3. Crosscutting Concerns in Database Systems

Chapter 2 introduced the problem of crosscutting concerns in software systems. This chapter discusses crosscutting concerns specific to database systems and how their tangling with other concerns adversely affects customisability, extensibility, maintainability and evolvability. Crosscutting concerns in database systems can be categorised as:

- DBMS-level crosscutting concerns
- Database-level crosscutting concerns

The former, discussed in Sect. 3.1, relate to the software system managing the database. Conceptually they are not very different from crosscutting concerns in other software systems apart from some domain specific properties. Sections 3.1.1-3 discuss some concrete examples of such crosscutting concerns namely, the instance adaptation approach, the schema evolution approach and the transaction model. Some other crosscutting concerns at the DBMS level are listed in Sect. 3.1.4.

Crosscutting concerns at the database level, discussed in Sect. 3.2, relate to the data and meta-data (e.g., the schema) residing in the database. They are persistent as they are spread across persistent entities. This persistent nature must be taken into account by modularisation techniques. Sections 3.2.1-3 provide three concrete examples of such crosscutting concerns. These include links among persistent entities, versioning information and the instance adaptation approach. Some other database level crosscutting concerns are listed in Sect. 3.2.4. Section 3.3 concludes the discussion in the chapter.

3.1 Crosscutting Concerns at DBMS Level

Trade-offs between modularity and efficiency, and granularity of services and the number of interservice relationships result in DBMS designs which lack customisability. This section discusses the problems faced when trying to achieve a high degree of customisability at the DBMS level in monolithic, layered and component-based designs. These problems arise due to the crosscutting nature of customisable features.

The structure of a monolithic DBMS is shown in Fig. 3.1a. For simplification only three key components: the storage manager, the transaction manager and the schema manager are shown. As shown in the figure, the various components are tangled together. While such a closely woven implementation provides good performance/efficiency, customisation is an expensive and difficult task as changes to the various components are not localised. Changing the indexing or clustering technique employed by the storage manager, the instance adaptation approach employed by the schema manager or the transaction model used by the transaction manager, for example, at either compile-time or run-time can have a large ripple effect on the whole system. Layered architectures (Haerder and Reuter 1983; Ramakarishnan 1997) only provide partial solutions to the customisation problem and only at a coarse granularity. The intra-layer and inter-layer interactions in such designs simply shift the code tangling problem to a different dimension (Pulvermueller, Speck, et al. 2000). In the layered architecture proposed by (Haerder and Reuter 1983), for example, the concurrency control components are spread across two different layers. Customisation of the lock management or recovery mechanisms (residing in the lower layer) have a knock-on effect on the transaction management component (residing in the higher layer).

The customisation approach employed by most commercial (esp. relational and object-relational) DBMSs is shown in Fig. 3.1b. In this approach the DBMS design is still largely monolithic (to improve performance). Special customisable points similar to *hot spots* in OO frameworks (Fayad and Schmidt 1997) allow custom components to be incorporated into the system. Examples of such components include Informix DataBlades (Informix White Paper, 1998), Oracle Data Cartridges (Banerjee, Krishnamurthy, et al. 2000) and DB2 Relational Extenders (Debloch, Chen, et al. 2000). However, customisation in these systems is limited to introduction of user-defined types, functions, triggers, constraints, indexing mechanisms and predicates, etc. Customisations such as introduction of new index types, for

example, trigger the need to adapt the concurrency control mechanism, particularly the lock manager, for performance reasons. Changes to the lock manager in turn affect logging, recovery, etc. (Dittrich and Geppert 2000).

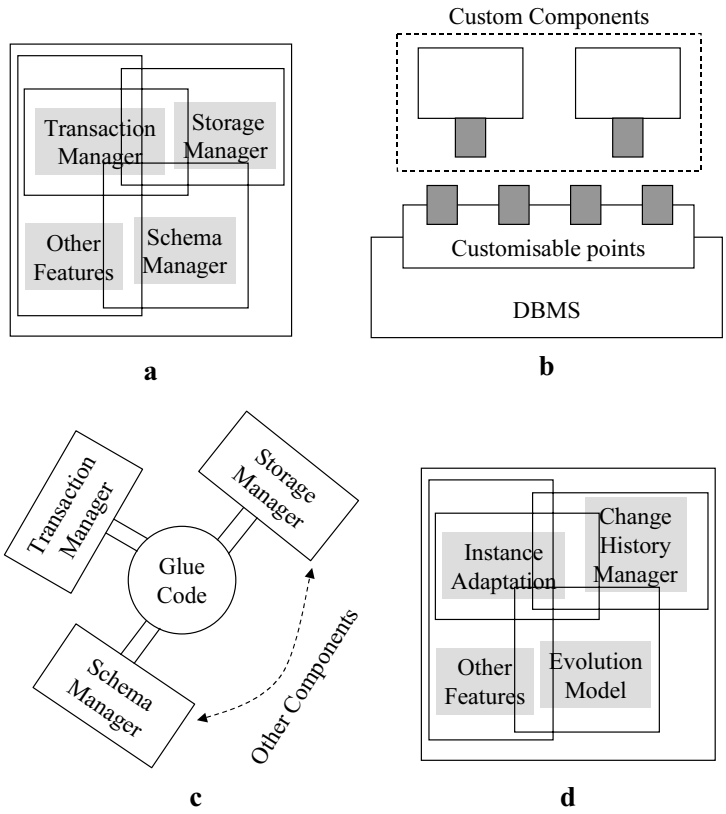

Fig. 3.1a-d. Crosscutting in existing DBMSs: (**a**) a monolithic DBMS (**b**) a DBMS with plug-in components (**c**) a component-based DBMS (**d**) a component (schema manager) in a component-based DBMS

Fig. 3.1c shows the modular, component-based implementation employed by systems such as KIDS (Geppert and Dittrich 1995; Geppert, Scherrer, et al. 1997), Navajo (Bobzin 2000) and Objectivity (Guzenda 2000). Again, only three key components: the storage manager, the transaction manager and the schema manager are shown for simplification. Such a design provides effective customisability. For example, the transaction manager (or any other component) can be exchanged with the ripple effect

mainly limited to the glue code. However, in order to strike the right balance between modularity and efficiency the design of the individual components is not highly modular. As shown in Fig. 3.1d, the modularity of the individual components is compromised to preserve both modularity and efficiency of the DBMS. As a result the coarse-grained component, the DBMS, is customisable. However, customisability at a finer granularity (i.e., the components forming the DBMS) is expensive as design at this level is largely monolithic – the customisation problem simply moves to a different granularity. Such customisation would be cost-effective if changes at the fine granularity were localised without compromising the system performance obtained through closely woven components, i.e., both modularity and efficiency need to be preserved. Even if componentisation is applied at a fine granularity cross-component constraints introduce tangling among these fine-grained components.

Note that database systems based on component architectures such as OLE DB (Blakeley 1996; Microsoft 1999) and CORBA (OMG 2003) also offer customisation support at a coarse granularity only and hence suffer from the same shortcomings as component-based DBMSs discussed above.

Next, we look at some concrete examples of crosscutting concerns at the DBMS level.

3.1.1 Instance Adaptation Approach

Instance adaptation is the process of simulated or physical conversion of objects across compatible class definitions upon schema evolution in object databases. Note that instance adaptation is a concern at both the DBMS level and the database level (cf. Sect. 3.2.3). It is, therefore, important to clarify the distinction between the *instance adaptation approach* and *an instance adaptation routine*. The former is a DBMS level concern and is the code that forms part of the schema manager. It defines the instance adaptation strategy for the system, for instance, simulated conversion or physical conversion. An instance adaptation routine, on the other, hand is the code specific to a class or its historical representation (depending on whether change histories are maintained). It is, therefore, a database level concern and defines adaptation semantics for instances of the particular class definition. The instance adaptation approach specifies the type (and at times structure) of the instance adaptation routines to be used, e.g., error handlers (Skarra and Zdonik 1986), update/backdate methods (Monk and

Sommerville 1993) or transformation functions (Ferrandina, Meyer, et al. 1995) and, upon detection of an interface mismatch between a class definition and the object accessed using it, invokes the appropriate routine with the correct set of parameters.

Organisations have highly specialised evolution requirements with reference to instance adaptation (Rashid 2000a; Rashid, Sawyer, et al. 2000). For one organisation it might be sufficient that object conversion is simulated while for another physical conversion may be essential. The choice is dictated by "local" organisational needs and at times custom variations of standard instance adaptation models can be required. The choice of instance adaptation approach may even vary from one application to another within the same organisation.

Traditionally, the instance adaptation approach in object database systems is tangled with other parts of the schema manager, for instance, the evolution model and change history manager (cf. Fig. 3.1d). Customising the instance adaptation approach in such a system, for example, requires exchanging the whole schema manager component as changes to the instance adaptation approach are not localised. This also limits possibilities for dynamic customisation. Application specific customisation of the instance adaptation approach, for example, requires dynamically exchanging the whole schema manager component which is an expensive task. Furthermore, a change to the instance adaptation approach can have a large ripple effect on the instance adaptation routines at the database level (cf. Sect. 3.2.3).

3.1.2 Schema Evolution Model

(Sjoberg 1993) provides measurements of the frequency and extent of changes to the conceptual structure of a database. The need for these variations (evolution) arises due to a variety of reasons, e.g.:

- to correct mistakes in the database design;
- to add new features during incremental design;
- to reflect changes in the structure of the real world artefacts modelled in the database.

Let us consider evolution models in object databases as an example. These models can be divided into four categories:

- *Schema modification* (Banerjee, Chou, et al. 1987; Banerjee, Kim, et al. 1987; Fishman 1987; Li and McLeod 1994; Ferrandina, Meyer, et al. 1995; Peters and Ozsu 1997; Dmitriev 1998), where the database has one logical schema to which all changes are applied. No historical representations of class definitions are kept.
- *Schema versioning* (Kim and Chou 1988; Lerner and Habermann 1990; Odberg 1992; Ra and Rundensteiner 1997), which allows several versions of one logical schema to be created and manipulated independently. Change histories are maintained at a coarse granularity.
- *Class versioning* (Skarra and Zdonik 1986; Bjornerstedt and Hulten 1989; Monk and Sommerville 1993; Rashid, Sawyer, et al. 2000), which keeps different versions of each type and binds instances to a specific version of the type. Change histories are maintained at a fine granularity.
- Hybrid models such as *context versioning* (Andany, Leonard, et al. 1991), which is based on versioning partial, subjective views of the schema or those based on superimposing one model on another, e.g., (Benatallah and Tari 1998; Rashid 2001a).

Traditionally, an object database management system offers the maintainer one particular schema evolution model coupled with a specific instance adaptation mechanism. For instance, CLOSQL (Monk and Sommerville 1993) is a *class versioning* system employing *dynamic instance conversion* as the instance adaptation mechanism; ORION (Banerjee, Chou, et al. 1987) employs *schema modification* and *transformation functions*; ENCORE (Skarra and Zdonik 1986) uses *class versioning* and *error handlers* to simulate instance conversion. It was argued in Sect. 3.1.1 that such "fixed" instance adaptation approaches do not serve local organisational needs effectively. The same is true for the evolution model. For one organisation (or application) it might be inefficient to keep track of change histories, hence making *schema modification* the ideal evolution model. For another organisation (or application) maintenance of change histories and their granularity might be critical. In some cases, custom variations of existing approaches might be desirable.

As shown in Fig. 3.1d, the evolution model tends to be tangled with other parts of the schema manager, e.g., the instance adaptation approach and the change history manager. Consequently, any change to the evolution model (whether carried out offline or dynamically) can have a large ripple effect on the schema manager. Furthermore, customisation of the evolution model is made virtually impossible by the fact that most database management sys-

tems do not distinguish between the schema evolution model and the schema implementation model. With the exception of the TSE system (Ra and Rundensteiner 1997), the schema evolution model is also used as the implementation model. This implies that customisation of the evolution model would invalidate the whole schema and, in fact, the whole database.

3.1.3 Transaction Model

Transactions transfer the database from one consistent state to another. A number of transaction models have been proposed. The classic flat transaction model enforces the ACID (atomicity, consistency, isolation and durability) (Gray and Reuter 1993; Elmasri and Navathe 2000) properties of transactions operating on the database. Each transaction is primitive with no sub-transactions. A transaction in the nested transaction model (Moss 1982; Moss 1987), in contrast, is a tree structure with a root or top-level transaction. Nested within the root are one or more sub-transactions (also known as child transactions). Nested transactions are useful when a transaction is required to hide intermediate results. This allows the usual rollback of persistent data to the state at the beginning of the transaction. However, nesting of transactions allows rollback of persistent data to some point after the beginning of the top-level transaction. Top-level transactions satisfy the ACID properties with respect to other top-level transactions while child transactions preserve atomicity, consistency, isolation and durability with respect to each other. Other transaction models, e.g., cooperative transactions (Zhang, Kambayashi, et al. 1999) relax the basic ACID properties to allow a transaction to observe the internal state of other transactions.

For traditional data-oriented, record processing database applications, the basic flat transaction model or the nested transaction model tends to be satisfactory. Advanced applications such as collaborative design or development environments, on the other hand, are better served by a cooperative transaction model (often one supporting long transactions (Katz 1990; Santoyridis, Carnduff, et al. 1997)) permitting visibility of changes made by a transaction before it commits. There can be other application specific properties desirable of a transaction model, for instance, whether a new transaction spawned from an existing one is started as a new thread or as part of an existing one (Dittrich, Gatziu, et al. 1995). Similarly, it might be desirable that a transaction may commit only if another transaction has committed (or aborted) (Dittrich, Gatziu, et al. 1995).

Traditionally, a DBMS offers one particular transaction model – basic flat or nested transactions or a set of slight variations of such a basic model – to the applications. The supported variations, if any, cannot cater for the needs of all applications. Customisation of the transaction model to suit the needs of a particular application (not satisfied by the basic model or its variations) is difficult and expensive as the transaction model is intertwined with other concerns such as cache management and synchronisation of persistent and transient state.

3.1.4 Other Crosscutting Concerns at DBMS Level

Sections 3.1.1-3 provided some detailed examples of crosscutting concerns at the DBMS level and how these inhibit adaptability and evolution. Other examples of crosscutting concerns at the DBMS level include:

- version management policy and workgroup support features, which are intertwined with the transaction model, the storage manager, the access manager and the evolution model (depending on whether versioning of schema elements is supported);
- security and access control policies, which overlap with the version management and workgroup support policies, the storage and access managers and the evolution model;
- cache management and object synchronisation policies, which have a crosscutting relationship with the transaction model and the recovery mechanism;
- strategy for lazy evaluation and resolution of transient references to persistent objects, a concern intertwined with cache management and the access manager.

3.2 Crosscutting Concerns at Database Level

This section provides examples of concerns that cut across data or metadata residing in the database and lead to tangled, inflexible representations that are costly to evolve. The examples also demonstrate how customisation of features, such as the instance adaptation approach, at the DBMS level has a direct impact on entities already residing in the database. It is shown that existing database systems provide little support for localising the impact (at the database level) of any changes propagated due to customisation at the

DBMS level. As a result propagation of customisation results in an increased maintenance overhead at the database level.

3.2.1 Links Among Persistent Entities

A number of relationships can exist among entities – objects, meta-objects (class and attribute definitions, etc.) and meta-classes – residing in an object database. Examples of these relationships are:

- association and aggregation relationships among objects;
- version derivation links among object or class versions;
- inheritance relationships among classes (or their versions);
- scoping relationships between classes and their members.

Traditionally object databases use attributes to embed relationship information within the entities hence spreading the relationships across them. Fig. 3.2a shows part of the meta-class in a system embedding inheritance links within class meta-objects. Each *class* meta-object maintains collections of references to *class* meta-objects forming its superclasses and subclasses. Such a structure introduces the evolution problem at the meta-object level. When any inheritance relationships are modified corresponding attributes in the affected meta-objects need to be updated to reflect the change. Modifying the structure of meta-classes and introduction/removal of existing classes is very expensive. This reduces the extensibility and maintainability of the system.

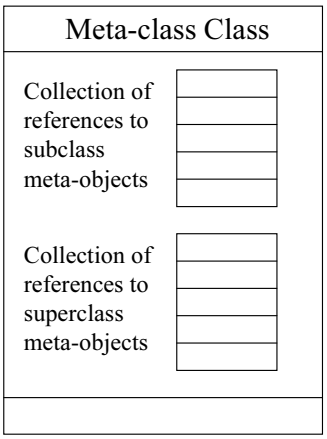

Fig. 3.2a. Meta-class structure showing inheritance relationship implementation in existing systems

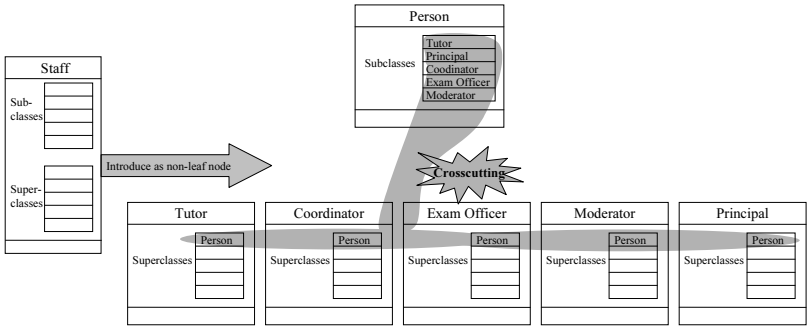

Fig. 3.2b. Modification of inheritance relationships upon introduction of a non-leaf class meta-object in existing systems

Fig. 3.2b shows an example schema evolution scenario depicting the meta-objects in such a system prior to the introduction of a non-leaf class. The scenario is based on a case study carried out at an adult education institution in the UK. The classes *Person*, *Tutor, Coordinator*, *Exam Officer*, *Moderator* and *Principal* already exist in the system while the class *Staff* is the non-leaf class being introduced into the system. Upon introduction of the meta-object for class *Staff*, all the references to subclass meta-objects will have to be removed from the *Person* meta-object and all the subclass meta-objects will have to be updated to remove the reference to *Person* in their respective collections of superclass references. A reference to the *Person* meta-object will have to be added to the superclasses collection and references to older subclasses of *Person* will have to be added to the subclasses collection in the *Staff* meta-object. A reference to the *Staff* meta-object will have to be added to the subclasses collection (not shown in Fig. 3.2b) in *Person* and to the superclasses collection in each of its former subclasses. The number of entities affected upon modification of a connection is m+n where m and n represent the number of participating entities at each edge of the relationship (in this case superclasses edge and subclasses edge). This example demonstrates that information about links among entities cuts across them.

3.2.2 Versioning Information

Some applications require hierarchical organisation of data based on change histories. For instance, versioning of objects (Chou and Kim 1986; Katz, Chang, et al. 1986; Klahold, Schlageter, et al. 1986; Ecklund, Ecklund,

et al. 1987; Ketabchi and Berzins 1987; Dittrich and Lorie 1988; Katz 1990; Oussalah and Urtado 1997; Santoyridis, Carnduff, et al. 1997) is often supported to service collaborative computer-aided design environments. Similarly, class versioning is often used as a means for schema evolution in object databases (Skarra and Zdonik 1986; Bjornerstedt and Hulten 1989; Monk and Sommerville 1993; Rashid, Sawyer, et al. 2000).

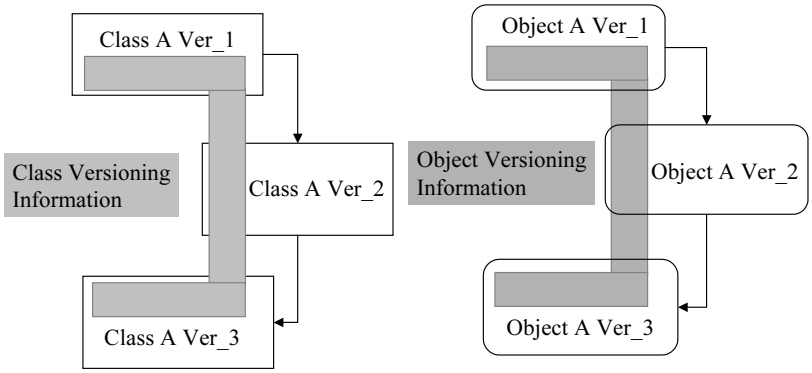

Fig. 3.3. Versioning information crosscutting versioned entities

Versioning information is often embedded within the versioned entities (cf. Fig. 3.3). Consequently, any changes to the versioning information at the DBMS level affect the versioned entities. Even if version derivation graphs (Loomis 1992) are used, the semantics of class and object versioning differ. The version management behaviour has to refer to different sets of graphs. Furthermore, the traversal and update semantics form part of the behaviour of the nodes forming the class and object version derivation graphs. Customisation of such traversal or update behaviour, for instance, to employ a more efficient strategy, can have an impact that is likely to span all the class and object version derivation graphs.

3.2.3 Instance Adaptation Routines

Customisation of instance adaptation in existing object database evolution systems is expensive because they introduce the instance adaptation routines directly into the class versions upon evolution. Often the same adaptation routines are introduced into a number of class versions. Consequently, if the behaviour of a routine needs to be changed maintenance has to be performed on all the class versions in which it was introduced. If the

instance adaptation approach at the DBMS level is customisable, any such customisation or adoption of a new strategy might trigger the need for changes to all or a large number of versions of existing classes. Furthermore, there is a high probability that a number of adaptation routines in a class version will never be invoked as only newer applications will attempt to access properties and methods unavailable for objects associated with the particular class version.

The above shortcomings can be demonstrated by considering instance adaptation in ENCORE (Skarra and Zdonik 1986) which employs a simulation-based approach for the purpose. As shown in Fig. 3.4, applications access instances of a class through a *version set interface* which is the union of the properties and methods defined by all versions of the class. Error handlers are employed to trap incompatibilities between the version set interface and the interface of a particular class version. These handlers ensure that objects associated with the class version simulate the version set interface, i.e., the presence of missing attributes or methods for an object. As shown in Figs. 3.4b and c, error handlers for missing information are introduced into all the former versions of a class upon each additive change: change which modifies the version set interface. If the behaviour of the *address* handler in Fig. 3.4c needs to be customised maintenance has to be carried out on all the former class versions. This reflects the crosscutting nature of instance adaptation routines.

The example in Fig. 3.4 describes a fairly simple evolution scenario. In a more complex scenario, introduction of error handlers into former class versions is a significant overhead as, over the lifetime of a database, a substantial number of hierarchically related class versions will exist prior to the creation of a new one. If the behaviour of some handlers needs to be changed maintenance will have to be performed on all the hierarchically related class versions in which the handlers were introduced. Similarly, changes to the instance adaptation approach or moving to a different strategy will be very expensive too and will trigger changes to a large number, if not all, of the class versions as all the error handlers will need to be removed and adaptation routines (e.g., update/backdate methods (Monk and Sommerville 1993)) required by the new adaptation approach introduced.

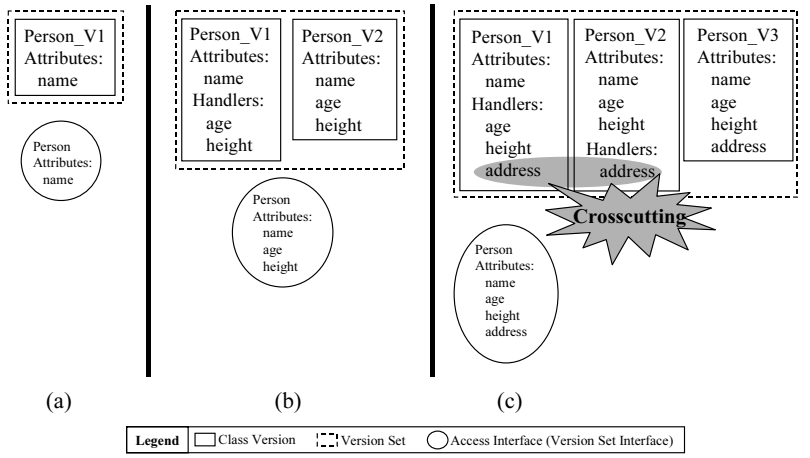

Fig. 3.4a-c. Instance adaptation routines in ENCORE: (**a**) before evolution (**b**) upon an additive change (**c**) upon another additive change

3.2.4 Other Crosscutting Concerns at Database Level

Several other concerns also cut across the data and meta-data residing in the database. These include:

- access rights on data;
- constraints on data;
- data representation.

All the above concerns relate to or affect multiple tables or objects in a database, a crosscutting effect.

3.3 Conclusion

This chapter employed several concrete examples to demonstrate how crosscutting concerns affect the modularity of database systems, at both the DBMS level and the database level. It was argued that this has direct consequences in terms of increasing customisability, adaptability and evolution costs. The knock-on effect of customising DBMS level concerns on database level concerns was also discussed. Although most of the examples were based on object database systems, the discussion can be generalised to other types of database systems. For instance, some of the DBMS-level concerns,

such as the transaction model, security and access control policies (and need for their customisation), exist in all types of database systems. Similarly, most of the evolution concerns discussed can be generalised to include complex data structures and their instances in object-relational database systems.

In Chap. 2 the effectiveness of AOP in addressing crosscutting concerns was described making it an obvious candidate for modularising such concerns in database systems. Chapter 4 discusses how AOP can be employed to address the concerns highlighted in this chapter.

http://www.springer.com/978-3-540-00948-1

Aspect-Oriented Database Systems
Rashid, A.
2004, XVI, 176 p., Hardcover
ISBN: 978-3-540-00948-1