From Static Methods to Role-Driven Service Invocation – A Metamodel for Active Content in Object Databases

Stefania Leone[1], Moira C. Norrie[1],
Beat Signer[2], and Alexandre de Spindler[1]

[1] Institute for Information Systems, ETH Zurich
CH-8092 Zurich, Switzerland
{leone,norrie,despindler}@inf.ethz.ch
[2] Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussels, Belgium
bsigner@vub.ac.be

Abstract. Existing object databases define the behaviour of an object in terms of methods declared by types. Usually, the type of an object is fixed and therefore changes to its behaviour involves schema evolution. Consequently, dynamic configurations of object behaviour are generally not supported. We define the notion of role-based object behaviour and show how we integrated it into an existing object database extended with a notion of collections to support object classification and role modelling. We present a metamodel that enables specific services to be associated with objects based on collection membership and show how such a model supports flexible runtime configuration of loosely coupled services.

1 Introduction

Object databases typically adopt the type model of object-oriented programming languages such as Java as the data model. Behaviour is usually tightly coupled to an object by defining methods in the object class and every instance of that class will have the same behaviour. The only way of adapting that behaviour is to introduce a subclass with overriding methods.

However, we have seen recent trends in programming and also system design that aim for a looser and more flexible coupling of objects and behaviour. For example, both aspect-oriented programming (AOP) and service-oriented architectures (SOAs) have been used as the basis for supporting context-aware applications by providing context-dependent behaviour [1, 2]. AOP deals with the coupling of objects and behaviour at the programming level and requires recompilation to cope with changes. SOAs offer much more flexibility as the binding of services can be done at runtime. Our aim is to have that same flexibility *within* a database to allow services to be bound to objects in a role-dependent way and further to be able to change these bindings dynamically.

We present a model that allows active content to be bound to database objects dynamically to support a notion of role-dependent services. The behaviour

of an object is defined through a combination of *intrinsic* and *extrinsic* behaviour with methods in the object class defining the former and services associated with object roles defining the latter. We describe how this concept has been integrated into a system based on the db4o object database[3] extended with a notion of collections to support object classification and role modelling.

We begin in Sect. 2 with a discussion of related work and then provide an overview of our approach along with the associated three levels of application models and the metamodel in Sect. 3. Details of the architecture required to realise the model are presented in Sect. 4 and a description of our implementation is given in Sect. 5. We provide a discussion of the approach in Sect. 6 and concluding remarks are given in Sect. 7.

2 Background

Most object databases, including db4o, provide transparent persistence of programming language object instances. Application developers therefore typically use programming languages such as Java as the data modelling language and there is a one-to-one mapping between application entities and object instances. Essentially, the database schema corresponds to the classes that define the attributes and methods available on object instances. It is well-known that this can lead to certain tensions when it comes to dealing with issues of role modelling due to the fact that the type models of object-oriented programming languages like Java do not support concepts such as multiple instantiation and object evolution. It is therefore difficult to model the fact that application entities may have multiple roles simultaneously and that these roles may change over time. Support for role modelling in object databases was an active area of research in the 1990s and a variety of approaches have been proposed (e.g. [3–6]). For example, the programming language Smalltalk [7] was extended to support role modelling by having coexisting class and role hierarchies [6]. Each class that is situated somewhere within the class hierarchy can be the root of a role hierarchy which solves the problem of copying data and creating a new data object every time an object has to take a new role. Furthermore, an object can have multiple roles at the same time which is something that is not offered by object-oriented programming languages but is sometimes "enforced" in languages with multiple inheritance by introducing some kind of artificial class hierarchies.

More recently, the notion of adaptive behaviour in databases has received a lot of attention. Traditionally, object behaviour is represented by methods defined within a class and tightly bound to an object through its class definition. Every object instance of a specific class therefore shows the same behaviour defined by its class methods and any behaviour inherited from its superclasses. However, there are cases where a developer may want the behaviour of an instance to vary according to context [8] or for that behaviour to evolve over time. It is therefore desirable to have a distinction between fixed class-based behaviour and some role-driven runtime behaviour that can be flexibly adapted over time.

[3] http://www.db4o.com

The adaptation of behaviour in object-oriented programming languages is normally achieved through inheritance and the overriding of methods in a subclass. Sometimes the inheritance mechanism is misused just to get access to some service functionality provided by another class. However, inheritance should only be used if there is a proper is-a relationship between a class and its superclass and not simply for the sake of code reuse. The problem of these artificial class hierarchies is more serious if we consider programming languages that offer multiple inheritance where it becomes tempting to have one true is-a relationship with multiple other inheritance relationships that are only used for behaviour reuse. Even if the overriding of methods provides a mechanism for behaviour adaptation, this form of adaptivity is only available at compile time since the class definition can generally no longer be changed at runtime. In most object-oriented programming languages, it is not possible for an object to evolve and gain or lose certain behaviour over time. Only a few dynamic object-oriented languages such as Smalltalk offer the possibility to alter class definitions at runtime so that objects may evolve. Other dynamically typed approaches for runtime behaviour adaptation include prototype-based programming languages such as Self [9] where the concept of classes does not exist at all and a cloning mechanism is used for object instantiation.

Methods that do not directly describe any object behaviour are often implemented as library functionality. These library services are generally represented as static methods that access object instances only by passing these objects as arguments within method calls. Also, there is no binding between classes of objects and their associated services. It is up to the programmer to make an explicit connection from an object instance to its services as part of the application implementation process.

Another solution for adding behaviour to a class is offered by AOP [10]. Extra behaviour is defined by so-called *advices* which are executed at well-specified locations (*pointcuts*) within class methods defining the default behaviour. Functionality or services shared by various classes of a software system (e.g. some logging functionality) can be managed in a modular way by this separation of cross-cutting concerns offered by AOP. The modelling of different types of cross-cutting concerns at various levels of concerns is addressed in aspect-oriented modelling (AOM). Note that the introduction of new behaviour in an aspect-oriented program requires the recompilation and reloading of classes.

Web Services [11] and SOAs [12] enable the composition of services and components in distributed computing. While these solutions offer a language independent reuse of business services, their use often requires significant effort from a developer. A service-oriented DBMS (SDBMS) architecture based on the layered architecture presented in [13] is introduced in [14]. The SOA offer some advantages over monolithic architectures in terms of flexibility. However, in this case, it is important to note that the SOA is used for building and adapting a DBMS by coupling different services rather than for developing an application.

Our aim was to get the same flexibility of service-orientation in terms of dynamically coupling services to objects *within* the database in order to be able

to support the variable and dynamic aspects of object behaviour as well as maximising the *reuse* of behaviour. Our approach allows domain data objects to be associated with flexible role-driven services.

3 Approach

Our approach extends existing object databases with role modelling functionality to enable role-driven service invocation. We have implemented this in db4o, but note that the approach is general and could be used in other object databases.

A simple object model with standard object-oriented concepts such as classes and objects has been extended with a new classification model based on collections and multiple instantiation inspired by the semantic, object data model OM [15]. The collections semantically group a set of objects and the role of an object is defined by its collection membership. Specific services can be associated with a collection to dynamically extend the behaviour of its member objects. These services can either be executed manually by some user interaction or triggered automatically by specific system events (e.g. the insertion of an object into a collection). The classification of objects is orthogonal to the class hierarchy offered by the object model and, through multiple classification, an object can participate in multiple roles at the same time. The flexible runtime reclassification of objects provides a powerful mechanism to dynamically assign new services to an object without affecting its class definition.

Our solution for providing role-driven service invocation is based on a three-layered modelling approach including *type*, *classification* and *service* models as shown in Fig. 1. The type model deals with type specification in terms of attributes and methods. The classification model is used for defining semantic groupings of objects based on collections and relationships between objects. The service model specifies the bindings between services and collections. As part of our new application development process, each of these three models has to be defined. Note that by introducing a type model and a classification model, we clearly separate typing and classification as proposed in [5]. The three models are orthogonal to each other resulting in a clear separation of concerns. We describe each of these models in turn.

3.1 Type Model

The type model defines the types of the objects for a given application domain. As known from object-oriented models, a type declares a set of attributes and methods. In the example shown in Fig. 1, we have three different types `document`, `latexDocument` and `author`. The `document` type defines a set of attributes such as `creationTime` and `encoding` as well as a method `getSource()` which returns a document's content. The type `latexDocument` is a specialisation of the `document` type as represented by the subtype relationship. For example, the `latexDocument` type provides some special handling of LaTeX packages and further offers a method `compile()` which compiles a LaTeX source document

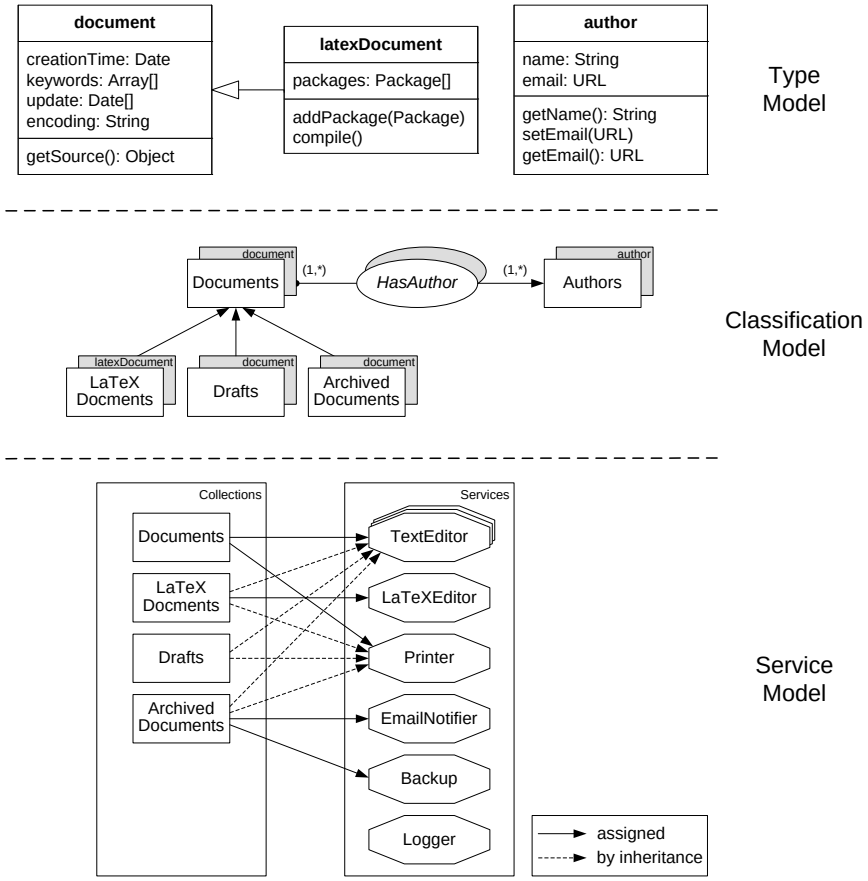

Fig. 1. Type, classification and service model

into an arbitrary output format (e.g. a PDF document). The `author` type defines typical author properties as well as a set of methods to manipulate them. Note that our extended object model supports objects that have multiple of these types through multiple instantiation. An object can gain or lose types at runtime based on specific operators for object evolution.

3.2 Classification Model

For object classification, we introduce the concept of *collections* that have a name and a membertype. In Fig. 1, we use the graphical notation introduced by the OM model [15] where collections are represented by shaded rectangles with the name in the unshaded part and the membertype in the shaded part. An object can be a member of multiple collections at the same time (multiple classification) and be dynamically added to or removed from collections. Furthermore, collections support the notion of a super- and subcollection relationship.

An object in a subcollection will also be in all supercollections and the object is automatically assigned to the corresponding roles.

We also introduce the concept of a *binary collection* with a tuple membertype to represent an *association* from one collection to another. Figure 1 shows a simple example where documents are associated with authors. The `Documents` collection contains objects of type `document` and has the three subcollections `LaTeXDocuments`, `Drafts` and `ArchivedDocuments`. Documents can be associated to authors via the `HasAuthor` association, with each author having authored at least one document and every document having at least one author as indicated by the `(1,*)` cardinality constraints.

The role modelling through classification is represented by the fact that documents can be in the collections `LaTeXDocuments`, `Drafts` and `ArchivedDocuments` simultaneously. Note that there are some collections which do not put further restrictions on the membertype. For example, the `Documents`, `Drafts` and `ArchivedDocuments` collections all have the same `document` membertype. The role of a particular document object can be manipulated by simply adding or removing it from these collections. The fact that a draft of a document may also be archived simply means that the object has to be added to both the `Drafts` and `ArchivedDocuments` collections. However, in some cases, roles may imply additional properties and methods by a more specific subcollection membertype. Through multiple instantiation, objects can therefore gain or lose types and be classified independently of the type hierarchy.

3.3 Service Model

The service model associates services with collections at design- or run time. On the left-hand side of the service model in Fig. 1, we show the set of collections defined in the classification model whereas the right-hand side gives a set of services provided by the system. A service defines arbitrary functionality that can be bound to an object. Services further specify to which type of objects they can be assigned. A service exposes the `Service` interface which contains an `invoke()` method. The binding happens on a collection level where an arbitrary number of services can be assigned to one or multiple collections. These bindings can further be constrained by a given context. Note that the collection membertype must be compatible with the type declared by the service. As a result, a collection defines a context to its members which specifies the set of available services. Furthermore, since all members of a given subcollection are also members of their supercollections, they inherit the service assignments via their supercollections memberships.

We distinguish two types of service invocation. A service can be invoked either automatically based on system events (e.g. if an object is updated, added to or removed from a collection) or explicitly by some user interaction. Our example shows both automatic and manual services. The `Backup` service is an automatically invoked service assigned to the `ArchivedDocuments` collection. It reacts to events generated when a document is inserted into the `ArchivedDocuments` collection. In addition, it has a parameter `periodicity` with the value `daily`

which means that the service is invoked once a day for a daily backup of all collection members. There are multiple collections with membertype `document` but only the ones in the `ArchivedDocuments` will be backed up. This shows that it is the collection membership (role) that defines which services are available for a given object rather than its type.

A second automatically invoked service assigned to the `ArchivedDocuments` collection is the `EmailNotifier` service. This service has been configured to react to the removal of an object from the `ArchivedDocuments` collection to automatically send an email to the authors to inform them that the document is no longer archived. Note that to get access to the corresponding authors and their email addresses, the `EmailNotifier` makes use of the `HasAuthor` association in the classification model. Of course, a service can also be bound to multiple collections and therefore the `EmailNotifier` service could be used for various kinds of notifications.

The `TextEditor`, `LaTeXEditor` and `Printer` services are invoked explicitly by some form of user interaction. For an explicit service invocation, the user is normally presented with a dynamically generated graphical user interface from where they can select one of the available services to be executed. In our example, `Documents` are assigned the `TextEditor` and `Print` services. Due to the fact that `LatexDocuments` is a subcollection of `Documents`, the `TextEditor` and `Printer` services are also available to the members of that collection by means of the collection hierarchy. The `Logger` service that is currently not bound to a collection automatically logs information when objects are accessed. Note that there can also be different implementations of a single service which can be exchanged at runtime as indicated for the `TextEditor` service.

It is also possible to compose new services based on existing ones in order to define more complex functionality out of modular service components. For example, the `Backup` service is a composition of a compression service followed by a copy service. For this purpose, each service may have an arbitrary number of services associated in a specific order defining the sequence of execution.

The service layer is extensible in that new services can be added easily. As described later, a service defines the expected type of object to which it can be applied. For example, the `Printer` service is compatible with the `document` type. This means that objects of type `document` or any subtype can be used with that service. The functionality of a service is implemented in its `invoke()` method. The method implementation may contain calls to external applications as in our example where a `Printer` service is used to initiate the print job.

A metamodel of our system with all the necessary concepts for the three models described in this section is shown in Fig. 2. As discussed earlier, a collection contains objects of a specific type which is represented by the `HasMembertype` association between `Collections` and `Types`. In our metamodel, collections and types are also objects which means that the `Collections` and `Types` collections are subcollections of the `Objects` collection. `Collections` can be associated with `Services` over the `HasServices` association which can be further constrained by contextual conditions (`Contexts`) defined via the `InContext` association.

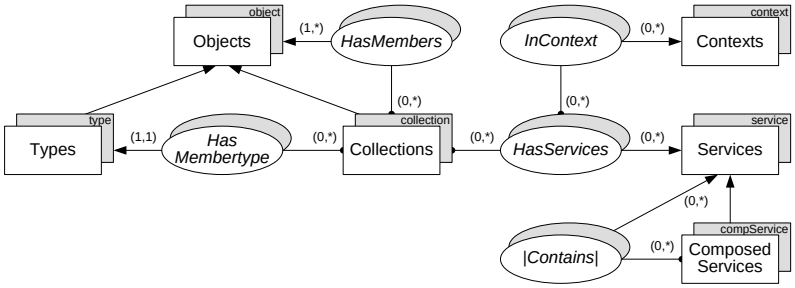

Fig. 2. Role-based service metamodel

A context instance defines a condition that can be evaluated based on information available in the metamodel as well as any external contextual information and returns true if the condition is satisfied. The service is only executed if all associated contextual conditions are satisfied. New services can be composed from existing services based on the `Contains` association and are handled by the `ComposedServices` collection. Note that the `Contains` association is a ranking which means that there is an order defined on the subservice relationship defining the order of precedence when executing multiple cascaded services.

4 Architecture

Our system architecture shown in Fig. 3 combines standard data management components, depicted on the left-hand side of the DBMS, with service components on the right-hand side. The system offers a uniform API that allows an application developer to make use of the functionality presented in the previous section through the database and service API. The data management component implements the typing and classification models and makes them available through the database API, while the service management allows services to be registered and service bindings to be managed based on the service API.

The service manager handles everything that has to do with services including the service library where all available services are registered. Services can be registered and unregistered at design time as well as at runtime. The service manager also manages the service bindings. When assigning a service to a collection, an entry is created in the binding registry which maintains all bindings of services to collections. Note that a service can be assigned to multiple collections and a collection can have multiple services assigned. In summary, the service manager implements the service API offered to the application developer and basically exposes the service model functionality.

As already mentioned, services can either implement functionality themselves, or act as a bridge to third-party functionality and applications. The fact that they can access external functionality is illustrated by the three clouds in the system architecture representing a printer, LaTeX editor and text editor.

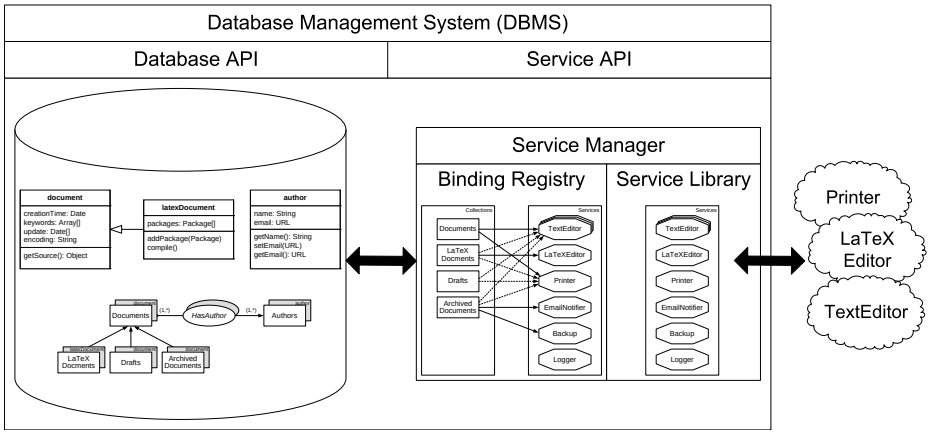

Fig. 3. System architecture

The `Printer` service, for example, accesses printing functionality provided outside the database. In contrast, a `Logger` service would implement the logging functionality within the database.

The service manager is a runtime component that handles service invocation. An object can invoke a service only in the context of a collection which defines the role of that object. Based on the collection, the service manager determines which services can be invoked by performing a lookup in the binding registry. In the case of manual service invocation, the service manager returns the set of available services. Note that since there is no fixed set of services and the number of assigned services may change at runtime, the interface for selecting a service has to be created dynamically. For example, for an object in `LaTeXDocuments`, the service manager returns the `TextEditor`, `LaTeXEditor` and `Printer` services and the user then has to explicitly select the service to be invoked.

Automatic service invocation is handled in two different ways. In the case of periodic invocation, the service manager invokes the service based on the defined periodicity. In the case of event-based invocation, the service manager is notified upon an event such as the insertion of an object into a collection. The notification contains the event type, the object that triggered the event and its role and the service manager then invokes the corresponding service.

5 Implementation

The extended object database has been implemented in Java using the db4o object database for persistent object storage and retrieval. db4o offers the same object model as the programming platform it is embedded in, which in our case is the Java object model. We therefore implemented an additional software layer to run on top of db4o that enriches the Java object model with our additional concepts for role-based service invocation. We first describe the implementation

of the collection and association concepts before presenting our new object implementation for multiple instantiation. The complete set of classes forming the database API is shown in Fig. 4. We will not discuss the `DatabaseManager` and `Database` classes since they offer the same functionality already provided by the underlying db4o object database.

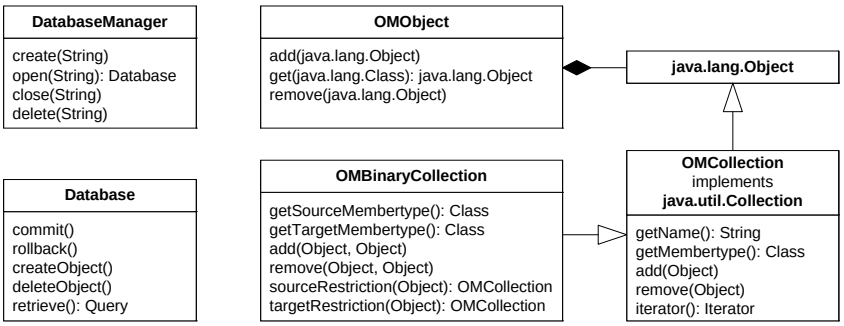

Fig. 4. Database API

The `OMCollection` class implements the Java collection interface and therefore can be used in the same way as regular Java collections. The main difference lies in the intrinsic behaviour of automatically storing and deleting all members to and from the database as soon as they are added and removed from a collection. Associations are implemented as a binary collection class (`OMBinaryCollection`), a subclass of the `OMCollection` class with a tuple membertype containing the types of the two associated objects.

Since members of a collection have to conform to the collection membertype, objects must be able to evolve and dynamically gain and lose types. For this purpose, we need a mechanism to add multiple types to an object at runtime independently of the inheritance hierarchy. In contrast to the Java object model where each object is an instance of its class, we introduce our own extended object model implementation. We distinguish between an *object* representing an identifiable entity and the concept of an *instance* serving as a container for attribute values defined by its type. Multiple instantiation can then be achieved by adding multiple instances to a single object. We use regular Java objects to represent instances whereas an additional `OMObject` class is introduced to deal with our new notion of objects. As shown in Fig. 4, the `OMObject` class manages a set of instances and provides methods for adding, removing and retrieving any of its instances at runtime. The `OMObject` class also offers transparent persistency for storing and updating objects automatically along with all their instances. Note that collections and binary collections are also represented as objects with the `OMCollection` or `OMBinaryCollection` Java classes as assigned instances.

We now explain how the service definition and binding mechanisms have been realised. After a new service has been developed, it has to be deployed to

the `ServiceLibrary` class shown in Fig. 5. The implementation of any service must conform to the `Service` interface definition which also forms part of the service API. The `ServiceManager` class provides methods to add and remove services from the service library. It also offers the `bind()` and `unbind()` methods for assigning services to the corresponding collections. In addition to the collection and service to be assigned, the bind method has further optional arguments to specify the event triggering the service and any context classes it depends on. A context is specified by implementing an interface declaring an `evaluate()` method returning a boolean value which indicates whether the service should be invoked or not. The evaluate method has access to any database content as well as the object on which the service has to be invoked. The service manager also contains the `ServiceLibrary` and `BindingRegistry` classes.

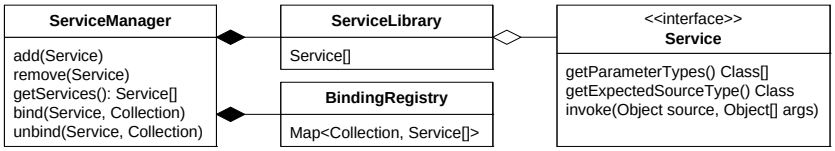

Fig. 5. Service API

To illustrate the usage of our software layer for role-driven service invocation, we show part of the implementation for the application modelled in Sect. 3. Any type represented in the type model is implemented as a regular Java class. For example, the `document` type is defined as follows:

```
class Document {
  Date creation;
  String[] keywords;
  ...
  public Document() {
    this.creation = new Date(System.currentTimeMillis());
  }
  public void addKeyword(String keyword) {
    ...
  }
  ...
}
```

In order to implement the classification domain model, collections and associations have to be created. As stated earlier, collections and binary collections are regular Java classes and can be assigned to `OMObjects` via multiple instantiation. After an object has been created using the database, it is assigned to the collection or binary collection type as shown in the following code excerpt:

```
/* handle to the database 'db' has been assigned previously */
OMObject documents = db.createObject();
documents.add(new OMCollection("Documents", Document.class))
```

Since collections are part of our system's metamodel, they can also be created in a more direct way using the `createCollection()` and `createBinCollection()` database methods. In the following example, the collections `Documents` and `Authors` are created as well as a binary collection for associating

documents and authors. Finally, the `LaTeXDocuments`, `Drafts` and `ArchivedDocuments` collections are defined as subcollections of the `Documents` collection. Note that the membertype of the collection is sent to the creation method in terms of a Java class and, in the case of binary collections, the two membertypes of the tuples have to be provided as shown below:

```
OMObject documents = db.createCollection("Documents", Document.class);
OMObject authors = db.createCollection("Authors", Author.class);
OMObject hasAuthor = db.createBinCollection("HasAuthor", Document.class, Author.class);
OMObject latexDocuments = db.createCollection("LaTeXDocuments", Latex.class);
OMObject drafts = db.createCollection("Drafts", Document.class);
OMObject archivedDocuments = db.createCollection("ArchivedDocuments", Document.class);
latexDocuments.get(OMCollection.class).addSuperCollection(documents);
drafts.get(OMCollection.class).addSuperCollection(documents);
archivedDocuments.get(OMCollection.class).addSuperCollection(documents);
```

Finally we show how the services can be created and assigned to specific collections. In the following example, a service is created as an anonymous class, registered as a service and bound to the `Documents` collection. Note that, in the invoke method, basic query functionality offered by the `Database` and `BinaryCollection` classes has been used to first retrieve the `HasAuthor` association and then access all author objects for the document that has been removed from the `ArchivedDocuments` collection. An email notification is sent to each author of the no longer archived document as indicated in the code fragment:

```
Service emailNotifier = new Service() {
  public Class[] getParameterTypes() {
    /* there are no parameters to this service */
    return new Class[] {};
  }
  public Class getExpectedSourceType() {
    return Document.class;
  }
  public void invoke(Object source, Object[] args) {
    OMBinaryCollection hasAuthor = db.retrieveBinCollection("HasAuthor");
    OMCollection authors = hasAuthor.sourceRestriction((OMObject) source);
    for (OMObject current : authors) {
     URL address = current.get(Author.class).getEmail();
     /* send email to address using Java API */
    }
  }
};
```

The newly created service is finally deployed to the service manager. A context object is created which allows the service only to be invoked if permitted by the general notification policy. The service is bound to the `ArchivedDocuments` collection for invocation on removal events depending on the context evaluation:

```
/* handle to the service manager sm has been assigned previously */
sm.add(emailNotifier);
Context context = new Context() {
 public boolean evaluate() {
  /* return true if notification permitted by general notification policy */
 }
};
sm.bind(emailNotifier, document, archivedDocuments, REMOVAL_EVENT, context);
```

6 Discussion

We have introduced a three-layered modelling approach for dynamic role-based object behaviour in object databases. The implementation of our metamodel covering the concepts of each of these three models resulted in a compact software layer on top of the db4o object database. Standard Java classes are used to represent instances of the type model whereas the classification model is covered by a collection and association framework reflected in an extended database API. Any services specified in the service model are implemented based on a well defined service interface and are bound to objects in a role-dependent manner through the collection interfaces.

The loose coupling and runtime binding of services is addressed by SOAs. While the publishing, registration and configuration of services in SOAs still requires a major effort from a developer, we offer the same flexibility within a database. Most SOAs deal with service invocation on a rather technical level, whereas we offer high-level conceptual constructs for role-based service binding and invocation. In addition to the explicit service invocation offered by SOAs, our approach supports an implicit invocation of services based on the handling of events in combination with an object's role.

In contrast to SOAs where service calls are explicitly reflected in the programming code, our role-based approach enables the configuration of services as extrinsic object behaviour. Instead of dealing with alternative service invocations by cumbersome if-then-else statements, our collection-based object classification enables a highly flexible and dynamic runtime behaviour adaptation by simple reclassification. Note that our active content approach has also been used for the database-driven development of highly interactive systems [16].

The *separation of intrinsic and extrinsic object behaviour* is not addressed in most object-oriented programming languages. While the intrinsic object behaviour is tightly coupled to an object's type, there is often no mechanism for object evolution and the flexible modelling of extrinsic object behaviour. Any form of additional functionality that is non-type-specific is normally implemented via static method calls to external software libraries. However, this implies that a programmer has to deal with if-then-else statements to make use of these library methods in a context-dependent manner. Furthermore, there is no explicitly modelled relationship between this additional functionality and the types of objects to which it should be applied. With our three-layered application development approach, this library service functionality can be bound to objects in a context-sensitive way without affecting an object's type definition and the reusability of object types across different applications.

Just as there is a trend to treat relationships and associations as first-class constructs in modern programming languages in order to enhance the reusability of components [17], we think that our solution leads to a clear separation of concerns in dynamic service binding. This finally results in cleaner component interfaces which are defined by object types and enhances the reusability of types as well as services across different application domains.

7 Conclusion

While the concept of methods in object models is a rather static way of binding behaviour to an object, we have presented an approach that enables a role-based definition of object behaviour. Our three-layered conceptual model provides a clear separation of concerns between the object type specification, the classification and association of objects and the dynamic role-based service invocation on individual objects. By means of a simple example application, we have highlighted how the role-driven service invocation mechanism leads to a cleaner development process by associating objects with role-based behaviour which would otherwise be spread across different static library method calls.

References

1. Dantas, F., Batista, T., Cacho, N.: Towards Aspect-Oriented Programming for Context-Aware Systems: A Comparative Study. In: Proc. of SEPCASE 2007, Minneapolis, USA (May 2007)
2. Gua, T., Punga, H.K., Zhang, D.Q.: A Service-Oriented Middleware for Building Context-Aware Services. Journal of Network and Computer Applications **28** (2005)
3. Pernici, B.: Objects with Roles. In: Proc. of OIS '90. (1990)
4. Albano, A., Bergamini, R., Ghelli, G., Orsini, R.: An Object Data Model with Roles. In: Proc. of VLDB '93, Dublin, Ireland (August 1993)
5. Norrie, M.C.: Distinguishing Typing and Classification in Object Data Models. In: In Information Modelling and Knowledge Bases, volume VI. (1995)
6. Gottlob, G., Schrefl, M., Röck, B.: Extending Object-Oriented Systems with Roles. ACM Transactions on Information Systems **14**(3) (1996)
7. Goldberg, A., Robson, D.: Smalltalk-80: The Language and its Implementation. Addison-Wesley (1983)
8. Grossniklaus, M., Norrie, M.C.: Supporting Different Patterns of Interaction through Context-Aware Data Management. JWE **7**(3) (2008)
9. Ungar, D., Smith, R.B.: SELF: The Power of Simplicity. Lisp and Symbolic Computation **4**(3) (1991)
10. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect-Oriented Programming. In: Proc. of ECOOP '97, Jyväskylä, Finland (June 1997)
11. Papazoglou, M.: Web Services: Principles and Technology. Prentice Hall (2007)
12. Krafzig, D., Banke, K., Slama, D.: Enterprise SOA: Service-Oriented Architecture Best Practices. Prentice Hall (November 2004)
13. Härder, T.: DBMS Architecture – New Challenges Ahead. Datenbank-Spektrum **14** (2005)
14. Subasu, I.E., Ziegler, P., Dittrich, K.R.: Towards Service-Based Database Management Systems. In: Proc. of BTW 2007, Aachen, Germany (March 2007)
15. Norrie, M.C.: An Extended Entity-Relationship Approach to Data Management in Object-Oriented Systems. In: Proc. of ER '93, Arlington, USA (December 1993)
16. Signer, B., Norrie, M.C.: Active Components as a Method for Coupling Data and Services – A Database-Driven Application Development Process. In: Proc. of ICOODB 2009, Zurich, Switzerland (July 2009)
17. Balzer, S., Gross, T.R., Eugster, P.: A Relational Model of Object Collaborations and its Use in Reasoning about Relationships. In: Proc. of ECOOP 2007, Berlin, Germany (July 2007)