Leonardo Journal of Sciences
ISSN 1583-0233

Issue 5, July-December 2004
p. 46-62

# A Database Integrity Pattern Language

Octavian Paul ROTARU[b], Mircea PETRESCU[a]

[a] *The Computer Science and Engineering Department, University "Politehnica" of Bucharest, Romania*

[b] *On leave of absence from [a], Currently at Amdocs Development Ltd. 141, Omonia Avenue, The Maritime Center, 3045 Limassol, Cyprus*

*Corresponding author:* Octavian.Rotaru@ACM.org, MirceaStelian@Yahoo.com

**Abstract**

Patterns and Pattern Languages are ways to capture experience and make it re-usable for others, and describe best practices and good designs. Patterns are solutions to recurrent problems.

This paper addresses the database integrity problems from a pattern perspective. Even if the number of vendors of database management systems is quite high, the number of available solutions to integrity problems is limited. They all learned from the past experience applying the same solutions over and over again.

The solutions to avoid integrity threats applied to in database management systems (DBMS) can be formalized as a pattern language. Constraints, transactions, locks, etc, are recurrent integrity solutions to integrity threats and therefore they should be treated accordingly, as patterns.

**Keywords**

Concurrency, Data Integrity, Integrity Threat, Data Quality, Locking, Pattern, Pattern Language, Transaction

## 1. Patterns and Pattern Languages

Patterns are experience-capturing methods, able to share it in a consistent manner with others. They are solutions based on experience to recurrent problems, describing best practices and proven designs.

Patterns originate on Christopher Alexander's work on architectural design [16]. Christopher Alexander considers that "each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem in such a way that you can use this solution a million times over, without ever doing it the same way twice" [16].

The "Design Patterns: Elements of reusable Object-Oriented Software" book of Erich Gamma, Richard Helm, Ralph Johnson and John Vlissidess, also known as The Gang of Four (GoF) book [18], had an important role in the pattern "evangelization" of the software engineers. Even if Alexander was an architect and his work refers architectural and urbanism patterns, his view is also valid for object-oriented design and his popularity in software engineering is at least as high.

Experience is an intangible but for sure valuable commodity, which distinguishes a novice from an expert. People acquire it slowly, through hard work and perseverance, and communicating it to the other is a challenge. Design patterns are a promising step towards capturing and communicating expertise in building object-oriented software. [19]

Design patterns provide a common design lexicon, and communicate both the structure of a design and the reasoning behind it [24]. They allow people to understand object-oriented software applications in terms of stylized relationships between program entities. A pattern identifies the roles of the participating entities, the responsibilities of each participant, and the connections between them. The use of patterns also raises the abstraction level at which designers and developers communicate, by providing a high-level shared vocabulary of solutions.

A pattern language is a collection of patterns and the rules to combine them into an architectural style. Pattern languages describe framework of solutions for a certain domain. A software framework is a good example of pattern language. It offers experience-based solutions to recurrent problems, in other word a collection of patterns.

## 2. Data Integrity Definition

Finding a proper definition for data integrity is not at all a trivial task. Integrity is the property of data to reach an a priori quality level, which is adequate and sufficient in a given context [1]. The above definition given by Courtney is one of the most general integrity definitions, but its main problem is that it is based on another concept: data quality.

Data quality is also difficult to define, and some authors define data integrity as an attribute of data quality. This leads to a logical circular dependency from where it is hard to evade. If integrity is the property of data to reach a certain quality level and data quality requires integrity than what is integrity and what is quality?

Furthermore, the definition given by Courtney and Ware considers integrity as a binary property of data. However, this is not the only opinion. Other authors define integrity degrees, in which case integrity becomes a measurement of the current state of the data, compared with the ideal perfect state.

Roskos, Welke, Boone and Mayfield [2] define integrity as having two meanings:

- The state that exists when the electronic data is identical with the data from the source documents or modeled problem domain and it was not exposed to intentional or accidental alterations or destroyed;
- The state that exists when the quality of the stored information is protected from being contaminated or degraded with lower quality information.

Another well-known definition for integrity is Biba's [3], which defines data integrity as the situation in which the flows from low data integrity objects to high data integrity objects are blocked. This definition sees information as a network of integrity labels, where the information flow is allowed only in one direction: from up to down. Biba associates the unauthorized data updates with down – up or lateral data flows in the integrity network. Bell-LaPadula and Denning [5, 6] gave a similar definition for integrity, the opposite direction of the allowed flows because of the reversed positioning of data in the integrity network being the only difference.

Sandhu and Jajodia [4] consider integrity to be insured by blocking or preventing improper modifications of the data. This definition is also incomplete, because it is unclear what improper means for data updates. Improper data modifications include both authorized and unauthorized information altering.

According to Motro [7], information has integrity if it is both complete and valid. Data is complete if it includes all the entities of the modeled real world and data is valid if it doesn't contain information that has no representation in the modeled real world. It is impossible to verify data completeness. However, validity is at least partially verifiable.

Data integrity represents a measurement of the logical data integrity. It can be considered as a representation of the internal data consistency or of how well is data conforming to its internal structure.

Logical data integrity is different from physical data integrity, which refers to the physical organization of data on the storage device. Physical integrity is usually taken care by the database management system and the operating system.

Integrity is also difficult to define because it is context dependent. Pfleeger defines in [8] data integrity as being:

- Precision - data is modified only in acceptable ways;
- Accuracy - date is accurate;
- Un-modifiability - only authorized processes modify data;
- Consistency - data is correct and meaningful.

If accuracy is considered an attribute of integrity as in the Pfleeger's definition, then the demarcation line between data integrity and data quality almost disappears.

If integrity is considered an attribute of data quality then integrity doesn't automatically insure a certain level of quality. Data quality depends on a greater extent on accuracy than on data integrity. Such a context excludes accuracy from the integrity attributes.

In conclusion, data integrity is a big umbrella term that refers to the consistency, accuracy, and correctness of data stored in a database.

## 3. Intent

Design patterns are to great extent work-around solutions for deficiencies in programming languages and technologies. For example, the Visitor pattern was created to overcome the lack of support for double-dispatch in nowadays object-oriented programming languages (OOPL). Design patterns offer a way to improve OOPL by reusing proven solution and to tame complexity. They resolve "misfits", as Christopher Alexander calls them [15, 16, 17].

This paper addresses the database integrity problems from a pattern perspective. Even if the number of vendors of database management systems is quite high, the number of available solutions to integrity problems is limited. They all learned from the past experience applying the same solutions over and over again.

We believe that all the data integrity related solutions applied to in database management systems (DBMS) can be formalized as a pattern language. Constraints, transactions, locks, etc, are recurrent integrity solutions to integrity threats and therefore they should be treated accordingly as patterns.

A database can be described as a pattern language. Locks, transactions, rollback mechanisms, access security, can all be implemented using patterns, integrated into a database pattern language. Also, a database is a solution to repetitive storage problems in the context of an application, and therefore it is a pattern in itself. Databases as entities are encapsulated, abstract, and have the generativity and openness necessary to be considered templates, instantiations of the same large-scale pattern. A database is a pattern or a pattern language depending on the context, to which everything is relative. [26]

Our aim is to capture all the recurrent solutions to database integrity threats into a pattern language. Mature design and engineering disciplines have handbooks describing common solutions to known problems. Database theory is a very mature domain, and therefore our intent to create a solutions handbook out of the proven existent mechanisms for database information integrity is fully justified.

## 4. Data Integrity - Theoretical Fundaments

Let $S$ be the schema of a database $DB$. $S$ defines a set of types $\{T_1, T_2, \ldots, T_n\}$, that determines the vocabulary used to express details about the problem domain.

A database instance $D$ is a set $\{D_1, D_2, \ldots, D_n\}$, where every $D_i$ is a set of instances of $T_i$. A database instance corresponds to a particular description of the modeled real world.

An element $t \in D_i$ is called a detail. Details describe situations that can or cannot appear in the real world modeled by the database. Therefore there are two categories of details: true and false.

For two database instances $D$ and $D'$ we say that $D \subseteq D'$ only if $D_i \subseteq D'_i$ for each i,$1 \le i \le n$. The set of all possible instances of a database is called the universe of a database

$U$ . Also, the rational universe *u* is the set of all the details about the modeled world that can be stored in the database. Therefore

$$u = \{\{D_1, D_2, ..., D_n\} \mid D_i \subseteq T_i, 1 \le i \le n\},$$

and *d* is the database instance which satisfies:

$$d \in U \wedge \forall D \in U : D \subseteq u .$$

Let $M = \{M_1, M_2, ..., M_n\}$ be a database instance in which $M_i = \{t : T_i \mid t$ represents a true situation in the modeled real world $\}$. In this context *M* represents an ideal state of the database, which corresponds completely to the modeled real world. Such an ideal state *M* is called a model.

Database Information integrity has two essential attributes:

- Validity, which guarantees that all false information are excluded from the database;
- Completeness, which guarantees that all true information are included into the database.

Database information validity is usually enforced through internal or external integrity constraints. If no constraint is violated than the database is in a valid state.

An integrity constraint can be defined as a function $f(U)$, which returns a Boolean, and implements a data assertion respected for all valid states of the database. $f$ is an integrity constraint, or more correctly said a validity constraint, if it returns true for every instance *D* from *M*:

$$\forall D \in U : (D \subseteq M \Rightarrow f(D)).$$

A set of validity constraints, which determine the validity state of each database *D* from *U*, is called a validity theory ($T_V$). $T_V$ is validity theory for *M* if:

$$\forall D \in U : (D \subseteq M \Leftrightarrow \forall f \in T_V : f(D)),$$

the validity of each database state *D* can be assessed by evaluating $T_V$ on *D*.

Analog, $f$ is a complete constraint if:

$$\forall D \in U : (M \subseteq D \Rightarrow f(D)).$$

A set of complete constraints is called a completeness theory $T_C$ if:

$$\forall D \in U : (M \subseteq D \Leftrightarrow \forall f \in T_C : f(D)).$$

Considering all the above, a complete integrity control realization based on constraints requires a complete theory *T* for *M*, able to evaluate both validity and completeness:

$$\forall D \in U : (M = D \Leftrightarrow \forall f \in T : f(D)).$$

## 5. Constraint – A Data Correctness Pattern

### *Context*

Some of the tables stored in the database are logically linked. The annexes always refer their master entity; some of the fields should be unique, while some others have only a limited set of possible values.

### *Problem*

Preserve the integrity of the data in the above-described context. Synchronize the links between annexes and master tables; ensure that the uniqueness of the required fields is preserved and that all the fields take values only from their domain.

Constraints can be used to verify the synchronizations between tables as well as the domains of the fields. Their main advantage is the simplicity and straight-forwardness. Constraints are also natively implemented in the databases, being a part of the relational model.

### *Solution*

A constraint is an interrogation that returns as result a zero-degree predicate. If the interrogation returns zero for a certain database D, then D satisfies the constraint or the constraint holds on D [10]. Constraints are assumptions about data that are used to verify the correctness of the data inserted into a database. Correctness is one of the attributes of integrity and therefore it is important in the context of our integrity pattern language.

There are four main types of data integrity, or better-said correctness compared to the domain model, which are addressed using constraints: Entity, Domain, and Referential.

Entity Integrity is a key concept of the relational model and it ensures that each row in the table is uniquely defined, which means that no table will have duplicate rows. Placing a primary key constraint or a unique constraint on a specific column or group of columns is the most often used method of enforcing entity integrity. A set of columns different from the primary key for which a uniqueness constraint is imposed is called and alternate or a

candidate key. The primary key can also be a surrogate key if its column or columns do not contain real data but a uniqueness identifier.

Domain integrity requires that a set of data values will be within a specific domain to be considered valid. Domain integrity restricts the data type, or domain of possible values for a column. Domain integrity is usually enforced using data types, default constraints, NOT NULL constraints, check constraints and foreign keys.

Referential integrity keeps synchronized the relationships between tables. Referential integrity is enforced using a combination of primary key (master table) and foreign key (annex).

Maintaining the referential integrity in case of data manipulations of any kind (inserts, deletions, and updates) was implemented in different ways by the database vendors.

The main referential integrity implementation strategies in case of insert and update operations are:

- Dependency – The strategy of dependent insert allows insert or modifications in the annexes only if a corresponding record already exists in the master table;
- Automatic Insert – Inserts and updates on the annexes are always allowed and the corresponding record from the master table will be created automatically if it doesn't exist;
- NULL Key – Inserts and updates on the annexes are always allowed and the columns from the foreign key will get the NULL value if there is no corresponding entry in the master table;
- Default – In case there is no corresponding record in the master table, the attributes from the annex's foreign key will be set to default values;
- Adaptation – Allows inserts in the annexes if certain validity constraints are enforced;
- No Effect – Inserts and Updates are always allowed in the annexes, without making any check if there is a corresponding entry or not in the master table.

The referential constraints are usually implemented in case of delete using one of the following strategies:

- Restriction – A record from the master table can be deleted only if there are no corresponding records in its annexes;
- Cascade – Always allows the deletion of records from the master entity, the corresponding records from the annexes being automatically deleted;

- NULL Key – The records from the master table can be always deleted and the foreign key attributes of the corresponding records from the annexes will be set to NULL;
- Default – The foreign key attributes from the annex tables are changed to defaults;
- No effect – The deletion of master record is always allowed without any check being made on the existence of related records in the annexes.

## 6. Transaction and Locking Patterns

### *Common Context*

Applications perform units of work on the database that are composed from multiple operations.

### *Common Problem*

The unit of work should be executed atomically and isolated, even if it comprises of multiple operations. It can fail or succeed only in its entirety.

A very famous example of atomic unit of work composed of multiple operations is a bank transfer from one account to another. The transferred amount should be debited from one account and credited to another, after verifying that the debit account has enough money to perform the transaction. Also, the bank transfer should be registered in the transfers log.

This bank transaction is composed of one read operation, two updates and an insert. In case the operation fails in the middle of the transaction, and the amount is only debited from the source account without being credited to the destination account, the consistency of the database will be affected. The database instance after the transaction failure will not reflect the modeled “real world”. The destination account balance will be incorrect and not conform to reality, and therefore the integrity will be affected.

Also, such bank transaction should be isolated from all the other transactions running concomitantly in the system. If two transactions will try to update the balance of the same account in the same time the result will be unpredictable. If two transactions will read the available balance of the account in the same time and they will both try to get money from it, but the amount available will be enough only for one of them to perform what it will be result of such an operation? Is it correct according to the problem domain model or to the business rules that the respective account has a negative balance? In some situations the bank will not

credit at all their clients, while in some others an assessment done by a credit officer will be required.

Every user or instance of the application will relay on a certain working copy in order to make its decisions (update balance, etc…). It is therefore necessary to use a mechanism able to tell if the working copy that I am reading was read by some other instance as well which intends to update it, so that the current instance will wait till the "common resource" is released.

If transactions are running sequentially than every transaction will start from the consistent state resulted at the end of the previous transaction. If transactions run concurrently then locking are required to insure that the database will end-up in a consistent state.

All this situations represents concurrency problems and they are potential threats for the integrity of the database.

***Transaction Pattern Solution***

Implement the units of work comprising multiple operations as transactions. Transactions are serialization mechanisms for such units of work, ensuring that the current operation do not conflict with other concurrent ones.

A transaction is a unit of work that has the following properties:

- Atomicity – the operations inside the transaction are executed as a single operation. If one of the operations of the transaction fails that all the other operations executed until that point are rolled-back;
- Consistency – the database remains in a consistent state after the transaction ends;
- Isolation – Each unit of work is isolated from all the other units of work running in the system and it has the same effect irrespective of states of other transactions;
- Durability – The effect of the transaction is persistent in time once it is committed;

Transactions are difficult to implement because they require complicate locking at the database level. Most of the database management systems provide transaction functionality. This consists usually in boundary demarcation syntaxes, able to mark the beginning and the end of transaction:

- Start – ark the starting point of a transaction;
- Commit – Mark the successful end of a transaction, the point from where the transaction's modifications become persistent;

- Rollback – Mark the failure of a transaction and undo all the transaction's operations.

Apart from the basic functionality presented above, some database management systems also offer some more transaction related functionalities, like save points, which are moments during the transaction that can be used for partial roll-back. A save point is an intermediate step within a transaction which represents its state at a given moment.

When the number of operations contained by a transaction grows the number of locks that it imposes on the data grows as well. As the number of locks that are active at a point of time in the system grows the performance of the database is depreciating. The waiting time required acquiring locks on data increases. The only solution to this is to keep the transaction small, so that the locks will be active for short period of time and another transaction will be able to lock the same data.

The logical structure of a transaction is presented in Figure 1. The concrete classes are the native implementation of the database or data driver.

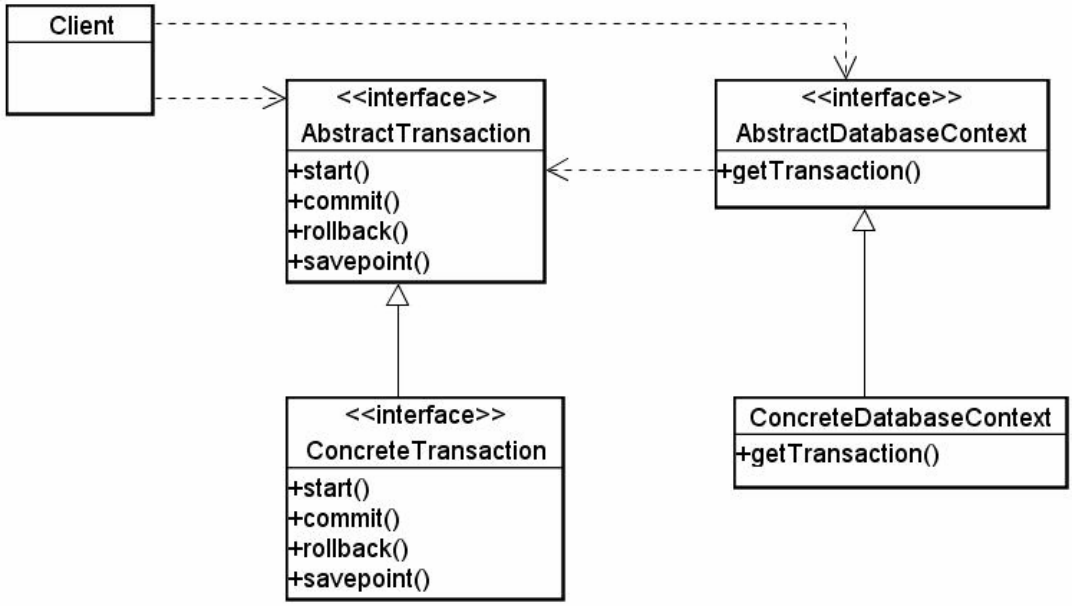


*Figure 1. Database Transactions Structure*

The result of a transaction is committed when the transaction abandon the possibility of rollback to the previous state, making the update values available for read to all the other transactions running in the system. Concurrent execution of transactions raises the problem of reading or writing data, which was modified by another transaction, without being committed yet.

Choosing an appropriate isolation level for your transactions can solve the problems of concurrent read and write. The common isolation levels implemented by most of the DBMS are:

- Read uncommitted
  - A transaction is not overwriting data that was already modified by another transaction;
- Read Committed
  - A transaction is not overwriting data that was already modified by another transaction;
  - A transaction does not commit any update operations before its end.
- Repeatable Read
  - A transaction is not overwriting data that was already modified by another transaction;
  - A transaction does not commit any update operations before its end;
  - A transaction does not read data modified by another uncommitted transaction.
- Serializable
  - A transaction is not overwriting data that was already modified by another transaction;
  - A transaction does not commit any update operations before its end;
  - A transaction does not read data modified by another uncommitted transaction;
  - A transaction does not modify data that was read by another transaction until the end of the transaction that read the data first.

The Read Uncommitted isolation level allows dirty reads, non-repeatable reads and phantom reads, but prevents missing updates. Read Committed also prevents dirty read. Repeatable Reads allows only phantom reads and prevents missing updates, dirty reads, and non-repeatable reads. Serializable isolation level also prevents phantom reads.

Accessing pervasive information using locks significantly increase contention for the data and reduce overall system scalability, allowing only one transaction to view the data in case of serializability.

Usually optimistic concurrency is implemented by attaching version or timestamp information to the updated record or suing state comparison if possible. When the above assumption does not hold true, the potential inconsistent updates will be detected by the version or timestamp check. In such a situation the conflictual update will not be executed and the user notified that the data has been changed.

Timestamp and versioning can be easily implemented in the application even if the server do not provide specific support for it, by adding an extra column to store the version

count or the timestamp. State comparison is more complex since it requires the object to be reread and compared with the original copy. This might involve numerous field comparisons and therefore is more laborious than the use of version count or timestamp.

The main problem of the Optimistic Concurrency Pattern is that users may loose work because the inconsistent update operations are not detected until trying to save to the database. Its main benefit is the lack of locking overhead since no lock is actually imposed on the physical data and therefore do not suffer the impact of associated negative scalability [25].

This pattern is also known as Optimistic Locking [25] or Optimistic Offline Locking [24]. In our opinion the above names are improper since no locking is actually imposed and therefore we decided to rename it in the context of our integrity pattern language to a much proper name, Optimistic Concurrency, even if the above name is well known and is used by many other authors [27, 28, 29…].

### *Locking Concurrency Pattern Solution*

Locking Concurrency Pattern locks the records on which the user is applying modifications. It assumes that the other users of the system will try to update the same data and therefore is trying to lock the required records at every access.

The Locking Concurrency pattern can either be implemented in the applications or the applications can used the native support for locking existent in most of the commercial databases.

In case the Locking Concurrency is implemented in the application it is required to add one more column to the master tables that will specify if the record is locked or not.

The main benefit of the Locking Concurrency pattern is the fact that applications can detect data conflicts before attempting to change the data. Its major disadvantages are the overhead added by locking and the potential orphan locks, occurred when the application crushes or neglects to release all its locks before exiting.

Locking Concurrency and Optimistic Concurrency are complementary, the benefits of one being the drawbacks of the other. This pattern is also known as Pessimistic Locking [25] or Pessimistic Offline Locking [24].

The Locking Concurrency pattern can be used in conjunction with the Resource Timer [25], which will automatically release the locks of the inactive transaction. A resource timer will solve remove the orphan locks when their maximum inactive lifetime will expire. Also,

in case a transaction that already acquired few locks is waiting indefinitely for one more lock, the resource timer will release its locks after some time, releasing the database from potential deadlock situations.

## 7. The Integrity Pattern Language

As shown in Figure 2, the proposed integrity pattern language consists of five patterns that insure the correctness and safe concurrent access to the data, maintaining the integrity of the database at any moment: Constraint, Transaction, Optimistic Concurrency, Locking Concurrency and Resource Timer.

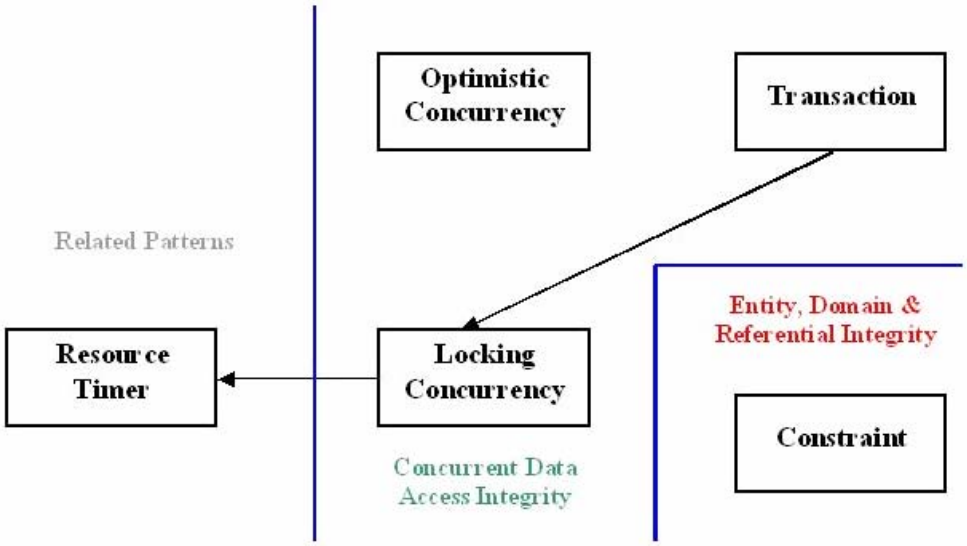


*Figure 2. The Integrity Pattern Language*

Concurrency problems on distributed data sources were not taken into account when defining the integrity pattern language presented here. They will are a future research subject for us and will be detailed in a future paper.

## 8. Conclusions

This paper formalizes the recurrent integrity solutions applied in database systems and applications into an integrity pattern language. We started from the assumption enounced by

us in [26] that the database concept constitutes a pattern in itself, solving repetitive storage problems in the context of an application.

Apart from being a meta-pattern, databases can also be described as a pattern language. Locks, transactions, rollback mechanisms, access security, integrity can all be implemented using patterns, integrated into a database pattern language. This paper constitutes the first step in this direction. We started by describing data integrity as a pattern language and we will continue to do the same for all the other services offered or expected from a database (security, backup, etc.).

We believe that the pattern language presented here is a useful data integrity solutions handbook for all practitioners in the area, containing most of the integrity solutions implemented in the nowadays database systems.

## References

[1] Courtney, R., “Some Informal Comments About Integrity and the Integrity Workshop”, Proc. Of the Invitational Workshop on Data Integrity (Ruthberg, Z.G. and Polk, W.T., editors), National Institute of Standards and Technology, Special Publication 500-168, September 1989, section A.1, pp. 1-18.

[2] Roskos, J.E., Welke, S., Boone, J. and Mayfield, T., ”A taxonomy of integrity models, implementations and mechanisms”, Proc. Of 13th NIST-NCSC National Computer Security Conference, Washington D.C., October 1990.

[3] Biba, K.J., ”Integrity Considerations for Secure Computer Systems”, MITRE TR-3153, Mitre Corp., Bedford, Massachusetts, 1977.

[4] Sandhu, R.S. and Jajodia, S., ”Integrity Mechanisms in Database Management Systems”, Proc. Of 13th NIST-NCSC National Computer Security Conference, Washington D.C., October 1990, pp. 526-540.

[5] Bell, D.E. and LaPadula, L.J., ”Secure Computer Systems: Mathematical Foundations and Model”, M74-244, Mitre Corp., Bedford, Massachusetts, 1975.

[6] Denning, D.E., ”A Lattice Model of Secure Information Flow”, Communications of ACM 19(5), 1975, pp. 236-243.

[7] Motro, A., ”Integrity = Validity + Completeness”, ACM Transactions on Database Systems, 14(4), December 1989, pp. 480-502.

[8] Pfleeger, C.P., ”Security in Computing”, Second Edition, Prentice-Hall, 1997, pp. 5-6.

[9] Grefen, P.W.P.J., ”Combining Theory and Practice in Integrity Control: A declarative Approach to the Specification of a Transaction Modification Subsystem”.

[10] Gupta, A., Sagiv, Y., Ullman, J.D., Widom, J., ”Constraint Checking with Partial Information”, Proc. of the 13th Symposium on Principles of Database Systems, 1994.

[11] Peng, L., Mao, Y., Zdancewic, S., ”Information Integrity Policies” .

[12] Slack, J.M., Unger E.A., ”A Model of Integrity for Object-Oriented Database Systems”, 1992.

[13] Sandhu, R.S., ”On Five Definitions of Data Integrity”, Proc. Of the IFIP WG11.3 Workshop on Database Security, Lake Guntersville, Alabama, September 12-15, 1993.

[14] Proceedings of the Invitational Workshop on Data Integrity, editori Ruthberg, Z.G. si Polk W.T., National Institute of Standards and Technology, Special Publication, 500-168, section A.1, pp 1-5.

[15] A. Aarsten, D. Brugali, G. Menga, “Patterns for Three-Tier Client/Server Applications”, Pattern Languages of Programs (PloP), Monticello, Illinois, 1996.

[16] A. Keller, R. Jensen, S. Agarwal, “Persistence Software: Bridging Object-Oriented Programming and Relational Databases”, ACM SIGMOD, May 1993.

[17] Christopher Alexander, “Notes on the Synthesis of Form”, Harvard University Press, 1964.

[18] Christopher Alexander, S. Ishikawa, M. Silverstein, “A Pattern Language”, Oxford University Press, 1977.

[19] Christopher Alexander, “The Timeless Way of Building”, Oxford University Press, 1979.

[20] D. Parnas, “On the Criteria to be Used in the Decomposition of Systems into Modules”, Communications of the ACM, December 1972.

[21] D. Parnas, “Designing Software for Ease of Extension and Contraction”, IEEE Transactions on Software Engineering, March, 1979.

[22] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, “Design Patterns – Elements of Reusable Object-Oriented Software”, Addison-Wesley, 1995.

[23] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerland, M. Stal, “Pattern-Oriented Software Architecture: A System of Patterns”, Wiley, 1996.

[24] Martin Fowler, D. Rice, M. Foemmel, E. Hieatt, R. Mee, and R. Standfford, “Patterns of Enterprise Application Architecture”, Addison-Wesley, 2002.

[25] Clifton Nock, “Data Access Patterns – Database Interactions in Object-Oriented Applications” Addison-Wesley, Pearson Education, 2004.

[26] Octavian Paul Rotaru and Mircea Petrescu, “An Architectural Pattern Language for Multi-Level Database Access”, Proceedings of INFOS 2005, Cairo, Egypt, March, 2005.

[27] George Lausen, “Concurrency Control in Database Systems: A step towards the integration of optimistic methods and locking”, Proceedings of the ACM’82 Conference, January 1982.

[28] Maurice Helihy, “Optimistic Concurrency Control for Abstract Data Types”, Proceedings of the fifth annual ACM symposium on Principles of distributed programming, November 1986.

[29] H. T. Kung, John T. Robinson, “On optimistic methods for concurrency control”, ACM Transactions on Database Systems (TODS), Volume 6, Issue 2, June 1981.