Managing Code Dependencies in C#

Riccardo Casero[1], Mirko Cesarini[1], and Mattia Monga[2]

[1] Politecnico di Milano, Dipartimento di Elettronica ed Informazione,
Piazza Leonardo da Vinci 32, I-20133 Milano, Italy
cesarini@elet.polimi.it, vant@inwind.it

[2] Università degli Studi di Milano, Dip. Informatica e Comunicazione,
Via Comelico 39/41, I-20135 Milano
mattia.monga@unimi.it

Abstract. Most modern object oriented programming languages do not offer constructs to specify dependencies among members of a class. Public interfaces are written using member types and method signatures only, which are not capable of expressing such kind of relationships. We show that stating which dependencies exist between class members, i.e. which methods could be affected by a change in the implementation of the others, constitutes a relevant information to be shipped to inheritors in order to help them in subclassing without inconsistencies. In this paper we present a tool that supports developers in this task by exploiting C# attributes, that are annotations accessible at runtime. The tool will be integrated in the popular developer environment Visual Studio .NET.

1 Introduction

Object-oriented systems are built upon the information hiding principle: a class is made by data and routines, but only a subset of them is available to external programmers. What can be used of a class is stated within the *module interface*, and access to private members is not granted. The module interface is a sort of use contract between the class and its users. Two kind of users can be profiled: the ones that simply need to employ the class as it is, and the ones that need to subclass it. We will call hereafter *clients* the first set of users and *inheritors* the second one. The module interface provided by most OOP languages is supposed to fulfil both the two sets of users by using member types, methods signatures and visibility modifiers. However, these linguistic constructs –if well suited for clients– are somehow not enough for inheritors. In fact, the module interface states what can be used by clients or inheritors as is and what they can adapt to their needs by providing actual parameters or by overriding. Furthermore the semantic of a method can be documented by stating under which conditions (*preconditions*) the method is guaranteed to produce a well defined effect (*postconditions*). Method signatures together with pre/post conditions provide enough information to statically catch most of the type errors, even in the case of dynamic bounded languages. This helps clients in writing their code, however, as far as inheritors are concerned, this provided information is poor. In particular,

it might happen that, in order to preserve a correct semantics, the overriding of a method requires other methods to be rewritten as well . For example suppose that we have the class `MySet` (shown in Tab. 1).

```
public class MySet{

  private ArrayList hidden_rep;
  public delegate void Action(object o);

  public MySet(){
    hidden_rep=new ArrayList();
  }

  public virtual void Add(object o){
    hidden_rep.Add(o);
  }

  public virtual bool RemoveIfPresent(
     object o){
    if(!hidden_rep.Contains(o))
      return false;
    hidden_rep.Remove(o);
    return true;
  }

  public virtual void ForEach(Action a){
    IEnumerator i=hidden_rep.GetEnumerator();
    while(i.MoveNext()) a(i.Current);
  }

  public virtual void AddAll(MySet s){
    Action a=new Action(this.Add);
    s.ForEach(a);
  }

  public virtual void Remove(object o){
    bool removed=RemoveIfPresent(o);
    if(!removed)
      throw new Exception("
 Object not present");
  }
}
```

```
public class MyCountedSet:MySet{

  private int cardinality=0;

  public override void Add(object o){
    base.Add(o);
    cardinality++;
  }

  public override bool
          RemoveIfPresent(object o){
    if (base.RemoveIfPresent(o)){
      cardinality--;
      return true;
    }
    return false;
  }

  public int Cardinality{
    get{
      return cardinality;
    }
  }
}
```

```
public class MyEvenSet:MyCountedSet{

  public override void Add(object o){
    if (this.Cardinality % 2 == 0)
      (MyCountedSet)this.Add(o);
  }

}
```

Table 1. Source code of MySet, MyCountedSet and MyEvenSet classes.

The methods `AddAll` and `Remove` rely on the other methods `Add`, `ForEach` and `RemoveIfPresent` which in turn rely on the hidden representation (i.e. the set of all fields declared in the class).

Suppose now we derive a new class `MyCountedSet` from `MySet` (see Tab. 1). Since a new field `cardinality` is declared, the hidden representation of the class is changed and potentially every method relying on it should be changed in order to maintain a consistent behaviour. In this case only `Add` and `RemoveIfPresent` requires overriding.

Another example is the further derived class `MyEvenSet` (see Tab. 1). A general inheritor may think it is still possible to obtain an even set by using the

`AddAll` method, but since it relies on `Add` it would add at most only one element. Instead an inheritor informed about the dependency link between `AddAll` and `Add` would consider to change both. Hence the need to specify in the inheritors interface the dependencies among different members of a class.

This paper is organised as follows: Section 2 illustrates how we choose to specify dependencies and the rationale behind that. Section 3 describes the tool that we built, that is able to extract and show such information to developers. In Section 4 future work and final remarks are listed.

2 Introducing Dependency Information in the Module Interface

2.1 Adding Dependency Information

Inheritors should know about the dependencies among methods. In order to cope with this problem Lamping [4] proposed to enrich module interfaces with information reporting the dependencies among class features. The rationale behind this suggestion is to provide inheritors with enough information about how the features of a class combine to produce its overall behaviour, so that programmers fully understand the consequences of overriding.

The commonly used approach is to insert comments expressing dependencies directly in source code [2] [6]. In this case a lexical-analyser would read source code and recognise comments expressing dependencies. The main drawback of this approach is that it relies on the availability of source code, while the usefulness of dependency checking arises mainly when the programmers want to evolve binary components of which they know only the interface information [8]. Several component frameworks (JavaBeans, COM) owe their success to the ability of deploying binary, third party developed objects, about which users knows only public member signatures.

In a previous work [3], we proposed to exploit a new opportunity offered by the .NET [9] runtime platform. Such a platform provides support for *attributes,* which are annotations associated with syntactic elements of a program: classes, members, method parameters, etc.. Attributes in the .NET platform have the great advantage of being metadata and hence directly shipped with the assembly, without the need of exchanging source code [3]. Custom defined attributes can be added and can be easily retrieved at runtime through the reflection services provided by the .NET framework (the typical way of retrieving attributes is to fetch the object that represent the entity we are interested in and to invoke the `GetAttributes`() method on it).

[3] In the .NET jargon an assembly is the unit of deployment, containing one or more binary modules. Each assembly contains virtual machine instructions (data) and information related to them (metadata): version numbers, character set used in strings, etc..

2.2 Using C# Attributes to store Dependencies

Since the inheritor interface is composed of public and private methods and properties, we defined a `Dependency` attribute, that can be applied there. The syntax of the attribute is the following:

[`Dependency` (`type` , `called_method_name` , `params_type` ,`type_of_call`)]

This line, inserted just before a method `M` declaration, states that inside the body of `M` there is a call on an object of static type `type` towards the method `called_method_name` with parameters of types specified in the array `params_type`. The last argument specifies the type of call. We define three type of call:

- `SELF` denotes that the receiver of the invocation is the same object which performs it.
- `FORCED` denotes that the dynamic type of the receiver is forced to be the one specified (e.g. through a cast operation).
- `OTHER` denotes all other types of call.

The rationale behind this distinction is to help narrowing the possible execution paths to be computed. For instance knowing that a call is a `SELF` call could allow the tree-generator cut some branches. For example, consider the classes `A1` and `A2` in Fig. 1.

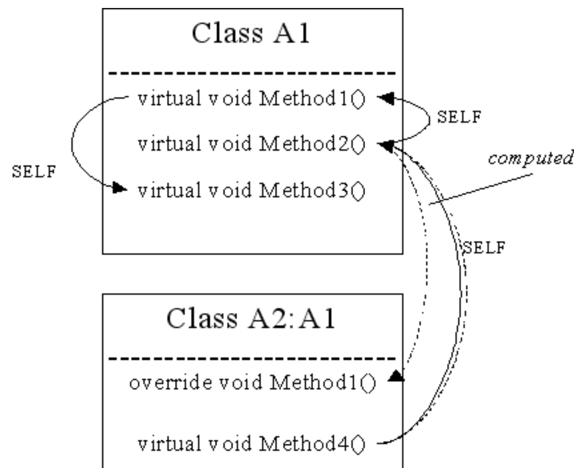


Fig. 1. An example of method calls (straight lines represent dependencies and the dashed line the path computed from `Method4`).

`Method3` influences `Method1`, that in turn influences `Method2`, that in turn again influences `Method4`. However, if one overrides `Method3` in a class `A3` –

derived from A2– she does not need to override Method4 since the dependency-chain is SELF and Method1, that was overridden in A2, does not call Method3 anymore. Suppose that instead of being a SELF call the call from Method2 to Method1 were FORCED: the other branch should have been considered. In the general case both branches should have been added to the invocation tree since we statically do not know the type of the object on which Method2 will be invoked.

In the case of a direct dependency on the hidden representation of a class (i.e. the set of all variables declared in the class) the syntax is simply:

```
[Dependency (type , special_hidden_dependency, type_of_call)]
```

3 Coding the Idea: an Add-in for the Visual Studio IDE

Visual Studio .NET is the primary IDE [4] for the .NET platform. It provides an add-in framework to build tools that easily integrates with the environment. Independent software vendors (ISVs) can implement new features (e.g. groupware, profiling tools, work flow, or life-cycle tools) that fit into Visual Studio .NET as seamlessly as if they were built in.

We exploited this feature to build our tool, that is able to retrieve and show the dependency attributes relevant for the code currently displayed. Even if it is fully integrated with Visual Studio, the tool is portable and it can be used as a stand-alone application.

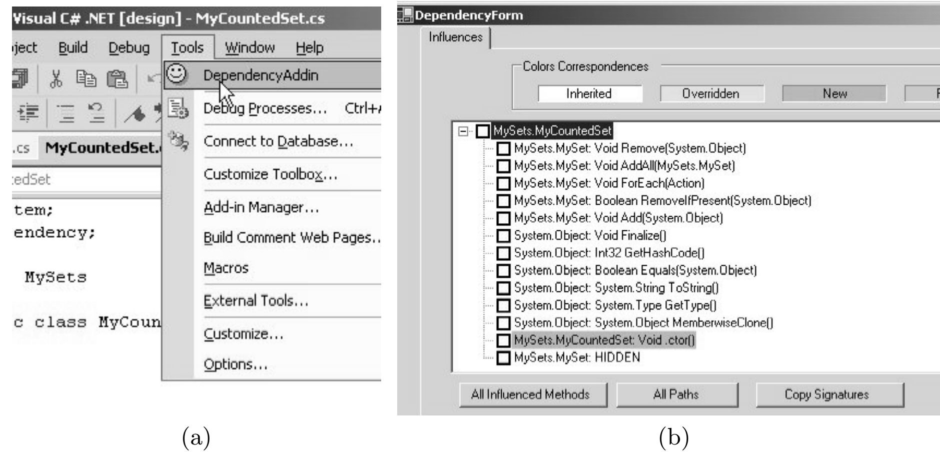
Suppose the MySet class shown in Tab. 1 has been created and annotated with Dependency Attributes. A programmer who wants to inherit it will have to declare and build the derived class, in the example the MyCountedSet class. Before overriding any method, she can check dependencies using our DependencyAddin (see Fig. 2(a)). A window showing all members of the newly created subclass will appear (see Fig 2(b)). Inherited members are correctly showed, their complete names tell also the base class they were declared in.

The chance is given to programmer to view dependencies in a flat fashion (the set of all members directly or indirectly influenced) or to view all possible paths of execution toward a given member. In the example all possible paths down to the hidden representation are shown (see Fig. 3). Checked members are the ones the programmer wants to override (or augment in case of the hidden representation). She can copy signatures to the source document in the environment and implement desired changes.

4 Concluding Remarks and Further Work

Reusability of components is a central issue in achieving quality and productivity of software development. Specifying dependencies among features could help programmers to correctly use and adapt components to their needs. In

[4] Integrated Development Environment



(a) (b)

Fig. 2. DependencyAddin command(a) and tool window(b).

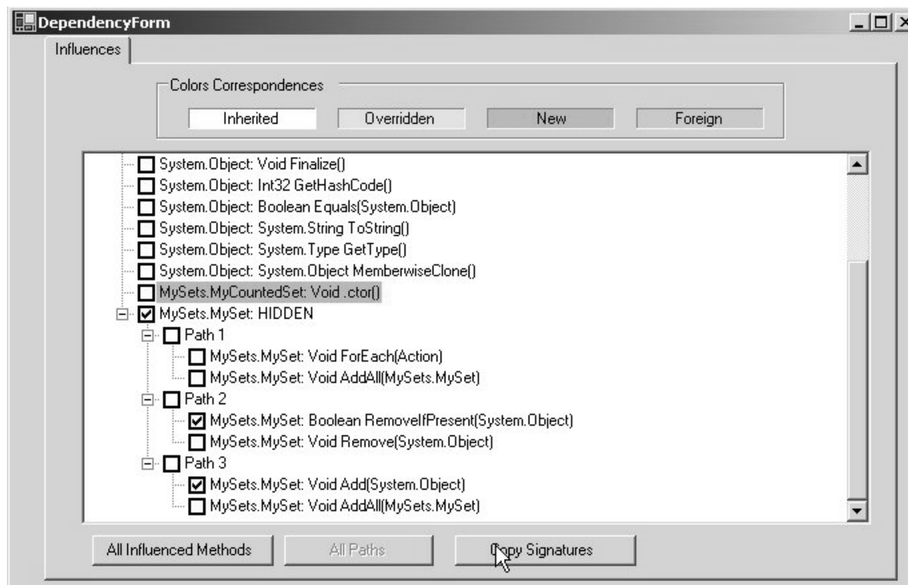


Fig. 3. A view of possible paths, calls are to be read bottom-up

particular inheritors of classes could use such information to infer how far a modification to a method would propagate and prevent undesired behaviours. On the other hand there is a fast-growing need of easy deployment of binary components, about which only public interface is known. The .NET framework permits to ship dependencies information directly with the binaries exploiting

attributes. In this paper we showed how to specify these attributes and a tool is presented that extracts attributes-defined dependencies from components and help programmers in sub-classing.

We are currently working in two directions. While useful, documenting the code with dependency attributes can be boring and error prone for programmers, mainly because the number of dependencies can be huge even in medium size source codes. However, most of them might be retrieved by a suitable tool. This tool will behave very similarly to a compiler in recording the syntax-tree of the code and in visiting it to resolve methods invocations and fields accesses. We're working to adapt part of the open source code of the Mono compiler [1] to reach this objective. Code will be then semiautomatically instrumented.

The actual set of dependencies cannot be statically computed, due to the features of OO languages, and C# in particular: the dynamic binding mechanism, delegates [5] and the capability to invoke methods whose names are dynamically build at runtime. Dependencies of these kinds of invocations cannot be retrieved statically. However, it is possible to compute sensible suggestions to programmers.

We are also investigating another approach using dependency attributes (see also [7] [5]): instead of being code-retrieved, dependencies could express imposed design decisions, thus realizing strong data abstraction. Investigation on this path is still needed.

Our experience shows that the new features provided by the .NET platform could be used to support developers in writing and reusing existing components. We think that our approach could be a step to make design by contract even more popular among developers, hence improving the average quality of the code.

References

1. Mono: an open source common language infrastructure implementation. `http://www.go-mono.com`.
2. D. Bartetzko, C. Fischer, M. Moller, and H. Wehrheim. Jass - Java with assertions. In *Workshop on Runtime Verification held in conjunction with the 13th Conference on Computer Aided Verification, CAV'01*, 2001. Published in *Electronic Notes in Theoretical Computer Science*, K. Havelund and G. Rosu (eds.), 55(2), 2001. Available from `www.elsevier.nl`.
3. C. Ghezzi and M. Monga. Fostering component evolution with c# attributes. In *Proceedings of the International Workshop on Principles of Software Evolution IWPSE 2002*, Orlando, Florida, May 2002. ACM.
4. J. Lamping. Typing the specialization interface. *ACM SIGPLAN Notices*, pages 201–214, 1993. OOPSLA'93.
5. K. R. M. Leino and G. Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(5):491–553, 2002.
6. C. Ruby and G. Leavens. Safely creating correct subclasses without seeing superclass code. In *OOPSLA 2000*, Minneapolis, Minnesota, Oct. 2000. ACM.
7. P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt. Reuse contracts: Managing the evolution of reusable assets. In *OOPSLA '96 Conference on Object-Oriented Programming Systems, Languagges and Applications*, pages 268–285. ACM Press, October 1996.
8. C. Szyperski. *Component Software — Beyond Object-Oriented Programming*. Addison Wesley Longman Limited, 1998.
9. E. TC39/TG3. Common language infrastructure. Technical report, ECMA, 2001.

[5] C# name for typed function pointers