Doublecpp: double dispatch in C++
User's manual - Version 0.6

Lorenzo Bettini
Dipartimento di Sistemi e Informatica, Università di Firenze
Viale Morgagni 65, 50134 Firenze, Italy
http://www.lorenzobettini.it

March 31, 2009

Contents

1 Introduction

`doublecpp` is a preprocessor for C++ that handles a new linguistic construct for defining branches of a *multi-method* [DG87, MHH91, Cas95b, BC97]. The "right" branch of such a method will be selected dynamically at run-time according to the actual type of the object on which the method is invoked and to the actual type of the first argument (*double dispatch*). This document is a manual for users of `doublecpp`; the reader who's interested in further details about the implementation (and the underlying theory) of `doublecpp` is referred to [BCV05, BCV06], which are also available from my home page.

Let us provide some definitions, so that the aim of this program can be fully achieved:

receiver it is the object on which a method is invoked, for instance in the following code:

```
MyObj *obj = new MyObj;
MyPar *par = new MyPar;
obj->m(par);
```

`obj` is the receiver of the message (method) `m`;

```
class A {
  // fields...
  // methods...

 public:
  virtual branches m
   virtual T (T1 *t) { ... };
   virtual T (T2 *t) { ... };
   ...
  endbranches

  virtual branches n
   virtual S (S1 *t, V1 *v) { ... };
   virtual S (S2 *t, V1 *v) { ... };
   ...
  endbranches
};
```

```
#include "a.h"

class B : public A {
  // fields...
  // methods...

 public:
  virtual branches m
   virtual T (T2 *t) { ... };
   virtual T (T3 *t) { ... };
   ...
  endbranches

  virtual branches n
   virtual S (S3 *t, V1 *v) { ... };
   ...
  endbranches
};
```

Figure 1: A first example in C++ with double dispatch

double dispatch the ability of dynamically selecting a method not only according to the actual type of the receiver (*single dispatch* or *dynamic binding*), but also to the actual type of the argument (when all arguments are considered, we have *multiple dispatch*);

multi-method a collection of overloaded methods associated to the same message; the selection takes place dynamically according to multiple dispatch. However, in our context, we limit our multi methods to double dispatch. Each method in this collection is called a *branch*.

client, target we refer to classes (or structs[1]) defining and redefining branches of a multi method as the *clients* of the *target* classes (or structs), i.e., those used for declaring the type of the first parameter of a branch.

Trying to summarize the features of the new linguistic construct, we can say that it allows to define methods in overloading, but the selection of the "right version" does not take place statically (as it happens in C++) according to the static type of the argument, but dynamically according to the actual type of the (first) argument (apart from, of course, the actual type of the receiver). In a sense, it provides *dynamic overloading* capabilities.

Let us illustrate the new construct by the example in Figure 1 declares two multi methods, m and n. In each class, all the branches of a multi method represent a standard definition of a C++ overloaded method; as a consequence, for instance, the return type is not used in method selection. However, differently from C++, multi methods with all their branches are implicitly inherited in derived classes (of course, if they are not private in the parent class). Message passing is the key point, from the semantic point of view. Besides including the two standard mechanisms of method invocation, i.e., static overloading and dynamic binding (single dispatch), multi methods allow to model the additional mechanism of double dispatch: if m is declared as a multi method, then the invocation of m with the suffix `_DB`, m`_DB`, is interpreted by using dynamic overloading. Namely, when m`_DB` is invoked, the selection among the available branches takes place dynamically, choosing the most specialized branch among the ones available in the classes, according to the dynamic type of the receiver (as in standard single dispatch) and also according to the

[1]From now on, unless when required, we will use only the term class, also for struct.

dynamic type of the first parameter (i.e., double dispatch)[2]. Notice that, since we provide only double dispatch (instead of multiple dispatch), the other parameters should be of the same type, otherwise their types would be used statically for the selection (as in standard C++ overloading).

Thus, this extension provides dynamic overloading. For instance, if we have the following code (assuming that T_j is a subclass of T_i if $i \leq j$):

```
A *a = new A;
T1 *t = new T2;
a->m(t); // static overloading
a->m_DB(t); // dynamic overloading
```

then, the first method invocation is performed according the static overloading semantics, i.e., `A::m(T1 *)` is (statically) selected, but the second method invocation (dynamically) selects `A::m(T2 *)`.

Moreover, the new linguistic construct, also enables *covariant specialization* of the branches of a multi-method, i.e., subclasses are allowed to redefine a method also by specializing its argument (we refer the reader to [Cas95a] for a clear treatment of this subject): indeed, B redefines the branch `m(T2 *)` but also specializes the multi-method by adding a new branch, `m(T3 *)`, where `T3` is a subclass of `T2`. Again, the most specialized branch of a multi-method will be dynamically selected for invocation on objects belonging to subclasses; thus, considering the following code:

```
A *a = new B;
T1 *t = new T1;
a->m_DB(t); // dynamic overloading
t = new T2;
a->m_DB(t); // dynamic overloading
t = new T3;
a->m_DB(t); // dynamic overloading
```

the first invocation will select `A::m(T1 *)`, the second one will select `B::m(T2 *)`, because B has redefined the branch `m(T2 *)` and the third one will select `B::m(T3 *)`, because B has defined a specialized branch for `m(T3 *)`. This shows that our multi methods are different from the *encapsulated multi methods* of [BCC$^+$95, Cas97], in that the receiver and the first parameter participate together in the dynamic selection of the method as in [Cas95b], while in the case of encapsulated multi methods, the receiver has the precedence over the parameters. Moreover, in encapsulated multi methods, the (re)definition of a multi method in a subclass completely overrides the old one (see [BCC$^+$95]). This means that the branches defined in the superclass are not automatically inherited in a subclass that specializes (i.e., adds a branch) a multi method. For instance, in Figure 1, class B would not inherit the two branches of n defined in A, with encapsulated multi methods.

Let us observe that C++ scoping rules are still valid, so the programmer can rule the dynamic selection of a specific branch by using the `::` scope resolution operator. For instance, if in the code above we replace `a->m_DB(t)` with `a->A::m_DB(t)` we basically restrict the dynamic branch selection to the scope of class A. Thus, the specialized branch `B::m(T3 *)` will not be considered during branch selection. However, since dynamic binding is still employed for selecting the implementation of a specific branch, `B::m(T2 *)` will be selected (summarizing the three method invocations above will select `A::m(T1 *)`, `B::m(T2 *)` and `B::m(T2 *)` again, respectively). This holds because we implicitly consider each method as virtual. In the following we will show how the programmer can effectively specify whether branches are virtual or not.

[2]Instead of implicitly adding a method with a suffix, `_DB`, we could have introduced another keyword in the language, e.g., **ddinvok**`(x->m)`, or another operator, e.g., `x=>m`, with the semantics of double dispatch invocation; however, we think that our solution is simpler and, furthermore, it does not require to translate the code that invokes methods with double dispatch.

multi-method	::=	*virtual* **branches** *name branches* **endbranches**
branches	::=	*branch*
	|	*branch branches*
branch	::=	*virtual type* **(** *type* ***** *arg* **)** **;**
	|	*virtual type* **(** *type* ***** *arg* **)** **{** *body* **}** **;**
virtual	::=	**virtual** | ϵ

Table 1: Syntax for multi-method definition. *name* is the name of the multi-method, *type* is a class defined by the user and *method definition* is a standard C++ method definition.

The syntax for multi-method definition is in Table 1. Notice that in this version of `doublecpp` the first parameter of a multi-method branch must always be a pointer, and that all branches have to be terminated with `;`.

Declaring a multi method as virtual ensures that all the branches in the class hierarchy are considered during branch selection to execute the most specialized one. Declaring a single branch as virtual, instead, has exactly the same meaning of declaring a (overloaded) method as virtual in C++. Finally, we notice that, by using *virtual* for multi method and branches in appropriate combinations, one can implement several flavors of multi method semantics, even encapsulated multi methods (e.g., by not using *virtual* at all).

> IMPORTANT: In the previous releases of `doublecpp` all multi methods and branches were automatically considered virtual. For backward compatibility, and also for those who do not feel like specifying `virtual` for all their multi methods and branches, we now provide the option `--assume-virtual` (see Section 3).

C++ programs using this new linguistic construct have to be preprocessed by the `doublecpp` program, that will produce standard C++ code, with the same semantics described above. `doublecpp` will inspect both the classes using `branches`, their superclasses, and the classes used as parameters for branches of multi-methods (i.e., target classes). These classes can be spread in separate files and do not need to be written in the same file.

> **Remark.** `#include` headers are correctly inspected by `doublecpp`. However, only files that are specified in double quotes (`"`) are inspected, while those specified in `< >` are not. You can still force a file specified in `"` not to be inspected by using the command line option `--excludeheaders` explained later in Section 3, page 9.

By default, `doublecpp` is based on the following assumption:

> if a class `MyClass` has not already been found in an analyzed source, `doublecpp` will try to read a source with the same name of the class in lower case, e.g., `myclass.h` in this example.

If you do not feel comfortable with this "automatic" behavior and want to explicitly `#include` target class' header files, you can disable it with the command line option `--dont-infer-headers` (See Section 3 for the complete list of options). With this option, class `A` in Figure 1 must explicitly `#include` the header files for `Ti` and `Si`.

Notice that `doublecpp` should be called on the last class of the hierarchy (the most derived) that introduces a new branch or redefines a previously defined branch. Assuming that the classes for the example in Figure 1 are developed in the following sources:

A	`a.h` and `a.cc`
B	`b.h` and `b.cc`
T1, T2, T3	`t3.h` and `t3.cc`
S1, S2, S3	`s3.h` and `s3.cc`
V1	`v1.h` and `v1.cc`

```
// main.cc
#include "b.hpp"
#include "t3.hpp"
#include "s3.hpp"
#include "v1.h"

int main() {
  A *a = new B;
  T1 *t = new T1;
  a->m(t);    // static overloading
  a->m_DB(t); // dynamic overloading
  t = new T2;
  a->m_DB(t); // dynamic overloading
  t = new T3;
  a->m_DB(t); // dynamic overloading

  return 0;
}
```

Figure 2: The main source using classes in Figure 1

then `doublecpp` should be invoked as follows:

```
doublecpp -i b.h
```

This will generate the following files:

`a.hpp` and `a.cpp`	for A
`b.hpp` and `b.cpp`	for B
`t3.hpp` and `t3.cpp`	for T1, T2, T3
`s3.hpp` and `s3.cpp`	for S1, S2, S3
	nothing for V1

The `.hpp` header files contain the modified (preprocessed) version of the original classes, and thus they contain standard C++ code. This means that all the other sources of our project should include these files and not the original `.h` ones. The `.cpp` files contain the definition of the additional methods needed for implementing the double dispatch semantics, thus they are complementary to the `.cc`, i.e., both the `.cc` and `.cpp` sources are to be linked in the final program.

Finally, the code using the preprocessed code, e.g., the `main` of our program can be written as shown in Figure 2. Notice that, since `v1.h` was not preprocessed it can be included as it is.

Finally, the program can be built with the following command line:

```
g++ -o test main.cc a.cc b.cc t3.cc s3.cc v1.cc a.cpp b.cpp t3.cpp s3.cpp
```

NOTE: If this behavior may seem to require to know too many things about the classes of the program and may force to modify existing code using `T1`, ..., `S1`, ..., in Section 3 we will see how to instruct `doublecpp` so that fewer files are generated.

2 Installation

`doublecpp` comes with sources being under the GPL license, and can be downloaded from its home page:

`http://doublecpp.sourceforge.net`

First, you have to unpack the tarball file `doublecpp-x.x.x.tar.gz`, where `x.x.x` is the release version, in an appropriate directory. Then, once you entered that directory, it can be compiled and installed like any other GNU programs, i.e., with the typical command sequence:

```
./configure
make
make install
```

remember that by default this will install the program and all its files starting from the directory `/usr/local`, thus you have to be super user in order to do that. Otherwise, should you want to perform the installation in a different (possibly personal) directory, say `/myhome/usr`, you have to pass this option to the `configure` script:

```
./configure --prefix=/myhome/usr
```

Optionally, before `make install`, you may want to run `make check`, that tries to compile some programs preprocessed with `doublecpp`. Notice that you will experience problems if you have a version of `gcc` earlier than `3.x`, due to a non-standard treatment of `using` clause for explicitly inheriting overloaded methods from a super class (this is part of the code generated by `doublecpp`). So you need to install a more recent version of `gcc`.

You can also obtain the most recent sources via anonymous CVS (just hit enter when prompted for the password):

```
cvs -d:pserver:anonymous@doublecpp.cvs.sourceforge.net:/cvsroot/doublecpp login

cvs -z3 -d:pserver:anonymous@doublecpp.cvs.sourceforge.net:/cvsroot/doublecpp co -P doublecpp
```

If you obtain sources from CVS, before running configure, you must run the script `autogen.sh`; This will run the autotools commands in the correct order, and also copy possibly missing files. You should have installed recent versions of *automake* and *autoconf* in order for this to succeed. You will also need *flex* and *bison*.

3 `doublecpp` options

`doublecpp` supports the options reported in Table 2. In order to fully understand the following options please refer to the definitions in Section 1.

We will now go into details of the most relevant options:

`--force` : since `doublecpp` may have to preprocess many files, and since this may require the compilation of many sources in the program, by default, `doublecpp` will regenerate only the sources that actually need to (if the preprocessing results in a class that it is exactly the same as the one generated in a previous preprocessing it will not write the changes). This option allows to force the regeneration of sources.

`--invade-target` : if a class `C` defined in the source `c.h` is used only as the first parameter of a branch (and it does not define a multi-method itself), with this option the result of the preprocessing will not be written into `c.hpp` but directly in the original file `c.h`. This makes programming easier since the code that uses `C` can still safely include `c.h` even after another class uses it for a parameter of a branch of a multi-method. IMPORTANT: please make a backup of the original file, since with this option it will be directly modified (and in case of failures or bugs of `doublecpp` it may get corrupted). NOTE: this is the encouraged way of using `doublecpp`.

```
 -h, --help                       Print help and exit
 -V, --version                    Print version and exit
 -i, --input=FILENAME             input file (default std input)
 -v, --verbose                    verbose mode on
 -F, --force                      force regeneration of output code
     --invade-target              directly modify sources of targets
     --modular                    instead of modifying targets, generate RTTI
                                    tests and casts.  Thus targets are not
                                    modified
     --clean-targets              remove possible previously modifications
                                    applied to target classes.  It makes
                                    sense only when used together with
                                    --modular option
     --input-header-ext=STRING    input header file extension  (default=‘h’)
     --output-header-ext=STRING   output header file extension  (default=
                                    ‘hpp’)
     --output-source-ext=STRING   output source file extension  (default=
                                    ‘cpp’)
     --output-header-suff=STRING  output header file suffix (def: none)
     --output-source-suff=STRING  output source file suffix (def: none)
 -r, --rename-overloaded          rename overloaded methods  (default=off)
     --rename-suffix=STRING       suffix for renamed method  (default=‘_’)
     --rename-db-suffix=STRING    suffix for renamed double disp method
                                    (default=‘_DB’)
     --no-linenum                 do not generate #line statements
     --dont-infer-headers         do not use the name of the class to infer
                                    the header file
     --assume-multimeth-virtual   assume all multimethods as virtual
     --assume-branch-virtual      assume all branches of multimethods as
                                    virtual
     --assume-virtual             assume all multimethods and all branches of
                                    multimethods as virtual (i.e., equivalent
                                    to --assume-multimeth-virtual and
                                    --assume-branch-virtual)
     --stats                      print some statistics, e.g., number of
                                    scanned files
     --test-mode                  even with errors exit with 0 (this is only
                                    for testing purposes)
     --sourcepath=STRING          where to search for include files, and for
                                    files that can be modified
     --excludeheaders=STRING      header files that must not be processed
```

Table 2: `doublecpp` command line options

`--modular` : if you use this option the target classes will not be modified. Thus, the whole program will be modular in the sense that the target classes will not depend on the client classes. The generated code, however, will be less efficient, since the right branch will be selected using a cascade of **`if`** statements, employing RTTI (i.e., `dynamic_cast<>`).

`--clean-targets` : if you switch to modular mode, you should also use this option (at least the first time you use `--modular` in order to remove code inserted in the target classes. If you switch to modular mode after using `--invade-target` you should use also use `--invade-target` together with `--clean-targets` so that the original target sources are restored to their original form.

`--input-header-ext` : by default, as hinted above, `doublecpp` looks for a class not already analyzed in a source with the same name of class (in lower case) and extension `.h`; with this option it is possible to change this file extension.

`--output-header-ext` : the extension for the generated header file.

`--output-source-ext` : the extension for the generated C++ source file.

`--output-header-suff` : a suffix to be added to the generated header file (before the extension).

`--output-source-suff` : a suffix to be added to the generated C++ source file (before the extension).

`--rename-overloaded` : instead of creating a method `_DB` for a multi-method m, the original name m is used for the double dispatch semantics, while the branches are renamed. This option is discouraged.

`--rename-suffix` : suffix for the renamed overloaded methods (see `--rename-overloaded`).

`--rename-db-suffix` : the suffix used for creating the additional method with double dispatch semantics (the default is `_DB`).

`--no-linenum` : `doublecpp` inserts `#line` directives in the generated files, so that compiler errors refer to the original sources, and debugging is easier. If you wish to disable this, you can use this option.

`--dont-infer-headers` : By default, as explained in Section 1, if `doublecpp` encounters a class name that has not already been inspected, it will try to read a header file with the same name of the class in lower case; you can disable this "inference" behavior using this option (in such case you must explicitly `#include` all the appropriate header files, otherwise you will get an error).

`--assume-multimeth-virtual` : if all your multi methods must be virtual, and you do not feel like specifying the keyword `virtual` for them, you can use this option, and they will be automatically considered as virtual.

`--assume-branch-virtual` : as above, but for branches.

`--assume-virtual` : All multi methods and all branches of multi methods are considered virtual.

`--stats` : Print some statistics, such as the number of inspected files, classes and elapsed time.

`--test-mode` : is only for testing purposes; with this option, `doublecpp` will return 0 (i.e., success) even in case of errors (e.g., type errors), but not in case of failed assertions.

`--sourcepath` : `doublecpp` does not use environment variables of the C++ compiler, such as, e.g., `INCLUDE` to search for a header file. We do consider this as a feature, since we wouldn't want a system header file to be modified by `doublecpp`; thus we require the programmer to specify all the paths (the current directory is always inspected by default) where the program header files (that must be processed) are. Notice that you can use this option many times in the command line.

`--excludeheaders` : this allows to force `doublecpp` not to examine specific header files. As hinted in the remark at page 4 files specified in `< >` are not inspected anyway.

We believe that the most important option is `--invade-target` in that it can limit the number of files generated by the preprocessor, and does not require changes in sources that use target classes. For instance, let us consider again the distribution of code in source files for the example in Figure 1:

A	`a.h` and `a.cc`
B	`b.h` and `b.cc`
T1, T2, T3	`t3.h` and `t3.cc`
S1, S2, S3	`s3.h` and `s3.cc`
V1	`v1.h` and `v1.cc`

now if we invoke `doublecpp` in the following way:

```
doublecpp -i b.h --invade-target
```

The only files that will be generated will be:

`a.hpp` and `a.cpp`	for A
`b.hpp` and `b.cpp`	for B

as for the sources of target classes, the original header sources will be modified by `doublecpp`[3]. Thus, only the sources using class *A* and *B* need to refer to `b.hpp`, while sources using all the other target classes can still refer to the original header files. In particular, the `main` source of Figure 2 will now contain the following header include section:

```
// main.cc
#include "b.hpp"
#include "t3.h"
#include "s3.h"
#include "v1.h"
```

Finally, the program can be built with the following simpler command line:

```
g++ -o test main.cc a.cc b.cc t3.cc s3.cc v1.cc a.cpp b.cpp
```

As you may have understood, `doublecpp` by default modifies also the target classes (and if you use `--invade-target` it will actually modify the original sources of the target classes). If you do not like this behavior you can use the `--modular` command line option. Let us stress that the resulting code will be more modular (since the target classes will not depend on the client classes) but also less efficient since the right branch will be selected using a cascade of **if** statements, employing RTTI (i.e., `dynamic_cast<>`).

We suggest to use the modular mode during the development stage: less compilations will be required since there will be less dependences among classes (client classes will depend on the target classes, but not the other way round). You can then switch to standard mode before deploying your application, making it far more efficient.

> IMPORTANT: the code generated in standard mode is equivalent to the one generated in modular mode: they have the same semantics.

[3]This is the main reason why we suggest to keep a backup of the original header files, in case `doublecpp` crashes somehow during the modification of the header files.

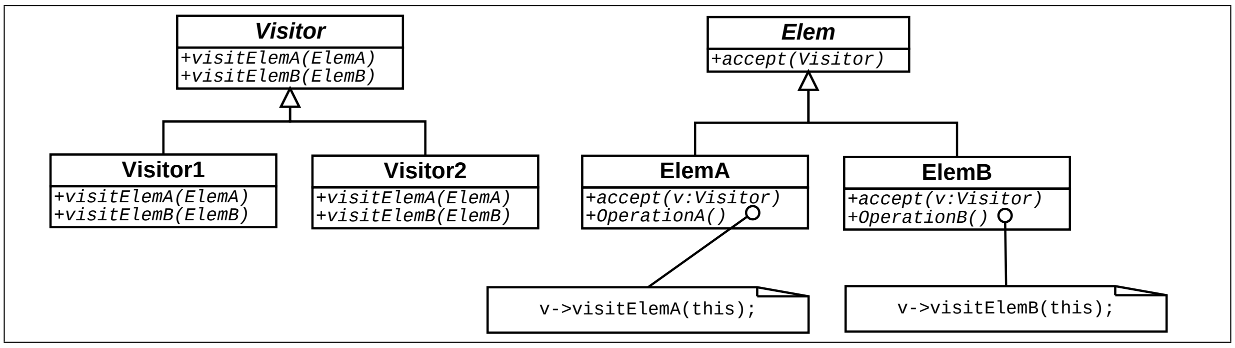

Figure 3: Visitor pattern structure.

Of course, if you switch to modular mode you should also take care of "cleaning" the target classes from previously generated code; you can easily do this by invoking `doublecpp` with `--clean-targets` the first time you use `--modular`. If you used to use `--invade-target` option, you should use it also when you use `--clean-targets`. Thus, in the previous example, if you wish to switch to modular mode you have to invoke:

```
doublecpp -i b.h --invade-target --modular --clean-targets
```

4 Some examples

A typical example where dynamic overloading is very useful is when there are two separate class hierarchies and the classes of a hierarchy have to operate on instances of classes of the second hierarchy according to their dynamic types. This is quite a recurrent situation in object-oriented design, as hinted in the introduction, since separation of responsibilities enhance class decoupling and thus re-usability. In such situations, in order to avoid the awful use of RTTI and type casts, the design pattern *Visitor* [GHJV95] can be used. The structure of the pattern is illustrated in Figure 3.

The idea behind this pattern is that the base class `Visitor` defines all the methods that have to operate on the elements of the second hierarchy. The classes of the `Elem` hierarchy implement the method `accept` by calling the method of the visitor corresponding to their actual type, passing **this** as argument. Then, a client can perform an operation, implemented by a specific visitor subclass, by invoking the method `accept` on an element:

```
Elem *elem = new ElemB;
Visitor *visitor = new Visitor1;
elem->accept(visitor);
```

This way, the method `Visitor1::visitElemB` will be finally executed, i.e., the method of the actual class of the visitor corresponding to the actual class of the element. It is then easy to observe that the Visitor pattern is basically a way of implementing the behavior of dynamic overloading and double dispatch in a language that does not provide it.

While the result is basically the same of dynamic overloading, it requires additional effort to the programmer, that has to explicitly program the dynamic dispatch mechanism. Furthermore, and this is one of the main drawbacks of this pattern, a cyclic dependence raises from the structure of the pattern, i.e., the visitor classes depend on all the concrete classes of the `Elem` hierarchy, and, on the other hand, the `Elem` hierarchy depends on the `Visitor` hierarchy (because of the method `accept`). Introducing a new `Elem` to handle requires changing the `Visitor` hierarchy. Introducing a new operation requires introducing a new `Visitor` hierarchy. Both cases require a recompilation of the entire `Elem` hierarchy. A

```
class Visitor {
 public:
  virtual branches visit
   virtual void (ElemA *t);
   virtual void (ElemB *t);
  endbranches
};
```

```
class ExtVisitor : public Visitor {
 public:
  virtual branches visit
   virtual void (ElemA *t);
   virtual void (ElemC *t);
  endbranches
};
```

Figure 4: An implementation of the visitor in C++ with double dispatch

```
class Point {
  int x, y;
 public:
  virtual branches equals
   virtual bool (Point *p)
   { return
     x == p->x &&
     y == p->y; };
  endbranches
};
```

```
class ColorPoint : public Point {
  string color;
 public:
  virtual branches equals
   virtual bool (ColorPoint *p)
   { return
     Point::equals(p) &&
     color == p->color; };
  endbranches
};
```

Figure 5: An implementation of the binary method `equals` in C++ with double dispatch.

variant of the pattern, called *Acyclic Visitor* was proposed [Mar98] in order to limit these dependences, but the solution makes use of dynamic casts

With our proposed language extension, the visitor classes can be easily programmed by defining a multi-method `visit` with a branch for each `Elem` that has to be handled, as illustrated in Figure 4. The `Elem` hierarchy does not have to be modified by the programmer: all the internal issues will be handled by the preprocessor of the construct for multi methods. The programmer can visit an element by simply calling `visit_DB` on any `Elem` instance. Furthermore, a subclass of `Visitor` can add a branch for a new `Elem` subclass (as `ExtVisitor` for `ElemC`) without affecting the base class (covariant specialization); in the Visitor pattern this cannot be done smoothly, without resorting to casts.

Thus, with double dispatch, Visitor ceases to be a design pattern and becomes more a good programming technique that allows to abstract operations from the elements these operations are to be performed on. This is the case, for instance, of a compiler: the visitor classes are all the classes performing several controls on the abstract syntax tree, and the nodes of the tree are the elements that are to be visited.

Dynamic overloading and covariant specialization come in hand also in implementing many other design patterns, that, as already hinted, are often based on the collaboration of separate class hierarchies. Another example is the pattern *Observer* [GHJV95], where the instances of classes of the first hierarchy (the *observables*) notify the instances of another hierarchy (the *observers*) when some changes took place. The observable instance typically passes itself to the observer, through a method called `update`; the parameter of this method is of type `Observable`, the base class of all the observable classes. Thus, in order to perform useful operation in the method `update`, in case the language does not provide double dispatch, the observer classes typically have to check the dynamic type of the observable and perform type casts.

Double dispatch also enables safe *covariant specialization* of methods, i.e., subclasses are allowed to redefine a method also by specializing its argument (we refer the reader to [Cas95a] for a clear treatment of this subject). A typical application of covariant specialization is *binary methods* [BCC$^+$95], i.e., methods that act on objects of the same type: the receiver and the argument.

Also binary methods can be easily implemented with multi methods as suggested in [BCC$^+$95]. For instance, exploiting the usual example of `Point` and `ColorPoint`, we can implement the method `equals` as illustrated in Figure 5. Equality of points can be tested smoothly as follows:

```
Point *p1 = new Point;
Point *p2 = new ColorPoint;
Point *p3 = new ColorPoint;
p1->equals_DB(p2); // (1) invoke Point::equals(Point *)
p2->equals_DB(p1); // (2) invoke Point::equals(Point *)
p2->equals_DB(p3); // (3) invoke ColorPoint::equals(ColorPoint *)
```

Notice that all the branches of the same multi method are implicitly inherited by the subclass, so `ColorPoint` is able to handle also `Point` instances passed to `equals`: in this case the implementation `Point::equals(Point *)` will be called. However, the programmer of `ColorPoint` may want to consider a `ColorPoint` and a `Point` different (since the latter has no color); in that case he can simply redefine that branch in `ColorPoint` as follows

```
branches equals
bool (Point *p) { return false; }
...
endbranches
```

Of course, in this case, the second invocation in the previous code snippet would invoke `ColorPoint::equals(Point *)`.

Let us observe that it is crucial to have the possibility of using both dynamic overloading (using `_DB` methods) and static overloading, as in the case of `ColorPoint::equals(ColorPoint *)` that relies on the implementation of `Point::equals` by using static overloading.

5 Advanced issues

Branches of a multi method can actually be seen as standard C++ overloaded methods apart from, as hinted before, that they are implicitly inherited in subclasses even in case the subclass introduces a new branch[4]. However, the first parameter assumes a more important role, since it is used for double dispatch when the `_DB` corresponding method is called. The other possible parameters do not participate in double dispatch but they still have an important role in static overloading.

In particular, a `_DB` is created with the first parameter of a type that is the super type among the all the types used for declaring the first parameter of the branches of a multi method (considering also the branches of the same method defined in the superclasses); for instance, considering the example in Figure 1, the methods `m_DB(T1 *)` and `n_DB(S1 *, V1 *)` are created for the classes `A` and `B`, since `T1` is the topmost class among `T2` and `T3` (and `S1` is the topmost class among `S2` and `S3`). Notice that, since only the first parameter is used for dynamic selection, the other parameters, e.g., `V1` in this example, are not specialized and are all the same in the branches of `n`. However, the programmer is still allowed to declare "unrelated" branches of the same multi method, i.e., the following code would still be accepted:

```
  branches n
   S (S1 *t, V1 *v) { ... };       // (1)
   S (S2 *t, V1 *v) { ... };       // (2)
   T (S2 *t, V2 *v) { ... };       // (3)
   T (S3 *t, V2 *v) { ... };       // (4)
   D (T1 *t, C *v, D *x) { ... };  // (5)
  endbranches
```

[4]In C++ you should use the clause **using** in order to make the overloaded versions of an overloaded method *m* visible in a subclass, in case the subclass introduces a new version for *m*.

and the following methods are created:

- `S n_DB(S1 *, V1 *)` for (1) and (2)
- `T n_DB(S2 *, V2 *)` for (3) and (4)
- `D n_DB(T1 *, C *, D *)` for (5)

thus, static overloading is still used when one calls one of these methods, according to the static type of all the arguments; then the most specialized one is dynamically selected using the actual dynamic type of the first argument[5]:

```
A *a = new A;
S1 *s1 = new S2;
S2 *s2 = new S3;
V1 *v1 = new V2;
V2 *v2 = new V2;

a->n_DB(s1, v1);
// statically select S n_DB(S1 *, V1 *)
// dynamically select S n(S2 *, V1 *)
a->n_DB(s1, v2);
// statically select S n_DB(S1 *, V1 *)
// dynamically select S n(S2 *, V1 *)
a->n_DB(s2, v2);
// statically select S n_DB(S2 *, V2 *)
// dynamically select S n(S3 *, V2 *)
```

As a degenerate example let us consider the following one:

```
class A {...};
class B : public A {...};
class C : public A {...};

class D {
 branches m
 void (B *b) {...};
 void (C *b) {...};
 endbranches
};
```

`B` and `C` are not related, they are simply sibling classes since they inherit from the same class. For this reason, two methods are created:

```
void m_DB(B *);
void m_DB(C *);
```

Thus, basically, this case reduces to standard overloading even when using `m_DB` (unless this multi method is further specialized in the subclasses with subclasses of `B` and `C`). Probably, what the programmer wanted to write is:

[5]The example can be found in the file `slightly_diff.h` in the directory `tests`.

```
class D {
 branches m
 void (A *b) {...};
 void (B *b) {...};
 void (C *b) {...};
 endbranches
};
```

In fact, in this case, only one method, `m_DB(A *)`, is created, and double dispatch can be exploited.

5.1 Automatic code inspection

There can be many cases where you have two different unrelated client class hierarchies, say `Client1` and `Client2`, that use the same target classes, say `Ti`, in their multi methods. It is likely that these client hierarchies are in separate files, say `c1.h` and `c2.h`, respectively. The question may raise: Which client header file should I preprocess first? If I preprocess `c1.h` first, and then `c2.h`, will the previous modifications made to header files of the `Ti` be lost? Fortunately, the answer is no.

Indeed, when `doublecpp` inspects a file, before actually modifying any source (or generating new files), it inspects possible previously generated output files (That's why you should always be consistent in using different header file extensions for input and output header files – see the options `--input-header-ext` and `--output-header-ext` in Section 3). According to the "traces" that `doublecpp` left in the generated files, it will not overwrite modifications required before by other client class files.

The modifications made by `doublecpp` in the output files are surrounded by comment blocks of the shape:

```
// doublecpp: ...<some explainations>..., DO NOT MODIFY
...
// doublecpp: end, DO NOT MODIFY
```

Of course, it is crucial that you never modify these blocks of code, otherwise, on a successive invocation of `doublecpp` you can experience problems.

6 Known limitations

`doublecpp` is free software and it comes with sources being under the GPL license. If you feel like adding a feature, or remove some of the limitations below, the author welcomes patches :-)

- In this version of `doublecpp` the first parameter of a multi-method branch has to be a pointer type (references are not handled yet).
- Argument types or super classes that have nested templates are not handled yet.
- `const` methods are not handled yet.
- There are problems when parameters in a multi method branch are not classes but typedefs (for instance `ostream`). In particular you cannot use such types in any parameter of a multi method.
- Possible ambiguities and type errors will be reported by the C++ compiler itself, not by `doublecpp`; indeed `doublecpp` reduces error checks to the minimum, relying on the C++ compiler (we don't think this is a limitation anyway). However, the `#line` directives inserted by `doublecpp` in generated files will make the compiler errors refer to the original sources, thus understanding where the error is should be quite easy.

References

[BC97] John Boyland and Giuseppe Castagna. Parasitic Methods: Implementation of Multi-Methods for Java. In *Proc of OOPSLA '97*, volume 32(10) of *ACM SIGPLAN Notices*, pages 66–76. ACM, 1997.

[BCC$^+$95] K. B. Bruce, L. Cardelli, G. Castagna, The Hopkins Object Group, G. Leavens, and B. C. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):217–238, 1995.

[BCV05] L. Bettini, S. Capecchi, and B. Venneri. Translating Double-Dispatch into Single-Dispatch. In *Proc. of Int. Workshop on Object-Oriented Developments (WOOD) 2004*, volume 138 of *ENTCS*. Elsevier, 2005.

[BCV06] L. Bettini, S. Capecchi, and B. Venneri. Double Dispatch in C++. *Software – Practice and Experience*, 36(6):581 – 613, 2006.

[Cas95a] G. Castagna. Covariance and contravariance: conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, 1995.

[Cas95b] G. Castagna. A meta-language for typed object-oriented languages. *Theoretical Computer Science*, 151(2):297–352, 1995.

[Cas97] G. Castagna. *Object-Oriented Programming: A Unified Foundation.* Progress in Theoretical Computer Science. Birkhauser, 1997.

[DG87] L.G. DeMichiel and R.P. Gabriel. The Common Lisp Object System: An Overview. In *Proc. ECOOP*, volume 276 of *LNCS*, pages 151–170. Springer, 1987.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[Mar98] Robert C. Martin. Acyclic Visitor. In Martin, Riehle, and Buschmmann, editors, *Pattern Languages of Program Design 3*, pages 94–104. Addison-Wesley, 1998.

[MHH91] W.B. Mugridge, J. Hamer, and J.G. Hosking. Multi-Methods in a Statically-Typed Programming Language. In P. America, editor, *Proc. ECOOP '91*, volume 512 of *LNCS*, pages 307–324. Springer, 1991.