Managing Evolution Using Cooperative Designs and a Reflective Architecture

Emiliano Tramontana

Department of Computing Science,
University of Newcastle upon Tyne,
Newcastle upon Tyne, NE1 7RU, UK
Emiliano.Tramontana@newcastle.ac.uk

Abstract The separation of concerns is important to attain object oriented systems which can be easily evolved. This paper presents a reflective architecture which enforces the separation of concerns by allocating functional, interaction and synchronization code to different levels. A variant of collaborations (CO actions) is used to capture interactions between objects and avoids spreading the description of interactions among the participating objects. Functional and interaction code are also separated from synchronization code by means of metalevel components. Introducing changes into the reflective architecture to consider evolution needs is facilitated by the loose coupling of different concerns. Hence, changing a concern often consists of modifying only one component of the reflective architecture. The paper describes the reflective architecture in terms of a case study. The evolution of the reflective implementation of the case study is compared with the evolution of an alternative implementation and the benefits of the proposed architecture are shown by using an evolution metric.

1 Introduction

Object oriented systems consist of a collection of interacting objects. In such systems interactions are mixed with functional code and spread among several objects. Therefore, both objects and interactions are difficult to express and reuse. When those systems need to evolve, the code is difficult to understand, since it mixes functionalities of objects with their interactions, and difficult to modify, since a change in an object might cause all the interacting objects to change.

Several approaches, which can be grouped under the name of *collaborations* (or *contracts*), have been proposed to describe sets of interactions between objects, and to avoid interactions being scattered among objects [24,29]. However, most of these approaches do not enforce a clear separation between functional and interaction aspects, so changing one implies also changing the other [2,3,14]. Other approaches which achieve such a separation require extending the existing object oriented model, thus compromising the feasibility of implementations using standard object oriented languages [12,19].

This paper uses *cooperations* to represent interactions between objects, and a reflective architecture for implementing software systems using objects and cooperations [8,9]. In a *cooperative object oriented* design relationships between objects are only captured by cooperations. A reflective implementation of a cooperative object oriented design allocates cooperations at the metalevel, thus enhancing the evolution of software systems, and providing a means of implementing the control of access to objects transparently from the objects [10,28]. The aim of the present paper is to show, using a case study, how the reflective architecture supports evolution. Some evolution scenarios are analysed for the case study and an evolution metric is presented to quantify the effort to change the reflective system. The evolution scenarios are compared with the equivalent evolution of an alternative implementation of the case study.

This paper is an extended version of [28] which presents the detail of a reflective architecture implementing a cooperative object oriented design, and where the enhancement provided by the reflective implementation is quantified by an evolution metric. A different version of the case study was previously presented in which evolution was discussed in the context of a cooperative object oriented design [27].

The paper is structured as follows. Section 2 describes the cooperative object oriented design and how it enhances evolution. Section 3 introduces the relevant background concerning reflection and the motivation for using reflection in our approach. Section 4 shows the cooperative object oriented design of a case study and its reflective implementation. Section 5 analyses evolution scenarios for the case study and compares the reflective implementation with an alternative one. The related work is presented in section 6 and eventually conclusions are presented in section 7.

2 Cooperative Object Oriented Design and Evolution

Object oriented systems consist of a collection of interacting objects. Interactions are usually described within objects, thus a complex set of interactions is difficult to express since it is scattered among objects. In a *cooperative object oriented (COO)* design, the object oriented model is extended by explicitly representing collaborative activities between objects which are expressed in terms of *cooperative actions (CO Actions)* [8,9]. Instances of CO actions are called *cooperations*. Using the COO design, objects are employed to model components' behaviour, and cooperations are used to model interactions between objects. A CO action specifies the collaborative activity of its participating objects and the pre-conditions, invariants and post-conditions associated with a collaborative activity. These express the respective conditions for a group of objects to start, hold and finish a cooperation.

Classes of a COO design are not aware of each other. The behaviour of a class is described only in terms of its own methods. This yields the construction of classes which are easy to understand and modify, since they do not mix functional and interaction code. Classes are also likely to be reused, because they are not specialised with interaction code. Similarly, since interactions between classes are described as CO actions and clearly separated from classes, they are easy to express and likely to be reused.

Changes that reflect evolving needs of a system can be incorporated only in the definition of new interactions among the existing components of the system. Hence, a COO design can evolve by changing only its CO actions, while classes remain unchanged. The diagram in figure 1 represents a COO design. Boxes illustrate classes, rounded boxes illustrate CO actions and lines connect a CO action with its participating classes. The three classes of the diagram participate to CO action `COActionA`, and its evolution attained by defining CO action `COActionB`. Consider a system which handles data for flight reservations. The classes of figure 1 then represent flights, customers, and tickets. CO action `COActionA` is responsible for booking tickets and thus describes interaction between those classes. Evolution of the system may necessitate changing the rules for booking a ticket, for example allowing a flight to be held for a while without being paid. That evolution scenario is incorporated into a new CO action (`COActionB`) describing a new set of interactions between the classes. Those classes remain unchanged when defining the new CO action.

A COO design can also evolve by changing classes. The CO actions where the changed classes participate can still be used as long as the methods used and the roles the classes play remain unchanged.

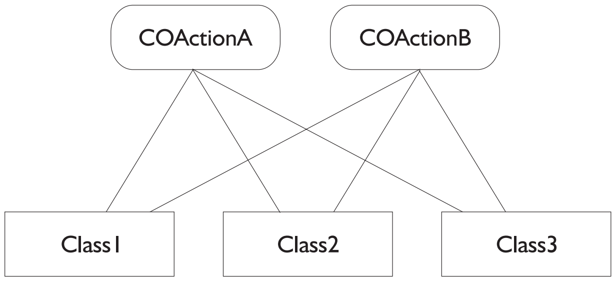


Fig. 1. Evolution of a COO design.

3 Reflective Architecture for Supporting Evolution

A reflective software system is a software system which contains structures representing aspects of itself that enable the system to support actions on itself [23]. A reflective architecture consists of a baselevel and a metalevel, where objects and metaobjects are respectively implemented. Two major reflective approaches have been identified [5,13]: communication reification and metaobject model. In the former approach the metalevel represents and operates on messages between objects, whereas in the latter the metalevel represents and operates on objects.

In the metaobject model, metaobjects are instances of a class able to intercept messages sent to their associated objects. The interception of messages allows metaobjects to perform some computation on the messages before delivering them to the objects, therefore making it possible to intertwine different concerns. The execution of an object's method can be suspended to verify constraints, to achieve synchronization, etc. [1,17].

A key concept in reflection is that of transparency: in a reflective system baselevel objects are not aware of the presence of metalevels above them. Thus, the development of objects is independent of that of metaobjects, and the connection of metaobjects to objects is performed without changing any of them.

In a COO design, objects and cooperations are easier to evolve and more likely to be reused because they only describe functional and interaction code, respectively. Evolution of COO systems can be attained by assuming that objects remain unchanged while cooperations can be modified or replaced. In the reflective implementation of a COO design, objects and cooperations are located respectively at the baselevel and metalevel. Different levels of a reflective architecture can be used to implement different parts of the same application, as in the Operating Systems Apertos [21] and Mach [25]. The metalevel adds functionalities to the baselevel, since the behaviour of the latter is altered by metalevel computation.

Reflection enhances evolution by allowing aspects of objects to be customised at the metalevel [18], and by encapsulating into the metalevel the part of an application more likely to change when new requirements are taken into account [4]. Since cooperations are located at the metalevel, this encapsulates the part of a COO system which is more likely to change when such a system evolves. Replacing cooperations is facilitated because of the clear division between baselevel and metalevel.

The evolution of COO systems is also enhanced by separating the application software from the control of the access to objects by several cooperations. The concurrency control is implemented by managers associated with objects and cooperations, which are located at the metalevel.

The reflective architecture supports evolution since it provides the separation of functional and interaction aspects, the hiding of the complexity of a changeable application, and the separation between concurrency control and application, which makes it possible to change both application and concurrency control independently.

4 COO Design and Reflective Architecture of a Case Study

To describe the reflective implementation of a COO design we use the electronic height control system (EHCS), which aims to increase the driving comfort by adjusting the chassis level to different road conditions [26]. It controls the height of a vehicle by regulating the individual height of the wheels through pneumatic suspensions. Figure 2 illustrates the physical system which is composed of four wheels, each of which has a valve and a height sensor, connected to a compressor and an escape valve. For each wheel of the vehicle, the height of the suspension is increased by opening the wheel valve, closing the escape valve and pumping in the air, and decreased by opening the escape and wheel valves, and blowing off the air.

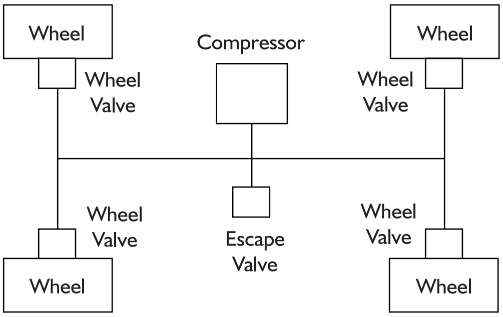


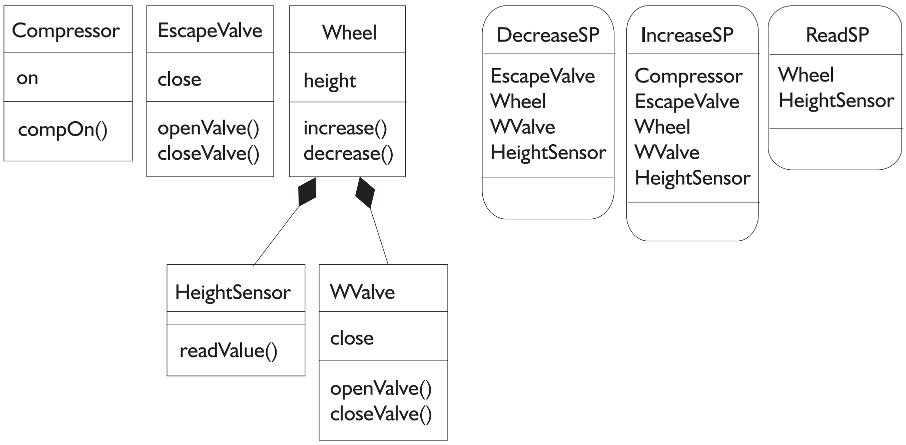
Fig. 2. Electronic height control system.

4.1 COO Design of the EHCS

The COO design of the EHCS is achieved by mapping each physical component into a class and describing interactions between such classes using CO actions. As figure 3 illustrates, the COO design of the EHCS is composed of classes: `Compressor`, `EscapeValve`, `Wheel`, `WValve` and `HeightSensor`; and CO actions: `DecreaseSP`, `IncreaseSP`, and `ReadSP`. In the diagram, CO actions list the participants of their collaborative activity.

Classes. Class `Compressor` provides a method which starts and stops the compressor pumping air and a variable which holds the state of the compressor. Class `EscapeValve` provides methods for opening and closing the escape valve, and a variable holding the state of the valve. Class `Wheel` provides methods for increasing and decreasing the set point of the height of the wheel, and a variable which holds the height of a wheel. Class `Wheel` is composed of two other classes: class `HeightSensor` which can read the height of a wheel; and class `WValve` which provides methods to open and close the valve of the wheel, and a variable holding the state of the valve.

Since in a COO design a class does not incorporate interactions, it can only be described in terms of its own methods. In the COO design of the EHCS class `Compressor` is not aware of the other classes such as, for example, `EscapeValve` and `Wheel`. However, class `Wheel` is aware of classes `HeightSensor` and `WValve` since it is composed of those two classes. Nevertheless, the interactions between such classes are also described by means of CO actions.



(i) class diagram (ii) CO action diagram

Fig. 3. COO design of the EHCS.

CO Actions. CO actions are defined by their four methods: `pre()`, `inv()` and `post()` to check pre-conditions, invariants and post-conditions, respectively, and `coll()` to execute the collaborative activity involving a set of objects. Each CO action is implemented into a metaobject to keep it clearly separated from application objects and other CO actions.

CO actions `IncreaseSP` and `DecreaseSP` respectively describe how to attain an increase and a decrease in the height of a suspension. The former CO action activates methods of objects `:Compressor` and `:WValve`; and checks conditions on other objects to start, hold, and properly end the collaborative activity. CO action `ReadSP` describes how to read the height of a suspension. The value of the height is provided by method `readValue()` of class `HeightSensor` and it is stored in variable `height` of class `Wheel`. The activation of cooperation `:ReadSP` is carried out by cooperation `:IncreaseSP`, thus `:ReadSP` is a nested cooperation. To keep this description simple detail about nested cooperations will be given in the following sub-section.

When describing a CO action a new class is created. Such a class inherits from the abstract class `Cooperation` a method (`collab()`) which is responsible for activating the methods which check preconditions, execute collaborative activity, and check postconditions. Method `collab()` is also responsible for starting a new thread which checks the invariants while the collaborative activity is executed. Moreover, class `Cooperation` provides the primitives which allow the introspection of baselevel objects.

In the following we show an example of CO actions, which is the Java code of CO action `IncreaseSP`.

```
class IncreaseSP extends Cooperation {
    protected int SetPoint = 100;
    protected Object compressor, escapevalve, wheel, wvalve,
       heightsensor;

    // initialization code

    public boolean pre() {
       if (introspect(escapevalve,"close") &&
          introspect(wheel,"height") < SetPoint) return true;
       return false;
    }

    public boolean inv() {
       if (introspect(escapevalve,"close") &&
          ! introspect(wvalve,"close") &&
          introspect(compressor,"on")) return true;
       return false;
    }

    public boolean post() {
       if (introspect(escapevalve,"close") &&
          introspect(wheel,"height") > SetPoint) return true;
       return false;
    }

    public void coll() {
       returnRef1 = invokeMethod(wvalve, "openValve", null);
       returnRef2 = invokeMethod(compressor, "compOn", null);
       returnRef3 = invokeMethod(wvalve, "closeValve, null);
       isOK = readSP.collab(wheel, heightsensor);
    }
}
```

The code has been developed using Dalang, a reflective version of Java [30]. Dalang implements metaobjects as wrappers for objects to allow interception of method calls of the object. Metaobjects exist during run-time, so they can be dynamically created, destroyed, and associated to objects. A meta configuration file is used to determine the association of metaobjects with objects. A run time adaptive version of the EHCS case study was previously presented [11].

The reflective implementation of a COO design does not depend on a particular reflective language. Thus, the code could be produced using other reflective languages that allow interception of method calls and introspection of objects, such as: OpenC++ [6], OpenJava [7], metaXa [15].

4.2 Reflective Architecture of the EHCS

To produce a reflective implementation of a COO design, each object is associated with an *object manager* and each cooperation with a *cooperation manager* [10,28]. However, as explained in the following, nested objects and nested cooperations can generate a different reflective architecture. Object and cooperation managers are used to provide support to the application and their responsibilities include control of access to objects in a concurrent environment.

When some objects in the COO design are parts of an enclosing object, the reflective architecture can be changed in order to optimise the implementation. In this case the manager of the enclosing object can provide its services for the inner objects, and those objects do not need their own managers. It is essential, nevertheless, to assure that the inner objects are relevant to the other objects of the system if only related with its enclosing object.

Figure 4 shows the reflective architecture of the COO design of the EHCS. To simplify the reflective architecture, not every object has been associated with an object manager. Objects `:WValve` and `:HeightSensor` can be considered inner objects of `:Wheel` and so the services provided by object manager `:WManager` for `:Wheel` are also used by those two inner objects.

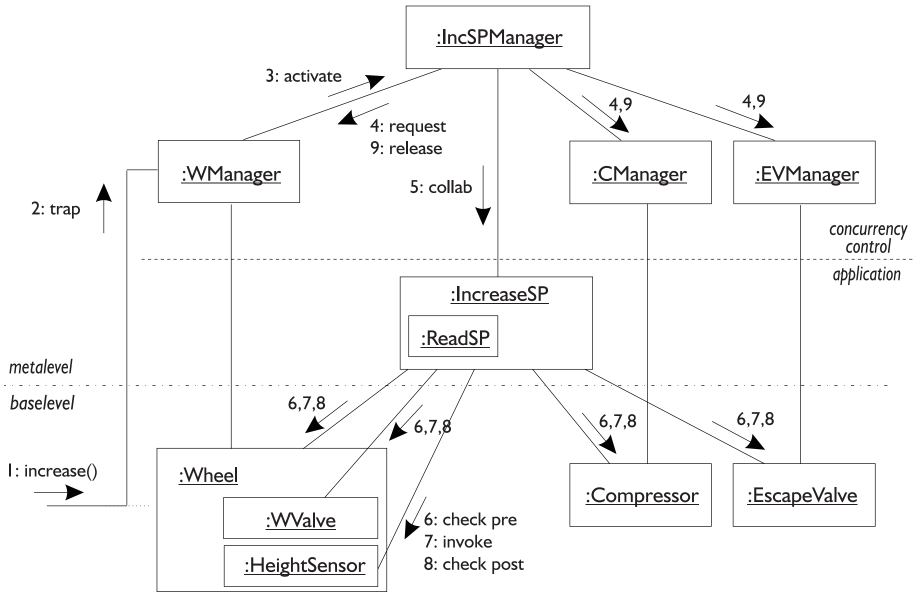


Fig. 4. Reflective architecture of the EHCS.

Object Managers. The role of an object manager is to control the access to its associated object, and to establish the rules for cooperations to invoke the services of that object. Another service that can be provided by an object manager is the invocation of a cooperation manager when control is trapped at the metalevel.

Concurrency control can be achieved by evaluating the state of the object, thus the object manger handles the rules which check the object state and determines whether to grant access to the object. Another way to achieve concurrency control is by enforcing mutual exclusion. To allow synchronization between threads two methods are provided to request an object for 'read' accesses and 'write' accesses, respectively, `readRequest()` and `writeRequest()`. The object manager allows multiple 'read' accesses while denying all 'write' accesses, or allows only one 'write' access while denying all the others. Two other methods are provided to release the object (`readRelease()` and `writeRelease()`).

The result of a request depends on the state of the object associated with the object manager. The object manager will return a failure indication if the associated object cannot be used. Hence an object manager implements a *balking* policy; however, other policies such as *guarded suspension*, *rollback/recovery*, etc. can be also implemented [20].

In the EHCS concurrency control has to be enforced since more than one `:Wheel` could ask to use the compressor at the same time. To implement the concurrency control object manager `:CManager` is able to inspect `:Compressor` to check whether the state of object `:Compressor` allows its use. Concurrency control can be enforced also by mutual exclusion, so `:CManager` will grant access only if the object `:Compressor` is not being used.

An object manager is described as a class which implements the interface defined by the generic `ObjectManager`, which consists of methods `readRequest()`, `writeRequest()`, `readRelease()`, `writeRelease()` and `invokeMethod()`. The implementation of object manager `CManager` when concurrency control is enforced by mutual exclusion is shown in the following. The `synchronized` keyword of Java is used to ensure that only one thread executes inside object `:CManager` at a given time. The methods of `CManager` update two variables according to the number of 'read' and 'write' accesses, and determine the availability of object `:Compressor`. When access is granted the reference to object `:Compressor` (i.e. `target_obj`) is passed to the requesting cooperation manager.

```
class CManager implements ObjectManager {

   protected Object target_obj = null;
   protected boolean intialization = true;

   protected int readers = 0;
   protected int writers = 0;

   public synchronized Object readRequest() {
      if (writers == 0) {
         readers++;
```

```
            return target_obj;
         }
         return null;
      }

      public synchronized Object writeRequest() {
         if (readers == 0 && writers == 0) {
            writers++;
            return target_obj;
         }
         return null;
      }

      public synchronized void readRelease() {
         if (readers > 0) readers--;
      }

      public synchronized void writeRelease() {
         if (writers > 0) writers--;
      }

      public void invokeMethod(Object target,
         String methodname, Object arg[], String metaParam) {
         if (intialization) init_reference(target);
      }
   }
```

Cooperation Managers. The services provided by a cooperation manager are the coordination of the requests for accessing a group of objects participating in a cooperation and the activation of the associated cooperation. A cooperation manager is also responsible for selecting alternative cooperations in case of failure when requesting objects, and for starting recovery actions if the executed cooperation cannot meet the post-conditions. Cooperation managers hold references to the object managers of the objects involved in a cooperation.

A cooperation manager acquires the rights to access objects participating in a cooperation by asking the respective object managers, then it gives the control to one of the associated cooperations. While enforcing control of access to objects, deadlock must be prevented or avoided, thus a cooperation manager implements a prevention or avoidance strategy.

In the EHCS cooperation manager `IncSPManager` is responsible for asking object managers to access objects `:Compressor`, `:EscapeValve` and `:Wheel`. When the access is granted by the object managers, `IncSPManager` will give control to cooperation `:IncreaseSP`. The following shows the Java code of `IncSPManager`. The interface of a cooperation manager, consisting of methods `init()` and `activate()`, is defined by the generic `CooperationManager`.

```
class IncSPManager implements CooperationManager {

   protected static CManager cmanager;
   protected static EVManager evmanager;

   // initialization code

   public void activate() {
      Object target0, target1, target2;
      target0 = wmanager.writeRequest();
      target1 = cmanager.writeRequest();
      target2 = evmanager.readRequest();
      if (target0 != null && target1 != null && target2 != null) {
         boolean isOK=increaseSP.collab(target0,target1,target2);
      }
      wmanager.writeRelease();
      cmanager.writeRelease();
      evmanager.readRelease();
   }
}
```

Nested Cooperations. Cooperations can be organised in a hierarchy, hence there is a top level cooperation which controls cooperations nested within it. Nested cooperations are lower level cooperations. Similarly to Moss's model of nested transaction, a nested cooperation is a tree of cooperations where the subtrees are either nested or flat cooperations [16]. Top level cooperations are able to organise the flow of control, determine when to invoke which cooperation, as well as carry out actual work, whereas leaf level cooperations only perform actual work. With respect to the concurrency control, all the objects held by a parent cooperation can be made accessible to its sub-cooperations.

In the proposed reflective architecture cooperation managers are associated only with top level cooperations. A cooperation manager is responsible for acquiring all the objects used by a cooperation tree. The accessing rights to the objects are then passed from the cooperation manager to the top level cooperation and from the latter to lower level cooperations.

In the COO design of the EHCS cooperation `:ReadSP`, which describes how to read the height of a suspension, is a nested cooperation of `:IncreaseSP`. In the reflective architecture cooperation manager `IncSPManager` is associated with cooperation `:IncreaseSP`, whereas no cooperation manager is associated with cooperation `:ReadSP`.

The collaborative activity of cooperation `:IncreaseSP` describes, among other things, the invocation of nested cooperation `:ReadSP`. The latter cooperation defines the interactions between objects `:Wheel` and `:HeightSensor`. The accessing rights to such objects are passed from `:IncreaseSP` to `:ReadSP`.

Flow of Invocations. The following describes the sequence of invocations for activating `:IncreaseSP` in the reflective architecture illustrated in figure 4. When `increase()` of `:Wheel` is invoked (1), the call is intercepted (2) by object manager `:WManager` which activates cooperation manager `:IncSPManager` (3). The latter requests access to the objects involved in the cooperation (4). When access has been granted by all the object managers, control is given to cooperation `:IncreaseSP` (5). At the application level, cooperation `:IncreaseSP` checks whether the pre-conditions are satisfied (6). Once the check returns true, the collaborative activity starts (7), and the invariants are checked. As part of the collaborative activity of cooperation `:IncreaseSP`, cooperation `:ReadSP` is activated to update the value of variable `height` of `Wheel`. When cooperation `:ReadSP` finishes the control goes back to cooperation `:IncreaseSP` where the post-conditions are checked (8). Finally, the control goes back to the cooperation manager which can release the objects (9).

5 Evolution of the EHCS

A COO system evolves by evolving its classes, CO actions, control to access objects, or adding some classes. The proposed reflective architecture facilitates evolution by providing separation between objects, cooperations and control of access to objects. Each element of the architecture can be easily changed, with little impact on the other elements. Although the proposed reflective architecture better supports evolution of COO systems when only changing their CO actions, it also supports changes of classes and managers. In this section we propose some evolution scenarios that might affect the EHCS and discuss the effectiveness of the proposed reflective architecture in dealing with changes.

The evolution of the reflective implementation of the EHCS will be compared with that of an alternative implementation which makes no use of CO actions nor reflection. A metric is used to measure the evolution effort for each implementation and to evaluate the effectiveness of the proposed reflective implementation.

The *alternative implementation* uses only the class diagram shown in figure 3 (i). However, classes `Compressor`, `EscapeValve` and `Wheel` provide methods that allow their acquisition and releasing (i.e. `readRequest()`, `writeRequest()`, `readRelease()` and `writeRelease()`). Method `increase()` of class `Wheel` describes the interactions between objects and consists of the calls to synchronize threads, the checking of preconditions, the execution of collaborative activity, etc. That is, it describes all the functionalities of `IncSPManager`, `IncreaseSP` and `ReadSP` of the reflective implementation. Analogously, method `decrease()` of class `Wheel` describes the functionalities of `DecSPManager`, `DecreaseSP` and `ReadSP`.

5.1 Evolution of Interactions

Suppose that the EHCS has to evolve by implementing a specialised control algorithm which is able to accommodate gravel as a new type of road. The new control algorithm describes a set of interactions between the existing classes to regulate the height of the suspension (for example, defining a new set point). The change in the reflective implementation for expressing the new requirement is restricted to the definition of a new CO action, `IncreaseSPGravel`. Cooperation manager `IncSPManager` will be changed to select one of two cooperations: `:IncreaseSP` or `:IncreaseSPGravel`, while classes and all the other CO actions and managers remain unchanged. In general changes in CO actions do not involve any changes in the related classes, since the reflective architecture makes classes not aware of the CO actions in which they participate.

Suppose now that the new control algorithm, which regulates the height of the suspension when the road is gravel, has to be introduced into the alternative implementation of the EHCS. The new control algorithm will be defined as a new method of class `Wheel` (i.e. `increaseGravel()`), which implements synchronization and interaction code (similarly to method `increase()`). Modifying class `Wheel` is difficult since it describes several concerns. Moreover, although only the interaction code is new, calls for synchronization and starting of threads have to be rewritten. As a result the code implementing `increaseGravel()` will be much longer than the code for `IncreaseSPGravel` of the reflective implementation and more difficult to write.

5.2 Evolution of Concurrency Control

Suppose that the physical system changes by introducing a new type of compressor which permits air to be pumped to more than one wheel at the same time. A different policy is then adopted to handle the use of the compressor so that more than one access is allowed for object `:Compressor` at the same time. The change in the reflective implementation consists of the definition of a new object manager for object `:Compressor` that will replace the old one. All the classes, CO actions and managers remain unchanged. In general changes in object managers have no impact on the other elements of a COO system. In fact, object managers interact only with cooperation managers and objects. Since cooperation managers are not involved with the rules of accessing objects, and objects are not aware of object managers changing object managers do not impact other elements.

When the new synchronization code is introduced in the alternative implementation of the EHCS, class `Compressor` needs to be updated by changing the methods which allow synchronization. Evolving the alternative implementation requires more effort than evolving the reflective one, since the rules for accessing the objects are not clearly separated from other functionalities of class `Compressor`.

5.3 Evolution of Classes

The EHCS can evolve by changing its classes to reflect changes in the physical elements where they are mapped. For example, class `Compressor`, which initially provides one method both to start and stop the compressor, could be changed to implement two methods, instead of one, one for each activity. Thus, method `CompOn()` has to be changed and method `CompOff()` is added. In the reflective implementation the only element, other than `Compressor`, which needs to be updated is CO action `IncreaseSP`.

Another example is when the physical system changes to accommodate an escape valve for each wheel, and the software needs to be restructured. In this scenario class `EscapeValve` can be considered an inner class of `Wheel` (the same as class `WValve`), so object manager `EVManager` is not necessary anymore, instead `WManager` will be used. Cooperation manager `IncSPManager` is changed to refer to object manager `WManager`, class `Wheel` needs to be changed to incorporate `EscapeValve`, and CO action `IncreaseSP` is changed to refer to the new inner object of `Wheel`. The other classes and their respective object managers remain unchanged.

Modifying classes is the sort of evolution scenario that can start substantial changes in systems based on the COO design. However, from the first example we can observe that when a class changes its interface only the CO actions which use that class have to be modified, whereas object and cooperation managers remain the same. From the second example, we can see that changing the structure of the classes implies changes in the CO actions and their respective cooperation managers, but not in all the classes and their respective object managers.

When method `compOff()` is added to the alternative implementation of the EHCS class `Compressor` is updated. Also method `increase()` of class `Wheel` is updated. Since method `increase()` intertwines fragments of code describing synchronization and interactions, evolving the alternative implementation is much more difficult than evolving the reflective implementation.

When class `EscapeValve` becomes an inner object of class `Wheel` in the alternative implementation, class `Wheel` will be changed accordingly and its method `increase()` will be updated to refer to it. Again method `increase()` has to be modified. Class `EscapeValve` does not need methods for synchronization anymore, so it needs to be modified as well.

5.4 Evolution Metric

An evolution metric is a number that represents the effort necessary to evolve a fragment of code. It quantifies the effort to understand the code, to modify it and to write a new fragment of code. The effort to understand and modify a code is related to the tangling between different concerns of the code, since tangled code is difficult to understand and to modify. When modifying a tangled code it is necessary to mentally untangle it, therefore the more tangled the code the more difficult it is to evolve.

According to [22] the tangling ratio of a code is calculated by means of the following:

$$Tangling = \frac{\begin{array}{c}\#\ of\ transition\ points\ between\\ different\ concerns\ of\ the\ code\end{array}}{LOC} \tag{1}$$

The transition points are the points in the source code where there is a transition between fragments of code dealing with different concerns. The concerns considered in this paper are the implementation of the functionality, the synchronization of threads and the interactions between objects. For example, referring to cooperation manager `IncSPManager` synchronization code is represented as non-underlined code, whereas the interaction code which consists of activating a cooperation is underlined. Then, a transition point occurs between the non-underlined code and the underlined one and another transition point occurs between the underlined code and the non-underlined one.

The evolution metric cannot be equal to the tangling ratio since a code implementing only one concern has a tangling ratio equal to zero. Instead the effort to evolve such a code is greater than zero. The *evolution ratio*, calculated according to the following, considers the effort to understand and modify a code.

$$Evolution = \frac{\begin{array}{c}\#\ of\ transition\ points\ between\\ different\ concerns\ of\ the\ code + 1\end{array}}{LOC} \tag{2}$$

Table 1 summarises the measurements of transitions and lines of code (LOC) and the evolution ratio of each object of the reflective and alternative implementation of the EHCS.

Table 2 summarises the average evolution ratio for the scenarios of evolution described above for the two implementations of the EHCS.

The evolution ratio of a group of classes is the average evolution ratios of such classes. For example, the third row of table 2 refers to the introduction of method `compOff()`, which in the reflective implementation consists of modifying cooperation `IncreaseSP` and class `Compressor`, and in the alternative implementation consists of modifying class `Wheel` and class `Compressor`. Thus, the figures of the third row of table 2 are the average evolution ratios of the classes involved.

The figures of table 2 suggest that evolving the reflective implementation requires less effort than evolving the alternative one, for all the scenarios that have been considered. However, the evolution metric reflects only partially the effort to evolve a code, since, while it measures the effort to understand and modify a code, it does not quantify the effort to write a new fragment of code. We expected that the figures of the first row were not so close to each other, while the figures of the second row were more close to each other. For example, a change

Table 1. Evolution ratios.

	Reflective EHCS			Alternative EHCS		
	transitions	LOC	evolution	transitions	LOC	evolution
Wheel	0	15	7 %	27	94	30 %
Compressor	0	6	17 %	7	28	29 %
EscapeValve	0	13	8 %	9	35	29 %
HeightSensor	0	5	20 %	0	5	20 %
WValve	0	13	8 %	0	13	8 %
WManager	0	48	2 %			
CManager	0	36	3 %			
EVManager	0	36	3 %			
ReadSP	0	24	4 %			
IncreaseSP	0	29	3 %			
IncSPManager	4	21	24 %			
DecreaseSP	0	27	3 %			
DecSPManager	4	19	26 %			

Table 2. Comparison of evolution ratios.

	Reflective EHCS	Alternative EHCS
Evolution of interactions	24 %	30 %
Evolution of concurrency control	3 %	29 %
Evolution of classes (i)	10 %	29.5%
Evolution of classes (ii)	11 %	29.5%

into the alternative implementation could necessitate writing a fragment of code specifying several concerns, and the same change into the reflective implementation could require the adding of code specifying only one concern (or vice versa), thus the effort just to write new code would have a different impact for the two implementations. We argue that the disagreement in our expectations is due to the lack of the evolution metric to quantify the effort to write a new fragment of code.

6 Related Work

Other approaches have been developed to separate functional and interaction code. The Composition Filters approach extends the conventional object oriented model with message filters [2]. Incoming messages pass through the input filters and outgoing messages pass through the output filters. Filters are used to pass messages to other objects (internal or external) and to translate messages. The Composition Filter model can be used to define Abstract Communication Types which are objects describing interactions among objects. When a message received by an object is accepted by a particular filter, named Meta filter, this is passed to an Abstract Communication Type object to handle it.

The Mediator design pattern is an object that encapsulates how a set of objects interact [14]. Mediator promotes loose coupling by keeping objects from referring to each other explicitly. It is responsible for controlling and coordinating the interactions of a group of objects. Each participant of a collaborative activity knows its mediator object and communicates with the mediator whenever it would have otherwise communicated with other colleagues.

Ducasse et al. propose to extend the object oriented model by using connectors [12]. Connectors specify dependencies between objects, and maintain and enforce all the activities between objects. A connector is a special object describing all the information relating to the interactions of objects, including data conversion, interface adaptation, synchronization and coordination.

Kristensen et al. define the concept of activity to describe collaborations between objects [19]. An activity is a modelling and language mechanism that may be used to capture the interplay between groups of objects throughout at different points in time. When the activity abstraction mechanism is objectified, it depicts the relationships that link interacting objects. To support the mapping from design with activities onto implementation entities, some language constructs have been proposed to be added to the existing object oriented languages. An activity is described by a relation-class, an extension of a standard class able to relate the participants of an activity.

In the Layered Object Model layers encapsulate objects and messages sent to or from an object are intercepted by the layers [3]. When a message is intercepted, the layer evaluates the contents and determines the appropriate action. The layered object model extends the object model providing some layer types (adapter, observer, etc.), and it can be also extended with new layer types.

All the above approaches have similarities with our work: they all provide means of explicitly describing interactions between objects into an explicit entity, however some of them lack a clear separation between objects and their interactions, thus changing one of them impacts the other [2,3,14]. Other approaches extend the object oriented model and so they do not allow implementation using standard object oriented languages [12,19].

7 Conclusions

Building a software system that can evolve to incorporate any requirements is impossible. Often systems can be evolved only to incorporate changes that are predictable during their development. However, it is possible to characterize general principles which result in easily evolvable systems. We argue that those principles include the separation between different concerns and the provision of the appropriate architectural support. The support for evolution is provided by considering objects independent of each other and allocating components which coordinate the interaction between objects. The increased number of components for representing a system in terms of the reflective architecture is effective in separating different concerns and reduces complexity since the various components can be understood and altered independently.

When systems implemented using the reflective architecture are evolved objects do not necessarily need to change, instead cooperations are able to capture new object configurations. Reflection has been used to separate different parts of a system, to provide means to intertwine synchronization, interaction and functional code, and to place at the metalevel the part of the system more likely to change.

A case study has been used to show the reflective architecture and some of its evolution scenarios have been described and compared with an alternative architecture. An evolution metric has also been presented to quantify the effort of evolving a system and to characterize the enhancement of evolution due to the reflective architecture.

Ongoing research concerns the expansion of the services provided by object and cooperation managers in order to allow the handling of role-based cooperative designs and of cooperations in a distributed environment. The consistency and effectiveness of the evolution metric is being tested for other evolution scenarios. Moreover, we foresee the inclusion into the evolution metric of means of measuring the effort to write a new fragment of code.

References

1. Mehmet Aksit. Composition and Separation of Concerns in the Object-Oriented Model. *ACM Computing Surveys*, 28A(4), December 1996.
2. Mehmet Aksit, Ken Wakita, Jan Bosch, Lodewijk Bergmans, and Akinori Yonezawa. Abstracting Object-Interactions Using Composition-Filters. In Rachid Guerraoui, Oscar Nierstrasz, and Michel Riveil, editors, *Proceedings of the Workshop on Object-Based Distributed Programming at the European Conference on Object-Oriented Programming (ECOOP'93)*, volume 791 of *Lecture Notes in Computer Science*, pages 152–184, Berlin, Germany, 1993. Springer-Verlag.
3. Jan Bosch. Superimposition: A Component Adaptation Technique. *Information and Software Technology*, 1999.
4. Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern Oriented Software Architecture: A System of Patterns.* John Wiley & Sons, 1996.
5. Walter Cazzola. Evaluation of Object-Oriented Reflective Models. In *Proceedings of the ECOOP'98 Workshop on Reflective Object-Oriented Programming and Systems*, Brussels, Belgium, July 1998. Extended Abstract also published on ECOOP'98 Workshop Readers, S. Demeyer and J. Bosch editors, LNCS 1543, ISBN 3-540-65460-7 pages 386-387.
6. Shigeru Chiba. A Metaobject Protocol for C++. In *Proceedings of the 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*, volume 30 of *Sigplan Notices*, pages 285–299, Austin, Texas, USA, October 1995. ACM.
7. Shigeru Chiba and Michiaki Tatsubori. Yet Another java.lang.Class. In *Proceedings of the ECOOP'98 Workshop on Reflective Object-Oriented Programming and Systems*, Brussels, Belgium, 1998.

8. Rogerio de Lemos and Alexander Romanovsky. Coordinated Atomic Actions in Modelling Object Cooperation. In *Proceedings of the 1st International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'98)*, pages 152–161, Kyoto, Japan, 1998.
9. Rogerio de Lemos and Alexander Romanovsky. Exception Handling in a Cooperative Object-Oriented Approach. In *Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 1–8, Saint Malo, France, 1999.
10. Rogerio de Lemos and Emiliano Tramontana. A Reflective Architecture for Supporting Evolvable Software. In *Proceedings of the Workshop on Software and Organisation Co-Evolution (SOCE'99)*, Oxford, UK, August 1999.
11. Rogerio de Lemos and Emiliano Tramontana. A Reflective Implementation of Software Architectures for Adaptive Systems. In *Proceedings of the Second Nordic Workshop on Software Architectures (NOSA'99)*, Ronneby, Sweden, 1999.
12. Stéphane Ducasse, Mireille Blay-Fornarino, and Anne-Marie Pinna-Dery. A Reflective Model for First Class Dependencies. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'95)*, pages 265–280, Austin, Texas, October 1995.
13. Jacques Ferber. Computational Reflection in Class Based Object Oriented Languages. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'89)*, volume 24 of *Sigplan Notices*, pages 317–326, New York, NY, 1989.
14. Eric Gamma, Richard Helm, Ralph Johnson, and Richard Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. Reading, MA, 1994.
15. Michael Gölm and Jürgen Kleinöder. Implementing Real-Time Actors with MetaJava. In *Proceedings of the ECOOP'97 Workshop on Reflective Real-time Object-Oriented Programming and Systems*, volume 1357 of *Lecture Notes in Computer Science*, Berlin, Germany, 1997. Springer-Verlag.
16. Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Mateo, California, 1993.
17. Walter L. Hürsh and Cristina V. Lopes. Separation of Concerns. Technical Report NU-CCS-95-03, Northeastern University, 1995.
18. Gregor Kiczales. Towards a New Model of Abstraction in Software Engineering. In Akinori Yonezawa and Brian C. Smith, editors, *Proceedings of the International Workshop on New Models for Software Architecture'92*, pages 1–11, Tokyo, Japan, 1992.
19. Bent B. Kristensen and Daniel C. M. May. Activities: Abstractions for Collective Behaviour. In Pierre Cointe, editor, *Proceedings of the 10th European Conference on Object-Oriented Programming (ECOOP'96)*, volume 1098 of *Lecture Notes in Computer Science*, pages 472–501, Berlin, Germany, 1996. Springer-Verlag.
20. Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison Wesley, Reading, MA, 1997.
21. Rodger Lea, Yasuhiko Yokote, and Jun-Ichiro Itoh. Adaptive Operating System Design Using Reflection. In *Proceedings of the 5th Workshop on Hot Topics on Operating Systems*, 1995.
22. Christina Videira Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, Northeastern University, 1997.

23. Patty Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87)*, volume 22 (12) of *Sigplan Notices*, pages 147–155, Orlando, FA, 1987.
24. Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall International Ltd, 1988.
25. Richard Rashid. Threads of a New System. *Unix Review*, 4(8):37–49, 1986.
26. Thomas Stauner, Olaf Müller, and Max Fuchs. Using HYTECH to Verify an Automative Control System. In O. Maler, editor, *Hybrid and Real-Time Systems*, volume 1201 of *Lecture Notes in Computer Science*, pages 139–153. Springer-Verlag, Berlin, Germany, 1997.
27. Emiliano Tramontana and Rogerio de Lemos. Design and Implementation of Evolvable Software Using Reflection. In *Proceedings of the Workshop on Software Change and Evolution (SCE'99)*, Los Angeles, CA, May 1999.
28. Emiliano Tramontana and Rogerio de Lemos. Reflective Architecture Supporting Evolution: a Case Study. In Walter Cazzola, Robert J. Stroud, and Francesco Tisato, editors, *Proceedings of the OOPSLA Workshop on Object Oriented Reflection and Software Engineering (OORaSE'99)*, pages 33–42, Denver, CO, November 1999.
29. Mike VanHilst and David Notkin. Using Role Components to Implement Collaboration-Based Designs. In *Proceedings of the 11th Annual ACM Conference on Object-Oriented, Programming Systems, Languages and Applications (OOPSLA'96)*, pages 359–369, San Jose, CA, October 1996.
30. Ian Welch and Robert Stroud. Dalang - A Reflective Java Extension. In *Proceedings of the OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, Vancouver, Canada, 1998.