Orthogonal Persistence and AOP: a Balancing Act

Mohammed Al-Mansari, Stefan Hanenberg, Rainer Unland
Institute for Computer Science and Business Information Systems (ICB)
University of Duisburg-Essen, Germany
{ Mohammed.Al-Mansari, Stefan.Hanenberg, Rainer.Unland }@icb.uni-due.de

ABSTRACT
In order to increase the productivity of the application developers, it is desirable to remove the persistence concern from their responsibility. For this purpose, the orthogonal persistence concept was introduced along with three principles: type orthogonality, persistence independence and transitivity. From an aspect-oriented point of view these principles have to be considered from the perspective of obliviousness. There is already a number of aspect-oriented persistence solutions where it is not that clear whether they handle the previous principles really in an oblivious way. In this paper, we discuss to what extent these aspect-oriented solutions really make the developer oblivious of the persistence concern. As a conclusion, we find that these systems in general defeat the orthogonal persistence and consequently, using them distracts developers from concentrating on the application logic. In order to increase the obliviousness of the persistence concern we propose a combination of two new concepts: persisting containers and path expression pointcuts.

Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Language Constructs and Features – *Classes and objects, Dynamic storage management.*

Keywords
Orthogonal persistence, obliviousness, persisting containers, locality of join point properties, path expression pointcuts.

1. INTRODUCTION
The term *orthogonal object persistence*, as defined by Atkinson and Morrison in [5], is about automating object persistence so that the application developer can focus on the application logic without having the persistence concern in mind. Using (implicitly) existing persistence mechanisms increases the developer's productivity. In order to achieve this goal, persistence systems must comply with the principle of orthogonal object persistence. According to [4], the available conventional persistence solutions such as Enterprise JavaBeans (EJB, see [17]) and Java Data Objects (JDO, see [16]) fail to support this principle.

From the perspective of aspect-oriented programming (AOP, [12]), the obliviousness characteristic [14] is synonymous to the term orthogonality. Applying it to the domain of persistence implies that the application code does not have to be prepared in order to introduce persistence.

Until now, the aspect-oriented community has made a significant effort to apply AOP in providing orthogonal object persistence (see e.g. [25, 27, 28]). Accordingly, it is crucial to assess whether these proposals comply with the principle of orthogonal persistence, i.e. whether they do not require to prepare code in order to make objects persistent.

This paper critically discusses the extent to which current AO persistence proposals meet orthogonal persistence. Thereto, we distinguish between different levels of code (depending on the role the code plays for persistence) and also consider conventional solutions such as EJB and JDO. Based on this characterization we see that current AO solutions provide a better localization of the persistence concern, however, they do not comply completely with the orthogonal persistence principle.

This paper goes a step further in providing a better level of obliviousness to the object persistence. In order to achieve this goal, we propose a combination of the two new concepts: *persisting containers* and *path expression pointcuts* [1]. We will show how this proposal solves the problem of breaking the orthogonal persistence principle resulting from current AO solutions. We insist that achieving full obliviousness for the persistence concern is not possible especially for complex systems like [19, 20]. However, the paper discusses how our proposal can be used to address some aspects that are considered in [20].

It should be noted that this paper neither considers all persistence issues nor does it provide a complete persistence framework. Rather, it analyzes the shortcoming of the current AO persistence solutions and gives a first step into the direction of providing an even more oblivious implementation of the persistence concern.

The paper is organized as follows: Section 2 motivates the need for (still) discussing persistence from the aspect-oriented perspective. In Section 3 we describe our proposal: the combination of persisting containers and path expression pointcuts that addresses the problem. Section 4 discusses some related work. In Section 5 we discuss some issues of our proposal and the future work, then we conclude the paper.

2. MOTIVATION
This section first describes the idea of orthogonal persistence. Then, it discusses the impact of current persistence approaches with respect to achieve orthogonal persistence. The problem is summarized at the end of this section.

Workshop ACP4IS '07, March 12-13, 2007 Vancouver, British Columbia, Canada

2.1 Orthogonal Persistence

Orthogonal persistence means that persistence can be achieved without requiring the programmers to address persistence issues on their own. To achieve this goal, persistence systems must comply with three principles (as proposed by Atkinson in [4]):

- *Type Orthogonality*: All objects can be persistent or transient irrespective of their types, sizes or any other property. This ensures that the programmer does not have to specify by hand the persistence support for any type. Such handwork preparation distracts the programmer from focusing on the application logic.
- *Transitivity*: Persisting the whole object (i.e. the object and its directly and indirectly referenced objects) ensures the *persistence by reachability* mechanism [11]. This assures the consistency of the stored data and also the correct interpretation of the objects when data retrieval is required.
- *Persistence Independence*: The source and byte code should not require any changes to persisting objects. The developer is not concerned with writing code for moving objects from and to the datastore. Hence, for the code it does not matter whether it is used in a persistent or in a transient environment.

In other words, orthogonal persistence does not require the developer to do any special preparation within the application code to request or receive the persistence service. From an aspect-oriented programming perspective, this requirement meets with the obliviousness property [14]: By examining the code, one cannot tell that the persistence aspect is applied.

From that perspective, it seems obvious to use aspect-oriented programming techniques in order to achieve orthogonal persistence, since orthogonality is considered to be an essential part of aspect-orientation. In the meantime, some AOP solutions for providing object persistence have been proposed, e.g. [25, 27, 28]. These systems provide somehow complete persistence frameworks that cover many issues of persistence: concurrency control, transaction management, distribution, object-relational mapping and SQL translation in case of relational databases, etc.

We distinguish between the following different kinds of how applications are required to be prepared in order to provide the functionality of persistence:

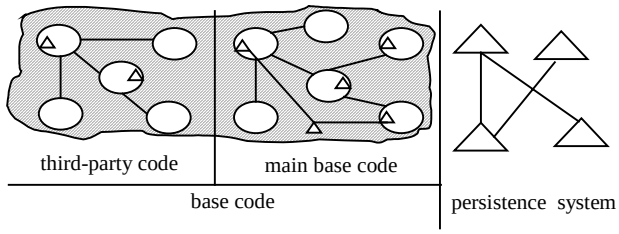
- The type-level preparation for persistence is about introducing persistence-related parts to the types definitions. For example making a persistent type implementing persistence-specific interfaces.
- The code-level preparation is about to include some code that participates in deciding when the objects should be made persistent. This kind of preparation can be noticed statically by examining the base code, for example, when the base code contains an invocation to a method that persists a given object.
- The object-level preparation cannot be figured out statically and it must be determined at run-time. For example, one cannot decide upfront whether an object is reachable from a persistent object.

The following sections discuss these different kinds of preparations in more detail and show that current approaches for achieving persistence do not completely fulfill the requirements of orthogonal persistence.

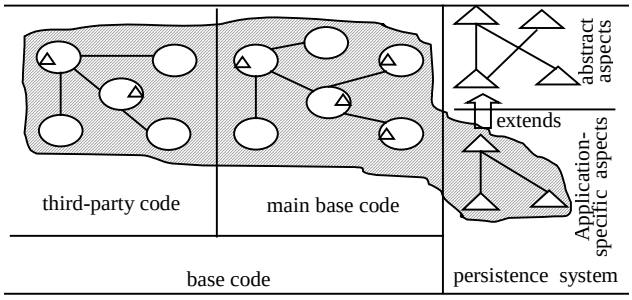
2.2 Preparation on Type-Level

One of the main problems with conventional persistence systems such as EJB is that they require the developers to define within their code the types and classes whose objects are to be made persistent. The programmers have to follow certain rules of the persistence framework. For example, in order to define a persistent type in EJB, the programmers define entity beans that must implement the interface `javax.ejb.EntityBean`. Also two other interfaces should be defined: `javax.ejb.EJBObject` and `javax.ejb.EJBHome`. Moreover, certain naming conventions must be followed. This breaks the type orthogonality principle since only the objects of types that follow these restrictions can be persisted. Also, this breaks the persistence independence principle since the code cannot be used in a non-persistence environment.

In the AO persistence solutions [25, 27, 28], developers have to write their own concrete application-specific aspect to define the persistent types, e.g., to let them extend the `PersistentRoot` class [27]. Moreover, they have to be sure that they can introduce a new superclass to the targets, i.e. it is not valid if the target class already extends a different class due to single inheritance in Java. Then the class that directly extends another class cannot be made persistent, which contradicts with type orthogonality.



Part(a) – type preparation in conventional persistence systems



Part(b) – type preparation in current AOP persistence systems

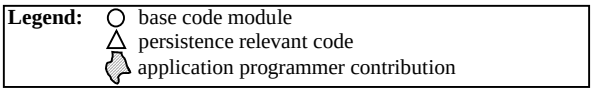


Figure 1. Preparation of persistent objects on type-level.

Moreover, these proposals require the developers to apply certain naming conventions for types and their interface. As an example, in [28] the names of business layer classes must have the postfix `Record`. Also, the setter and getter methods in [27, 28] should start with `set` and `get`, respectively, when defining the interface of a type whose objects are to be made persistent. In [25] the programmers are encouraged to follow such constraints when

defining their classes, otherwise they have to define such naming conventions in the configuration files of the framework which is still a type preparation.

Figure 1 illustrates this situation. In part(a), the developers are concerned with specifying the types to be persistent in conventional systems such as EJB. Developers add the persistence-related code (triangles) to a number of classes (circles) of the base code. This code is either code they wrote on their own or code they imported from a third-party (like a tree package or a given data model). This means that the base code as well as the developer is not oblivious from the persistence system.

Part(b) of the figure shows how application developers have to be aware of the persistence by developing concrete extensions to the aspect-based persistence framework. The definitions of the types in the base code are still not oblivious since these definitions contain persistence-related constraints (the triangles).

Consequently, available AO persistence systems defeat the obliviousness property when modularizing the issue of preparing persistent types. These systems *shift* the problem of identifying the persistent types from the application base code layer to the persistence system layer: Instead of having for each persistent type an explicit declaration within its code, the enumeration of such types becomes part of those aspects that introduce the corresponding type hierarchy. The persistent types interfaces still have persistence-related code. On the one hand, this breaks the type orthogonality since the types that do not follow the persistence-related constraints cannot be made persistent. On the other hand, the developers must be sure that their base code types (including the imported ones) follow these constraints otherwise they have redefine these types or modify them in order to use them in the persistent environments, however, this will break the reusability of those types, hence defeating the persistence independence principle.

2.3 Preparation on Code-Level

In conventional persistence systems such as JDO [16], deciding when an object should be persisted is done statically at the code-level. Developers explicitly invoke methods that persist instances of persistent types, e.g., by invoking the method `makePersistent` of a persistence manager in JDO with the object as an argument.

```
1.Company c1 = new Company();
2.Company c2 = new Company();
3.Address a1 = new Address();
4.Address a2 = new Address();
// ... till this point all objects are transient
5. c1.setAddress(a1);
6. c2.setAddress(a2);
7. pm.makePersistent(c1); // persists:c1 and a1
8. a2.setStreet("NewStreet"); // doesn't persist: a2
// ... c2 and a2 are transient
```

Figure 2. Persisting objects in JDO explicitly on code-level.

Figure 2 shows an example where company objects and address objects are instantiated. The `makePersistent` method persists the company instance `c1`. Its referenced address object `a1` is persistent, as well. Since the company instance `c2` is not explicitly requested to be made persistent, it remains transient. Figure 1(a) (see previous section) still represents this situation because the preparation for persistence spreads over the base code. However, the difference is that the persistence characteristics are not inherently connected to each object of a given type.

Similarly, in the AO persistence systems [25, 27, 28], the base code must include explicit invocations to the setter and the getter methods instead of accessing the fields directly in order to persist the corresponding objects. Also, the developers have to be sure that this constraint is fulfilled inside the types themselves, i.e., any access to a field inside its class should be done using the setter and the getter methods.

For example, in the first code line of Figure 3, the state change to the persistent company object is done by using the set method. This explicit invocation of the setter method will trigger the aspect to persist the changes in the object. However, the second line is changing the address field of the company object but this line of code does not follow the code constraints by the persistence aspects (the required invocation of the setter method for changing an object's state). Hence, the changes will not be persisted.

```
1. c1.setAddress(a1);
// if c1 is a persistent Company object then this state change is persisted
2. c1.address = a1;
// if c1 is a persistent Company object then this state change is not persisted
```

Figure 3. Preparing code for persistence in AO solutions.

This situation is depicted in Figure 4. Here, the application developers are concerned with ensuring persistence in the base code and in the persistence system level. Note that the application developer must prepare the third-party code classes for persistence (in order to fulfill naming conventions, etc.). If these imported classes are in byte-code format, it would be difficult to prepare them. This prevents developers from reusing such types in a persistent and in a transient environment unless the third-party developer is aware of the AO persistence framework being used. However, this is against the persistence independence principle.

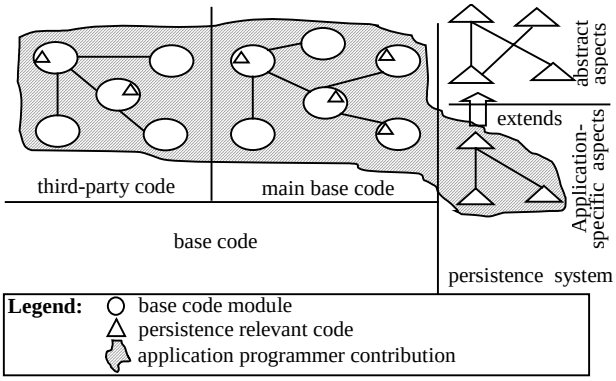


Figure 4. Code-Level preparation in AO persistence solutions.

2.4 Preparation on Object-Level

In Figure 5, the two concrete pointcuts are from the `DatabaseAccess` aspect of the persistence framework in [27]. If the `Company` and `Address` classes are declared to extend the `PersistentRoot` class in the `ApplicationDatabaseAccess` aspect, then the `trapInstantiations` pointcut selects all `Company` and `Address` objects at the time they are created. The associated advice will persist these objects preventing the developers from using, e.g., `c2` as a transient object unless it's declared `transient`.

A developer must thus consider an inverse problem: How to specify individual transient objects?

If the intention of the developer is to use the `Company` object `c2` as a temporary object then (s)he has to define this object as `transient`, defeating the persistence independence principle. Otherwise, if the developer performs a query to know how many `Company` instances the database contains so far then the query returns a wrong number, i.e. 2 instead of 1.

The difference between code level and object-level preparation is that in the former, one can see the preparation statically (e.g. by examining the base code). On the other hand, the object preparation could not be noticed by looking into the base code, however, it is determined dynamically. In Figure 4, the code does not reflect the fact that the four objects are to be made persistent, however, this fact is determined by the persistence aspect. Hence, we consider this preparation as an object-level situation.

```
pointcut traplnstantiations():
 call(PersistentRoot+.new(..));

pointcut trapUpdates(PersistentRoot obj):
 && (this(obj)
 && execution(public void PersistentRoot+.set*(..)))…;
```

```
// The base code
1.Company c1 = new Company();
2.Company c2 = new Company();
3.Address a1 = new Address();
4.Address a2 = new Address();
// ... till this point all objects are persistent
```

Figure 5. Persisting objects in AO systems on object-level.

Assume that companies are persistent objects and its address objects are only persistent by reachability. The `trapUpdates` (cf. Figure 5) pointcut uses the `this` designator to expose the current executing object that must be of type `PersistentRoot`. In Figure 6, an address instance `a` is being changed inside the context of a `PostCodeConverter` object. Since address objects do not extend `PersistentRoot`, the `trapUpdates` pointcut does not select the set join points that update addresses. Hence, the pointcut does not select the change of the postcode of the address object `a`, and the update is not recorded in the database. However, due to the transitivity principle, it is necessary that also changes of the address object lead to an update of its corresponding database representation (as an object owned by a company object).

```
// The base code inside PostCodeCoverter
public void chgPCode(Address a) {
  a.setPostCode("D-45117");  // ... a may be persistent
}
```

Figure 6. Persisting updates in AO systems on object-level.

Nevertheless, current AO systems do not permit to select a join point due to the reachability between objects (which is an information about object relationships). The reason is that this kind of object information is a *non-local join point property* [15, 1]: It cannot be derived from the available local context at this join point. Unfortunately, these systems do not provide constructs for accessing non-local join point properties that are based on object relationships. In such situations, the reachability would not be easy to figure and hence breaking the transitivity principle.

2.5 Problem Statement

Current aspect-oriented persistence systems do not support the obliviousness property. Firstly, the developers of the base code still need to prepare the objects at type-level for persistence that is against the type orthogonality. Secondly, the developers still have to concern with the persistence at the code-level that breaks the persistence independence property. Thirdly, storing and updating persistent objects defeat the persistence independence principle due to the overestimated pointcut specifications. Moreover, updates may also break the transitivity principle since there is no support for non-local object-information join point properties.

3. SOLUTION

This section describes our proposal to solve the problems described above. The first subsection describes the concept of persisting containers along with an example that shows how using these containers fulfills type orthogonality and persistence independence. The second subsection shows how path expression pointcuts can be used along with persisting containers to fulfill the transitivity principle of orthogonal persistence.

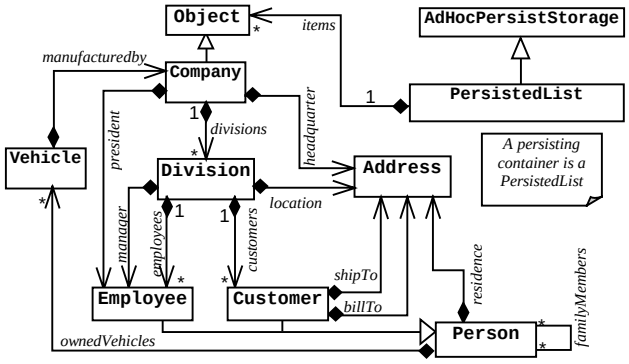


Figure 7. Persisting Containers for a Company object model.

3.1 Persisting Containers

The persisting container is an object of type `PersistedList` that is maintained by the aspect and provides the persistence service to all objects it contains as an ad-hoc functionality similar to the idea of *spontaneous containers* [26]: When an object is added to a persisting container, it is provided with the needed persistence manipulations, when the object is removed from the container, it will not receive this service anymore. All objects of any type have the same right to persist since any object can be added to the containers. Figure 7 shows the design of the persisting containers and an example of how they can be used.

Accordingly, the developers are free to use any object model without concerning with preparing the classes in that model by changing their inheritance structure. In Figure 7, e.g., the `Company` object model is used. Hence, any `Company` object that is added to a persisting container is going to be persisted along with its reference closure. In this way, the base code is completely oblivious with respect to specifying persistent types. Since there is no need to explicitly declare them as persistent types, e.g., by means of the `declare parents` construct. Moreover, the application developer and the third-party developer can remain oblivious to this persistence issue. Figure 8 illustrates this result,

where the base code modules (circles) do not contain any persistence-related parts.

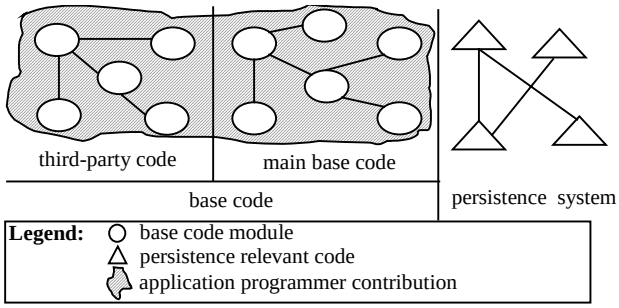


Figure 8. Obliviousness by Persisting containers on type-level.

Consequently, the type orthogonality and the persistence independence principles are met with respect to the preparation of persistent object on the type-level. Firstly, all objects can be persisted irrespective of their types. Secondly, the base code and its developers do not have to be aware of the persistence concern and the same code could be used in either transient or persistent environments, which in turn promotes the reusability of the code.

3.2 Path Expression Pointcuts

Path expression pointcut [1] (path pointcut for short) is a new pointcut construct that applies the well-known path expression technique [7] to AOP in order to provide aspects with access to the non-local object information and to solve the reachability queries between objects in the object graph. The general form is (see section 3.1 in [1] for the path pointcut syntax details):

```
path(PathExpressionPattern);
```

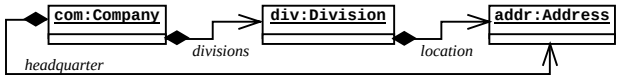
The path pointcut searches the object graphs for the reference paths that match the given path expression pattern. When there is at least one matching path at a given join point, then this join point is selected. The path expression patterns can specify some objects as source, target or intermediate objects of the paths. This can be achieved either by using the exact type patterns, the exact objects names or by using the wildcard patterns. Moreover, the associations between objects can be specified by names or by using the wildcards "`*`" and "`/`". The path pointcut can be used like all other pointcut designators and they can be composed by means of operators "`&&`", "`||`" and "`!`".

Consider the following pointcut:

```
pointcut pc(Company c, Person p, Object o):
    path(c -*-> Employee p -/-> o)
    && set(* *) && target(o);
```

This pointcut picks out every set join point whose target is the object `o` and there is at least one path between the objects `c` and `o` via `p`. The "`*`" in the path expression pattern indicates that there is a direct relationship between `c` and `p`: the object `p` is the value of a field in `c`. However, the name of the field is not relevant for this path specification. The wildcard "`/`" indicates that there may be many objects on the path between these objects: the objects p and o are indirectly related.

The binding mechanism in the path pointcut binds the objects described in the path expression to the corresponding variables in the pointcut's header. The result of a path expression is a set of distinct valid parameter bindings irrespective of the number of the matching paths. This set of parameter bindings is exposed from the join point context to the aspect.

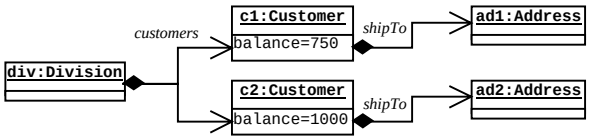



```
pointcut pc(Company c, Address a):
  path(c -/-> a) && target(a) …;
```

Figure 9. Two paths between a company and an address.

For example, in Figure 9, the given path expression matches two paths: `(com –headquarter-> addr)` and `(com –divisions-> div -location-> addr)`, however, the only valid parameter binding is: `(c=com, a=addr)`.

The path pointcut allows multiple occurances of a variable name and it applies a sort of unification so that all these occurrences points to the same object. In Figure 9, the variable name `a`, used as a target in the path expression and then it is used in the `target` pointcut, but the path pointcut binds both to the `addr` object.




```
pointcut pc(Division d, Customer c, Address a):
  path(d -/-> c -/-> a) …;
```

Figure 10. Two paths from Division to Address via Customer.

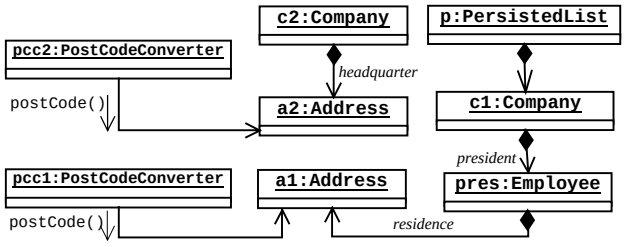
If the path pointcut returns more than one binding, then the associated advice must be executed as many times as the number of the bindings. For example, in Figure 10, the resulting bindings of the pointcut `pc` are: `(d=div, c=c1, a=ad1)` and `(d=div, c=c2, a=ad2)`. Hence, each advice that is associated with this pointcut must be executed two times, each with a single binding.

```
public boolean addrChg(Object[] o1, Object[] o2) {
  Customer cust1 = (Customer) o1[1];
  Customer cust2 = (Customer) o2[1];
  return cust1.getBalance() > cust2.getBalance();
}
pointcut pc(Div d, Customer c, Address a):
  set(* *)&& target(a)
  && path(d -/-> c -/-> a) orderBy(this.addrChg);
```

Figure 11. Possible ordering method specification.

Assume that there are two concurrent transactions each updates one of the objects `ad1` and `ad2` and that the developer wants to run these concurrent changes in a descending order by customers' balances. Since the balance of `c2` is greater, the advice must be executed first on the binding: `(d=div, c=c2, a=ad2)`.

For this purpose, the developer can use the `orderBy` construct with the path pointcut. It takes a name of the method containing the ordering code specified by the developer in a similar way to the `compare` method of the `Comparable` interface in the Java API. For example, in the pointcut `pc` of Figure 11, the parameter of the `orderBy` is the name of the method `addrChg` that has two array parameters of type `Object` each represents a binding. The method extracts the second element from both arrays, casts them to type `Customer` and returns the comparison result between their `balance` fields. When no `orderBy` clause is specified, then the order would be undefined.




```
pointcut trapUpdates(PersistedList pl, Object o):
  set(* *) && target(o) && path(pl -/-> o);
```

The resulting bindings: `(pl=p, o=a1)`.

Figure 12. Obliviousness by path pc and persisting containers.

Figure 12 illustrates how our proposal could be used to support persistence by reachability. Two `PostCodeConverter` objects, `pcc1` and `pcc2`, modify the states of objects `a1` and `a2`, respectively. Object `a1` is part of the company object `c1`, which belongs to the container `p`; hence `c1` and its object closure are made persistent. However, `c2` is not persisted since it is not a part of any container.

The pointcut `trapUpdates` uses the path pointcut and it picks out all set join points where the target object `o` is reachable from a persisting container `pl`. There is only one matching path between a container and an object being changed. This path is resolved to one binding: `(pl=p, o=a1)`, which is exposed to the aspect so that all relevant local and non-local object information at the selected join point can be accessed. As a result, the `trapUpdates` pointcut picks out only the state changes on the object `a1` and not `a2`.

According to this example, persisting containers and path pointcuts can be used in an oblivious way to the base code and the application developer when preparing object for persistence. The base code contains no signs of the availability of the persistence service with respect to any level (type, code and object). Hence, this solution complies with type orthogonality and persistence independence principles without defeating the transitivity.

4. RELATED WORK

In [26], the idea of spontaneous containers was introduced to provide dynamic middleware services to any object, e.g. mobile nodes, from the time this object is added to the container. When the object leaves the container, it won't receive the service anymore. In our proposal we use this idea in order to provide the persistence middleware service to the contained objects.

Path expressions [7] technique has been applied to many other domains, e.g. XPath language [8] that is used to address some parts of the XML [6] documents. In this paper, the intention of applying path expressions to AOP is to get access to different parts of object information in the object graph.

Adaptive programming [22] and strategic programming [21] use the so-called traversal strategies and schemas that are similar to the path expressions. In aspect-based version of these technologies, the advice is triggered whenever the visitor component visits an object from a path that matches the given traversal. This is in contrast to the path pointcut that could be used for selecting join points as well as exposing object paths.

A large research effort is done to provide access to the non-local join point properties. For example, there are approaches for accessing non-local execution trace [3, 9, 10, 30, 31]. The data flow pointcut from [23] provides access to non-local data flow information. Moreover, the `cflow` pointcut in AspectJ [18] provides access to the non-local call stack. However, these proposals neither point to nor solve the problem of using non-local join point properties that are based on object information and object relationships.

The importance of expressive pointcuts in AOP is discussed in [24] where it has been proven that expressive pointcuts increase the modularity and make aspects robust against the changes in the application base code. Also in [29] the authors insist on the need for more expressive join point models that reflect the mental model of the developer. Path expression pointcuts increase the expressiveness of the pointcut language in order to provide better designation over object relationships.

5. DISCUSSION AND CONCLUSION

In general, we believe that achieving a complete obliviousness is not possible, especially when concerning with the rest of persistence issues such as those applying sophisticated transaction management techniques [19]. However, the main goal of the paper is to find the extent to which the current AO persistence solutions support orthogonal persistence.

We find that these systems do not fulfill type orthogonality and persistence independence at the type level. Moreover, they break persistence independence at code level. Also, they defeat persistence independence and transitivity properties at the object level. For solving these problems, we proposed the use of persisting containers and path expression pointcuts.

One important facet of our analysis is that the current AO persistence proposals suffer from the problem of overestimated pointcut specifications that make aspects apply to all objects rather than individual objects. For example, the pointcuts `trapInstantiations` and `trapUpdates` in [27] exhibit this problem: Save "all" instantiated objects of the specified types and persist "all" changes to their persistent states, respectively.

This observation meets the one in the AO challenge case study [20]: How to assign different transaction manipulations to individual transactional objects? As for persistent objects, our proposed concepts of persisting containers and path expression pointcuts could be utilized to modularize some aspects considered in this study. For example, an aspect can maintain two different containers each provides a specific transaction mechanism for its objects. Also, the cascading version-based locking mechanism [11] requires access to all owner transactional objects of the currently updated one to check their versions, which can be solved by the extended path pointcuts [2]. The aspect `Versioned` can use path pointcuts to specify transactional objects in the matching paths from any source to the object being changed. Moreover, the path pointcuts can be used in the `Shared` aspect to designate the transactional-shared object to ensure accessing them in mutual exclusion.

Our proposal is still in its evolution phase, so that we have a lot of concerns that need to be resolved. For example, how to provide a better solution for the problem of specifying persistent objects at the time they are instantiated. This means how and when to add

an object to the persisting containers. Also, it is needed to consider the deletion of persistent objects, i.e., removing them from the persisting containers. Moreover, a special treatment must be identified for the objects of collections. Further discussion is needed on how to use this idea to cover other persistence issues. Other important future work is to discuss how to provide an efficient implementation to the path pointcut in order to minimize the complexity (see [2] for a more detailed discussion).

Being aware of these current limitations, we are still convinced that in order to gain a higher level of orthogonal persistence, it is essential to provide pointcut language constructs that operate on the object level instead of the type level.

6. REFERENCES

1. Al-Mansari, M., Hanenberg, S. *Path Expression Pointcuts: Abstracting over Non-Local Object Relationships in Aspect-Oriented Languages*. NODe'06 Erfurt, Germany; 2006.
2. Al-Mansari, M., Hanenberg, S., Unland, R. *Aspect-Oriented Programming: Selecting and Exposing Object Paths.* In Software Composition (SC07), co-located with ETAPS07; Braga, Portugal; (LNCS - to appear); March 2007.
3. Allan, C., Augustinov, P., Christensen, A. S., Hendren, L., Kuzins, S., Lhot´ak, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J. *Adding Trace Matching with Free Variables to AspectJ*. In OOPSLA 2005: 345-364.
4. Atkinson, M. P. *Persistence and Java - A Balancing Act*. Objects and Databases 2000: 1-31.
5. Atkinson, M. P., Morrison, R. *Orthogonally persistent object systems*. VLDB J. 4 (1995) 319–40.
6. Bray, T., Paoli, J., Sperberg-McQueen. (eds.). *Extensible Markup Language*. http://www.w3.org/TR/REC-XML, 1998.
7. Campbell, R., Habermann, A. *The Specification of Process Synchronization by Path Expressions*. Sym. on Operating Systems 1974: 89-102, V16, Springer Verlag, 1974.
8. Clark, J., Derose, S. (eds.). *XML Path Language (XPath)*. version 1.0. http://www.w3.org/TR/Xpath, 1999.
9. Douence, R., Fradet, P., Südholt, M. *Composition, Reuse and Interaction Analysis of Stateful Aspects*. In Proceedings. of AOSD'04: 141-150, Lancaster, UK, March 2004.
10. Douence, R., Fradet, P., Südholt, M. *Trace-based aspects*. In [13], pages: 201–217.
11. Elmasri, R. and Navathe, S. B. *Fundamentals of Database Systems*. (3rd ed.), Addison-Wesley, 2000.
12. Elrad, T., Aksit, M., Kiczales, G., Lieberherr, K., and Ossher, H. *Discussing Aspects of AOP*. Communications of the ACM 44, 10 (October 2001), 33–38.
13. Filman, R. E., Elrad, T., Clarke, S. and Aksit, M. (eds). *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2005.
14. Filman, R. E. and Friedman, D. *Aspect-Oriented Programming is Quantification and Obliviousness*. In [13], pages: 21-35.
15. Hanenberg, S. *Design Dimensions of Aspect-Oriented Systems*. PhD thesis, Duisburg-Essen University 2005.
16. Jordan, D. and Russell, C. *Java Data Objects*. O'Reilly Media, 1st edition, 2003.
17. JSR-220. *Enterprise JavaBeans v.3: Java Persistence API*. http://java.sun.com/products/ejb/docs.html.
18. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W. G. *Getting Started with AspectJ*. Comm. of the ACM, 2001, 59-65.
19. Kienzle, J., and Guerraoui, R. *AOP - Does It Make Sense? The Case of Concurrency and Failures*. In Proc. of ECOOP'02, Malaga, Spain, 2002, pp.37 – 61.
20. Kienzle, J. and Gélineau, S. *AO challenge -implementing the ACID properties for transactional objects*. In Proc. of AOSD'06, Bonn, Germany, 2006, pp.202 – 213.
21. Lämmel, R., Visser, E., Visser, J. *Strategic Programming Meets Adaptive Programming*. In Proceedings of AOSD'03, pages: 168-177, Boston, USA, March 2003.
22. Lieberherr, K., Lorenz, D. *Coupling Aspect-Oriented and Adaptive Programming*. In [13], pages: 145-164.
23. Masuhara, H., Kawauchi, K. *Dataflow pointcut in aspect-oriented programming*. In 1st Asian Sym. on Prog. Lang. and Sys., LNCS, vol. 2895, pp:105-121, 2003.
24. Ostermann, K., Mezini, M., and Bockisch, C. *Expressive pointcuts for increased modularity*. In Proc. of ECOOP'05, Glasgow, UK, 2005, Springer Verlag, pp. 214 – 240.
25. Pawlak, R., Seinturier, L., Duchien, L., Florin, G., Legond-Aubry, F. and Martelli, L. *JAC: an aspect-based distributed dynamic framework*. Softw., Pract. Exper. 34(12): 1119-1148 (2004).
26. Popovici, A., Alonso, G. and Gross, T. *Spontaneous Container Services*. In Proc. of ECOOP'03, Darmstadt, Germany, 2003, pp: 29-53.
27. Rashid, A. and Chitchyan, R. *Persistence as an Aspect*. In Proc. of AOSD'03, Boston, USA, 2003: 120-129.
28. Soares, S., Laureano, E., and Borba, P. *Implementing distribution and persistence aspects with AspectJ*. In Proc. of OOPSLA, 2002, ACM Press, pp. 174-190.
29. Stein, D., Hanenberg, S., Unland, R.: *Expressing Different Conceptual Models of Join Point Selections in Aspect-Oriented Design*. In Proc. of AOSD'06, ACM Press, pp: 15-26, Bonn, Germany, March 2006.
30. Vanderperren, W., Suvée, D., Cibrán, M. A., De Fraine, B. *Stateful aspects in JAsCo*. In Proceedings of SC 2005, LNCS, pages: 167-181, Edinburgh, Scotland, Apr. 2005.
31. Walker, R., Viggers, K. *Implementing protocols via declarative event patterns*. In ACM SIGSOFT Intel. Sym. on Foundations of Soft. Eng. FSE-12, p. 159–169, 2004.