective parameters
     if is_parameters_compatible(m, p)
     then
       // Attachement of the effective parameters to the formal parameters
       attach_parameters(m, p)
       // Actions that should be performed before the one of p
       before_execute(p)
       // Execution of p (taking into account parameters of m);
       // do_execute coordinates the calls to the execute link actions
       do_execute(p)
       // Actions that should be performed after the one of p
       after_execute(p)
       // Un-attachment of the effective parameters
       detach_parameters(p)
     else
       // Handling of semantics errors (the parameters of m are not compatible with p)
     endif
   else
     // (m does not correspond to any feature)
   endif
 end
```

Let's have a look now at the *look_up* action. We will deal first with the description action, then with an associated link action called *look_up_up*. Let's recall that they are only examples, because the search for features can take various forms : the precedence list, the priority of certain kinds of links, the order given by the programmer …

```
function {action of description} look_up(m : MESSAGE) : FEATURE
 local variables
   p : FEATURE
   l : LIST
   i : IMPORT_LINK
 begin
   p ← NULL
   // Looking for a feature in the current description
   p ← match(m)
   if p is not NULL
   // nothing corresponds locally
   then
     // Looking for a feature that could be accessed through polymorphic characteristics
     l ← list of all import links attached to the current description
     i ← l.head
     while l is not empty and p is not NULL
       // Handling of the ascendant polymorphism (to the targets)
       // with the use of a link parameter
       if it is polymorphic in the ascendant way
       then
         p ← i.look_up_up(m)
       endif
       if p is NULL
```

```
        then
          // Handling of the descendant polymorphism (to the sources)
          // with the use of a link parameter
          if i is polymorphic in the descendant way
          then
            p ← i.look_up_down(m)
          endif
        endif
        l.next
      end-while
    endif
    return p
  end

function {action of link} look_up_up(m :  MESSAGE) : FEATURE
  local variables
    p :  FEATURE
    l :  LIST
    d :  DESCRIPTION
    n :  MESSAGE
  begin
    p ← NULL
    l ← list of all the target-descriptions of the current link
    d ← l.head
    // Looking for a feature that could be accessed through a polymorphic characteristics
    while l is not empty and p is not NULL
      // Handling of a possible renaming made in d and dealing with m
      if there is a renaming of m made in d
      then
        n ← renaming of m
      else
        n ← m
      endif
      // looking for (recursively, deeply first, through the call to the action of the
      // look_up description) the p feature of the d description
      p ← d.look_up(n)
      l.next
    end-while
    return p
  end
```

3.5 Levels of customization and link composition

As far as customization levels are concerned, each link parameter may be redefined at the levels of the description-concept and the language-concept in order to favor reusability. Of course, in such a case, this redefinition may only restrict the specification given by the link itself. For instance: a multiple inheritance link may be used in order to describe a simple inheritance link (by restricting its cardinality), but not the opposite. It would not be something like redefining in the sense of inheritance, but rather introducing constraints according to the following rule. The parameters of language prevail over those of description which also prevail over those of link. For instance: a given language-concept uses a given description-concept but restricts the multiple inheritance accepted by the latter to simple inheritance. It is also possible to consider this parameter at the link level (which are occurrences of link-concepts). So that, if the cardinality of a link-concept is $1 - \infty$, the cardinality of its occurrences can be $1 - 3$. This additional information allows us to make the interoperability easier (for instance, an inheritance link $1 - 1$ coming from a multiple inheritance link-concept is possibly compatible with a simple inheritance link-concept) and to provide integrity and consistency controls.

The composition-of-link problem occurs when a description-concept allows the use of several link-concepts. We will present below four examples that may generate conflicts, with some elements of solution. The approach to solve such conflicts is always the same: problems between link-concepts are solved by the actions of description whereas the internal problems of link-concept are solved by the actions of link.

1. **Inheritance and aggregation**: For instance, in the *Eiffel* language, it is possible to hide some features in the heirs of a description which declares them as exported. This is an example which demonstrates that interaction facilities also exist between use (aggregation) and importation (inheritance). The implementation of this influence between links is made through the use of an action from importation links which allows to hide a feature (`hide`).

2. **Multiple inheritance**: Our model allows two kinds of multiple inheritance modeling. The first and easiest one is the direct use of a *multiple inheritance* link (`cardinality` to $1 - \infty$) which takes care of possible conflict problems. The second solution consists in the use of several *simple inheritance* links (`cardinality` to $1 - 1$) and in solving the conflicts at the level of the `look_up` description action thanks to the actions of link (adaptation clause). We recommend the exclusive use of the first solution because it is more natural from the point of view of modelization and it is easier to implement.

3. **Inheritance and sub-typing**: A description is set as heir (in the common sense of inheritance) and sub-type (this is a more restrictive link than inheritance) of a same other description[8]. These description actions are the ones which must remove ambiguities. Sub-typing being more restrictive than inheritance, the semantics of sub-typing are the ones that must be taken into account in those actions.

4. **Specialization and generalization**: These two links are *a priori* not compatible: that is to say they should not link, in the same way, two descriptions. Besides, they can be considered as inverse.

4 One example: a *code reuse* link

We have chosen to present only elements of one solution[9] to the following problem: we wish to experiment the integration of a *code reuse* link (for instance materialized by the keyword `reuses`) in the *Java* language. This is an import link which will be

[8]Let us forget about the reasons of such a programming choice that may be *a priori* surprising.
[9]A full solution can be modeled.

added to those describing *interface inheritance* (`implements`) and *class inheritance* (`extends`). Its aim is to allow a description to pick up a source code from another description. The interest of such a link clearly appears in the framework of software engineering. It is a question of better expressing what the programmer intends to do in his concern to get control, readability, maintainability and documentation in order to improve the produced software quality.

Without `reuses`, the code reuse may be achieved, either through inheritance or aggregation, and we do not question it. We only wish to demonstrate that an *ad hoc* link would be better. Let us imagine an `A` class that includes, among others, a `P` feature that we want to reuse within another `B` class.

4.1 Code reuse through aggregation

An attribute `Handle` of the `A` type in `B` is created. There can be some drawbacks to this method:

- `Handle` hasn't got any meaning from the point of view of modeling.
- `Handle` underlines a systematic indirection for reusing the code of `P`, it will always be necessary to call `o.Handle.P()`.
- It will be necessary that each class, which is a client of `B`, either explicitly calls `Handle` (implementation is not hidden to the client), or that `B` encapsulates the call to `Handle.P()` in a new feature (additional code).
- It is necessary to manage `Handle`: creation, initialization, life, destruction, memory freeing, . . .

4.2 Code reuse through inheritance

`B` inherits from `A`. There are some drawbacks to this method:

- If `P` is private (`private`), it is not possible to reuse its code because private features are not inherited[10].
- Polymorphism is possible[11] between `A` and `B` but is meaningless.
- `B` is a sub-type of `A`, yet this is false.
- All the occurrences of `B` (and of all its sub-types) are also occurrences of `A`, which makes the management of class extensions wrong.
- As there is only a simple class inheritance, `B` cannot inherit anymore.

[10] To decide whether it is right or not to wish to reuse a private feature will not be debated here.

[11] To say that it is possible comes to say that it is out of control except with a prohibitive overcost (for instance: some specific code that may sometimes need access to the meta-level [interpreter, compiler, . . .]).

4.3 Code reuse through an *ad hoc* link

About the link that we are describing, we propose to implement such a code reuse facility without the drawbacks that we have mentioned above. No composition with use links such as aggregation is necessary. The composition with the other import links is straightforward because code reuse has no influence on types and is compatible with everything. We can demonstrate this through the description of some examples of actions and parameters associated to links (we only present the most significant ones) that the meta-programmer has to define in order to implement such a link. In addition to one's interest in a specific link for readability, maintainability and control, one may find huge advantages when the linked descriptions evolve (for a new version of application or library, for instance). Indeed, a better specification allows programming environments to better adapt their behavior so that they may assist the programmer.

Parameters

`cardinality` The cardinality of this link is $1 - \infty$.

`circularity` Nothing prevents this link from being cyclic.

`polymorphism` No polymorphism is allowed with such a link. Besides, as an import link, this is one of its particularities.

`repetition` It does not cause any problem.

`symmetry` This link is not symmetrical

`opposite` In the present context, no inverse link is detected.

`rename` To rename features is allowed.

`redefine` To redefine features (signature, assertions or body) is prohibited (originality in relation to inheritance).

`remove` Feature removal does not underline any problem, thanks to the absence of polymorphism (originality in relation to inheritance).

`abstract` It consists in making one routine abstract and this is prohibited: abstracting one feature (that is to say *lose* its body) in a code reuse link has no interest (originality in relation to inheritance).

`hide` Hiding (which means to make a feature *private* and therefore not to export it anymore through the description interface — of its import links) is allowed without any constraint.

...

The parameters mentioned above guide the execution of the following actions for the link-concept that are being defined. Therefore the meta-programmer may redefine these actions in order to make them evolve or to modify their semantics or else, to take into account some situations not solved by the model.

Actions

- `match` It checks the validity of a feature thanks to a look-up based on its name, the number and types of its parameters, the presence or absence of returned values, ...
- `look_up` It implements feature searches (it may use `match` to achieve its objective). Because of the absence of redefinition, abstraction or selection clauses, it is much more simple than the inheritance one. Of course it uses the value of `polymorphism`. It can be broken up into two sub-actions i.e. `look_up_up` and `look_up_down` for the ascendant searches and the descendant searches respectively.
- `type_conformance` Two types are compatible only if the two types are strictly identical (another effect of the absence of polymorphism).
- `assign` It checks the validity of an assignment, then makes the attachment of the object in several steps: the evaluation of the right term of the assignment (source), the access to the left term of the assignment (target), the type compatibility control (much more simple because of the absence of polymorphism), the attachment of the source to the field which corresponds to the target (object or routine in case of a local variable), the management of persistent objects may also be taken into account ...
- ...

4.4 Summary

We have demonstrated, in this example, that such an approach was possible to implement. The newly defined link-concept allows to provide a source code reuse in *Java* without the drawbacks of both aggregation and inheritance. The meta-programmer's task consists mainly in creating one link-concept, in giving a value to each parameter, then in integrating a new link-concept into the `java-class` description-concept. By default, the actions of description take into account the parameters of link, so that the integration of a new link is made easier.

Here is below what our example written according to an extended Java syntax could look like. We could therefore choose to add a keyword `reuses` followed by a list of signatures of features (i.e the name of the routine, then the type of its parameters in brackets), then by the keyword `from`, then by the name of the target class. We do not consider the type of what is returned by the features in this signature because it does not present any possible ambiguity in *Java*. Besides showing the already mentionned `A` and `B` classes, we propose to add a `C` class in order to give a more significant picture of our approach.

```
public class A {
 public void P() {
   ...
 }
  ...
}

public class B
reuses P() from A {
  public int R(String name, double[] data) {
    ...
    P();
    ...
  }
  public void S(int i) {
    ...
  }
  ...
}

public class C
reuses R(String, double[]), S(int) from B {
  ...
}
```

5 Conclusion

In this paper, we first presented the outline of a model to customize the links between classes by defining the concepts of link, description and language. Then, we presented the system of parameters and actions associated to the heart of the model, and finally we developed an example of use link which shows, from our point of view, the interest of a mechanism which allows to better specify the links between descriptions within object-oriented languages based on classes.

We are now interesting ourselves in the specification of parameters defining language and in an extension of *Java* which allows to use, within a program, new types of link (we will start by adding the code reuse link described above). This will lead to the development of an *ad hoc* programming system which provides a test platform for the modification and the addition of links between descriptions: our study is turned towards a pre-processor of *extended–Java* generating pure *Java*. Finally, we will study a persistency service [CAP 99a, CAP 99b] which relies on this model and uses for itself link information to better manage persistency if necessary. This service will lead to build up the set of parameters that we have defined in order to take into account specific problems of persistency.

References

[ARN 98] K. Arnold and J. Gosling. The *Java* Programming Language. Second Edition, Sun Microsystems, The Java Series, September 1998.

[BOO 98] G. Booch, I. Jacobson and J. Rumbaugh. The Unified Modeling Language User Guide. The Addison-Wesley Object Technology Series, October 1998.

[CAP 99a] A. Capouillez. *ROOPS* : un service de persistance paramétrable pour *OFL*. Rapport de Recherche 99-14 et Rapport de stage de DEA, Laboratoire *Informatique, Signaux et Systèmes de Sophia Antipolis*, juin 1999.

[CAP 99b] A. Capouillez. *ROOPS* : un service paramétrable de persistance pour *OFL*. Rapport de Recherche 99-15 et Rapport de stage d'ESSI 3, Laboratoire *Informatique, Signaux et Systèmes de Sophia Antipolis*, septembre 1999.

[CHI 95] S. Chiba. A Metaobject Protocol for *C++*. *OOPSLA'95 ACM Conference* volume 30, 1995.

[CHI 96] R. Chignoli, P. Crescenzo and P. Lahire. An extensible environment for views in Eiffel. Research Report 96-53, Laboratory *Informatique, Signaux et Systèmes de Sophia Antipolis*, November 1996.

[CHI 97] R. Chignoli, P. Crescenzo et P. Lahire. Liens entre classes dans les object-oriented languages. Rapport de Recherche 97-22, Laboratoire *Informatique, Signaux et Systèmes de Sophia Antipolis*, juillet 1997.

[CHI 99a] R. Chignoli, P. Crescenzo and P. Lahire. *OFL*: An Open Object Model based on Class and Link Semantics Customization. Research Report 99-08, Laboratory *Informatique, Signaux et Systèmes de Sophia Antipolis*, March 1999.

[CHI 99b] R. Chignoli, P. Crescenzo et P. Lahire. *OFL* : une machine virtuelle objet flexible pour la gestion d'objets persistants et mobiles. Rapport de Recherche 99-09, Laboratoire *Informatique, Signaux et Systèmes de Sophia Antipolis*, mars 1999.

[CHI 99c] R. Chignoli, P. Crescenzo et P. Lahire. Un modèle de paramétrage des liens entre classes. Rapport de Recherche 99-13, Laboratoire *Informatique, Signaux et Systèmes de Sophia Antipolis*, août 1999.

[CLA 97] G. Clavel, N. Mirouze, E. Pichon et M. Soukal. *Java* : la synthèse. Masson, 1997.

[COP 92] J. O. Coplien. Programmation avancée en *C++* : styles et idiomes. Addison-Wesley Publishing Co., 1992.

[DUC 98] R. Ducournau, J. Euzenat, G. Masini et A. Napoli. Langages et modèles à objets : état des recherches et perspectives. INRIA Collection didactique, juillet 1997.

[GOL 89] A. Goldberg and D. Robson. *Smalltalk-80*: the Language. Addison-Wesley, 1989.

[GOL 98] M. Golm and J. Kleinöder. *metaXa* and the Future of Reflexion. UTCCP Report 98-4, 1998.

[JAC 99] I. Jacobson, G. Booch and J. Rumbaugh. Unified Software Development Process. The Addison-Wesley Object Technology Series, January 1999.

[KIC 91] G. Kiczales, J. des Rivières, and D.Bobrow The Art of the Metaobject Protocol. MIT Press, 1991.

[LAI 97] M. LAI *UML* : La notation de modélisation objet : Applications en *Java*. Masson, juillet 1997.

[MAS 89] G. MASINI, A. NAPOLI, D. COLNET, D. LÉONARD ET K. TOMBRE Les object-oriented languages : langages de classes, langages de frames, langages d'acteurs. MIT Press, 1991.

[MEY 94] B. MEYER. *Eiffel*, le langage. InterEdition, 1994.

[MEY 97] B. MEYER. Object-Oriented Software Construction (second edition). Prentice Hall, 1997.

[RUM 98] J. RUMBAUGH, I. JACOBSON AND G. BOOCH. The Unified Modeling Language Reference Manual. The Addison-Wesley Object Technology Series, December 1998.

[STR 97] B. STROUSTRUP. The *C++* Programming Language. Addison-Wesley Publishing Co., July 1997.