Table of Contents for Chapter 2

Part 1. Architecture 101 – 'The Language We Speak'

<<< … >>>

Chapter 2. Key Concepts

This chapter introduces basic vocabulary of some major notions that are of primary concern to the Enterprise Architect. We assume that Enterprise Architect has broad understanding of IT disciplines, as well as business context of IT solutions.

We collated here basic primary concepts that are fundamental to the Enterprise Architecture, and usually scattered in the IT literature. At the same time, we aim to provide enough details to grasp these concepts to the sufficient level of detail, beyond your usual glossary.

We explain some fundamentals of standards, business transactions, security etc. This is just an appetiser for the things to come, really.

Standards - 'guiding light'

All revolutionary inventions have been conceived by thinking 'outside the square'.
However, in order to get to the boundary of the square before the leap outside, we need to absorb the achievements and guidelines of geniuses and simply honest quiet achievers who did this before us and helped us to get where we are now.
After all, 'we see so far ahead because we stand on the shoulders of giants'.

Standards may depend on each other. Also, standards will have a different scope – be it industry, ubiquitous technology or geographical location.

Along the dimension of industry or application domain, we can have the standards of roughly following scopes:

- Global Standards and Guidelines that may cut across all industries and geographical boundaries and be recognised by all – like ones endorsed by UN. We can imagine that there will be some Universal Standards. Humans were so ambitious as to send a message into the space, and made our assumptions about the thought processes of some intelligent extra-terrestrial aliens out there in the Universe. Hopefully, they understand us and we can start by agreeing on some standards for communication.
- Industry Standards – recognised by the players in particular industry. Other industries may be blissfully unaware of their existence. For example, stockbroker would not be excited if he or she learned about the role of TCP/IP or XML in their share deals.
- Enterprise Standards, when company may wish to customise some standards (hopefully by narrowing and not by extending the source standard, to stay *compatible* or *standard-compliant*).

Countries may create own standards or customise international standards for better fit to the local conditions. For example, Russia had to create a Cyrillic equivalent for EBCDIC and ASCII, and had to content with drama of losing some special characters (due to 26 letters in the Latin alphabet and 32 letters in Cyrillic).
Some projects may have to create or customise standards too, if project breaks new grounds in the Enterprise, or even in the Industry. For example, project team may have to create programming guidelines and naming conventions for program variables, files, databases and other project artefacts.

Enterprise Architects need to relentlessly verify and test claims for standards-compliance. For example, ECMA distinguishes between standard *conformance* and *certification* [ECMA 1983].

Integration of Business Components and technologies is a major challenge. 'Sort of' compliant products should ring a warning bell, and very likely won't do.

Internet Standards

Internet standards process strikes the fine balance between openness and tight control and governance.

The Internet Society [WWW, ISOC] serves as a standardising body for the Internet community. It is organised and managed by the Internet Architecture Board [WWW, IAB].
The IAB itself relies on the Internet Engineering Task Force [WWW, IETF] for issuing new standards, and on the Internet Assigned Numbers Authority [WWW, IANA] for coordinating values shared among multiple protocols.

The Internet protocol suite is still evolving through the mechanism of Request For Comments (RFC).
The Internet Standards Process is concerned with all protocols, procedures and conventions that are used in or by the Internet. The Internet Standards Process is itself described in RFC 2026.
RFCs always have one version only – updates or revisions receive new RFC numbers. New RFC replaces or 'obsoletes' the old RFC.

The overall goals of the Internet Standards Process are:

- Technical excellence
- Prior implementation and testing
- Clear, concise and easily understood documentation
- Openness and fairness
- Timeliness

True to the well-balanced culture of the Internet community, some of the RFC submissions are impractical at best or somewhat outside the square, particularly those dated April 1 – due to the too much time on somebody's hands, or the need for the relaxing outlet in the usual work pressures.
For instance, RFC 1149 (dated April 1, 1990) describes the transmission of IP datagrams by carrier pigeon, and RFC 1437 (dated April 1, 1993) describes the transmission of people by electronic mail.

RFC may describe an Internet protocol, or just serve as an information document. The *Internet Standards Track* determines workflow of progressing the Internet protocol RFC towards the Standard.
Internet protocol is assigned the *state* and the *status* by the IETF.
An Internet protocol can have one of the following *states*: Standard, Draft Standard, Proposed Standard, Experimental, Informational and Historic.
Protocol *status* can be: Required, Recommended, Elective, Limited Use and Not Recommended.

When a protocol reaches the Standard state, it is assigned a standard number (STD). As with RFC, STD numbers do not change and do not have version numbers.
Therefore, to clearly specify which standard version you referring to, you should state the STD number and all related RFC numbers. For instance, the Domain Name System (DNS) is STD 13 and should be referred in a fully qualified form like "STD-13/RFC1034/RFC1035".

Some of the most important Internet standards are STD 1 (Internet official protocol standards, issued by IAB quarterly) and STD 2 (Assigned Internet Numbers, issued by IANA).

Transactions - 'who is afraid of commitment?'

Transaction is the fundamental concept that transcends every component, layer, technology and process in the Enterprise Architecture.
Transaction is a unit of work that performs a complete well-defined task of storing or retrieving data from repository, transforming data, controlling some process, transmitting data over some communication media, and presenting data to the user or client device in a required form.

In order to be correct, to guarantee determinism with reliable and predictable qualities in handling precious data, transaction must possess ACID properties (explained below).

Some transactions may be simple and flat. Other transactions may be complex and consist of several simpler transactions.
We call this umbrella transaction a nested transaction. Nesting may be more than one level deep.

Transactions may be as fast as lightning, or drag for long period of time.

Distributed transactions introduce some geographical spread. Transaction may be distributed in a sense of many geographically dispersed clients trying to access the same data repository. Or, some client may try to handle several distributed sources of data in the single unit of work.

Transactions may handle massive volumes of data, and utilise some limited system resources to perform the task.
Transaction management is one of the most crucial features and perrenial challenge in the enterprise application design, development and execution.

ACID Properties of Transactions

In the context of transactions, ACID stands for Atomicity, Consistency, Isolation, and Durability:

- ***Atomicity*** – transaction must successfully complete in its entirety. If transaction fails, all unfinished updates must be undone (rolled back) to the state as it was before the transaction. Operation should implement "all or nothing" scenario.
  For example, in the infamous deposit/withdrawal transaction, either part must succeed, or both parts discarded. Otherwise, some customer (either bank or yourself) will be very unhappy
- ***Consistency*** – transaction must not at any time expose inconsistent state of the data in-progress. In other words, a transaction should transform the system from one consistent state to another consistent state.
  In our example, deposited amount should not become available to other transaction until withdrawal is finalised
- ***Isolation*** – transaction must appear to execute independently of other transactions. Due to some limited shared resources, transactions may compete for the same resource. If transaction wants to *isolate* itself from the possible impact of other transaction, it must grab this resource for the duration and lock other transactions out. Other competing transaction must then wait until resource is released, or fail. In a system with high level of concurrency and competition for the shared resource, isolation may severely impact performance. In some instances we may have to find a compromising balance; we may decide to sacrifice some data integrity and safety for better concurrency and throughput of transactions in a system.
  For example, if we run some non-critical report that summarises balances of some accounts, we may decide not to lock all these balances in a process. We risk that other transaction may update some of the balances while our report is running, and report may produce the total amount that is incorrect when report is completed
- ***Durability*** – disastrous system failure is recoverable. If transaction is completed successfully, results of transaction must be persistent, transparently to the client.
  For example, if customer made a deposit, he or she expects the account balance to be adjusted accordingly, and to stay that way until other deposit or withdrawal (even if there is fire or flood in the IT department of the bank).

These ACID properties of transaction guarantee that transactions do not ever get stuck in the incomplete state, the system always appears to be in the consistent state, concurrent transactions appear to be independent, and results of a transaction are persistent.

You can imagine now that complex and nested transactions can make things very tricky. We have to decide what happens when sub-transaction fails, and be able to manage this situation, hopefully. Do we have to rollback the whole lot (if we can), or ignore failure, or proceed doing something different?

We may not want to rollback the whole complex transaction. Really, when we book the succession of flights for the holidays, we may decide to keep two successful bookings, and to seek the alternative for the third one, which causes the problem.

Transaction Processing

Transaction management is too complex and too important to leave it entirely in the hands of application developers.
Open Group consortium proposed X/Open Distributed Transaction Processing (DTP) model. X/Open DTP established a commonly accepted Reference Model for the transaction processing architecture and databases.

Figure depicts main entities and interfaces in a DTP system.

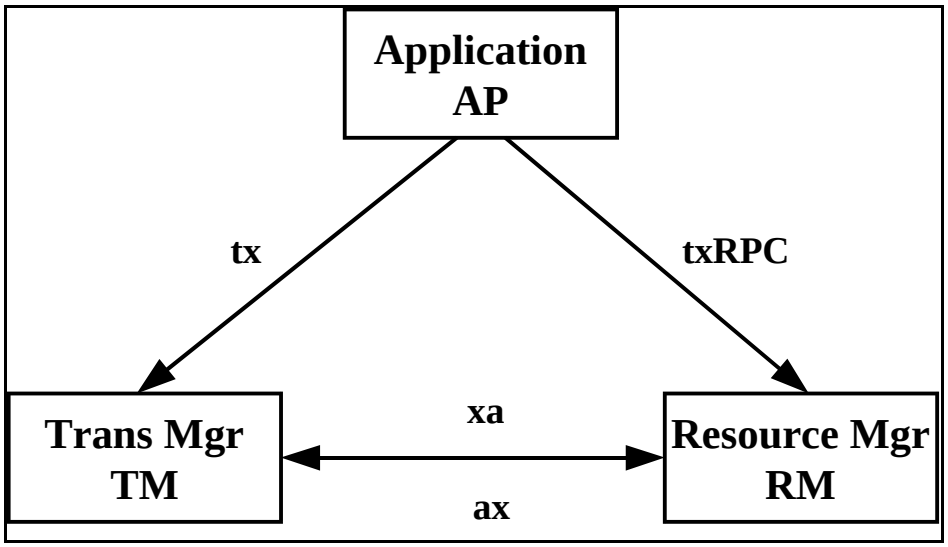

Figure 2.1. X/Open Distributed Transaction Processing Reference Model

Main entities in transaction processing are *Application* (AP), *Resource Manager* (RM), and *Transaction Manager* (TM).

Application is responsible for creating transactions and transaction demarcation, as well as decision to commit or rollback transaction.
Application communicates with *Transaction Manager* through TX interface. TX Interface is the main outcome of the Reference Model that application programmer should be concerned with. Programming languages will provide language-specific bindings to the functions in the TX API, but TX functions and transaction concepts remain the same.
Functions in the TX Interface of the X/Open DTP Model between *Application* and *Transaction Manager* are:

- ***tx_open*** – opens the *Transaction Manager* and the set of associated *Resource Managers*, allocates resources required to maintain the transaction context and execution threads
- ***tx_close*** – closes the *Transaction Manager* and the set of associated *Resource Managers*, releases and delists allocated resources
- ***tx_begin*** – begins the new transaction, or opens the transaction demarcation brackets
- ***tx_rollback*** – rolls back the transaction
- ***tx_commit*** – commits the transaction, directs participating *Resource Managers* to make updates of the transaction pertsistent

- ***tx_set_commit_return*** – commits the transaction
- ***tx_set_transaction_control*** – switches between chained and unchained transaction mode. In chained mode context of parent flat transaction controls commit and rollback of the child transaction
- ***tx_set_transaction_timeout*** – sets a transaction timeout interval
- ***tx_info*** – queries the transaction context

Transactions are identified by transaction id – XID, a data structure that uniquely distinguishes the transaction in the system. Most functions in the reference model take an XID as a parameter.

Resource Manager is a component in the reference model that manages persistent data storage in databases, allocates resources, and participates in the two-phase commit and recovery protocols (2PC) with the *Transaction Manager*.

Two-phase commit protocol ensures that both parties (in this case, *Transaction Manager* and *Resource Manager*), either both successfully complete *commit*, or both fail.
To achieve that, protocol introduces an additional step before actual *commit*. *Transaction Manager* issues a prepare request to all *Resource Managers* involved in the transaction. *Resource Manager* notifies if it is ready to commit. *Transaction Manager* issues a commit request to all *Resource Managers* only when all of them notified about readiness to commit.

Resource Manager may register (or, *enlist*) allocated resources with *Transaction Manager* so that the *Transaction Manager* can keep track of the resources participating in the transaction.

Transaction Manager is the core component of the transaction processing.
Transaction Manager creates transactions on request from the *Application*, establishes transaction context, allows resource enlistment for tracking their usage, and conducts the two-phase commit or recovery protocol with *Resource Managers*.

XA Interface is the bi-directional interface between *Resource Managers* and *Tranasaction Managers*.

XA Interface specifies two sets of functions.
The first set of ***xa_*()*** functions implemented by *Resource Managers* for use by the *Transaction Manager*, and controls the transaction demarcation from TX Interface, as well as two-phase commit.
The second set of ***ax_*()*** functions implemented by the *Transaction Manager* for use by the *Resource Manager* to dynamically register or enlist resources.

Figure does not show *Communication Resource Manager (CRM)* and its interfaces (XA+ Interface and CRM-OSI TP Interface) for interoperability between different transaction managers and domains.

Transaction demarcation defines boundaries of the unit or work, i.e what operations will constitute transaction. From the programmer's point of view, transaction usually represents operations in the control flow between begin/commit brackets.
However, component-based transaction processing systems (like Microsoft Transaction Server, or J2EE Enterprise Java Beans) may employ declarative demarcation, as an alternative to the programmatic demarcation. This way we remove transaction defrinition from the source code and postpone it to the deployment time, and can modify transaction configuration parameters without changing the source code. Component boundaries will implicitly play the role of transaction begin/commit brackets.

Number of commercial transaction management products support TX interface or interoperate with XA-compliant products.
TXSeries/Encina (IBM) and Tuxedo (BEA) support the TX interface. Microsoft Transaction Server can interoperate with XA-compliant databases, such as Oracle.
Most databases and Message Queues (like IBM MQSeries and Microsoft MSMQ Server) provide an implementation of XA interface.

Database Management

Storing large volumes of data and their prompt retrieval is the fundamental feature of the Information Technology, as important as other virtues of computing – voracious number crunching, tireless repetition of pre-defined procedures, connecting the world, presenting content in a form suitable for our consumption.

Database Management Systems (DBMS) prominently feature in the core of the Enterprise Architecture.
More likely than not, your Enterprise will rely on Relational DBMS on storing its precious data.

DBMS provides functionality for access to data by multiple client applications while enforcing the ACID properties of database transactions. Also, DBMS provides complete suite of utilities for the housekeeping of databases.

Persistent and Transient Data

Processes and entities in our IT solutions possess some *state* - attributes or parameters, or data.
You may require some data briefly for the duration of transaction only, or you store these data away for the future use, or for use by other processes and transactions.

We call data *transient*, if they exist for the duration of transaction, and are lost or discarded after transaction completed.
Conversely, we call data *persistent*, if data can be stored away for the future use, likely by the different process. Storage, safekeeping and access to deposited data are known summarily as *persistence*.

Create, Read, Update, Delete (CRUD)

In order to exercise control over persistent data, at the very least, we should be able to control the whole life cycle of data persistence:

- **C**reate new data in the storage
- **R**ead them back into the process when required
- **U**pdate or modify data when only part of it needs to be changed
- **D**elete the data when they are no longer needed

Understandably, this complete basic set of Data Management operations is lovingly called CRUD.

If you neglected to ensure completeness of CRUD operations for each piece of the application data while building your IT solution – rest assured you will be reminded before long.
For example, you may create some lookup table thinking that it never changes ('C' and 'R' – your main concern in this instance). Then time comes when you do have to update the table, or delete it (possibly temporarily, as part of housekeeping in your database) – you need processes for your 'U' and 'D' as well.

Data Modeling

In IT, we represent real-life systems and applications by modeling real entities and processes in their dynamics. Entities and relationships in the system in any given point in time constitute the system's state.
In its evolution and transformation, any complex process may be broken into discrete steps, when system starts off in some state and, through some transformation, morphs into another state.

We capture the state of the system in the Information Model of our application domain.
Data Modeling is the process of defining the data structures and relationships between parts of data for the purpose of providing the system state repository for the software application.
In other words, Data Modeling is the process of building the Information Model that follows the semantics and constraints of given database management tool.

We need to be that specific and limited from the start, because our modeling techniques depend on the idiosyncrocies of Database Management System and Data Modeling notation that we use (to some extent, at least). But that should not come as a shock, as we need to sacrifice some precision in modeling of the infinitely complex real world anyway.

Historically, major types of conceptual Data Models in modern DBMSs are *Relational*, *Hierarchical*, *Network*, *Object*, or mix of any of the above.

Obect Databases represent the Object-Oriented view of the world, and yet to make a dent in the commercial database market.

Hierarchical databases are represented by the IMS – old IBM workhorse that still diligently runs huge databases, mostly on mainframes, or Big Iron.
Hierarchical *conceptual* Data Model made a glorious comeback in the shape of LDAP and X.500 Directories. We stress the point of *conceptual* or *logical* Data Model, because on the *physical* level hierarchical data structures may be stored in non-hierarchical databases. For instance, IBM and Oracle implement LDAP based on data repository in Relational DB2 and Oracle databases respectively.

Network Data Model was represented by databases that comply with ANSI CODASYL standards. We can hardly expect many new deployments of Network databases. Note, in the context of discussion of Data Models, Network databases have nothing to do with Networks and Communications.
On the conceptual level, major difference between Network and Hierarchical Data Models is that some entries may have more than one parent entry, thus making the visual representation of the Data Model looking like net, and not like strictly hierarchical tree of entries.
Network Data Models may be partially implemented in Hierarchical databases, if entries may be indexed by more than one key, and retrieved through more than one access path. For instance, IMS has this capability.

Relational databases have won the major mindshare and become the *de-facto* standard for the Enterprise Data Models.
SQL (pronounced *Sequel*, and stands for Structured Query Language - sounds like rather meaningless term now) has become the *lingua franca* for data definition, data access, and data manipulation.
Relational databases have achieved their popularity to the great extent due to the natural, simple, well-understood and accepted Relational conceptual Data Model.
Relational databases implement some object-oriented features in their data typing as well, providing an excuse to call such databases *object-relational*, and to keep the object-orientation purists happy.

Relational databases define and manipulate data that are represented by the simple two-dimensional *tables*. Tables consist of *rows* (or, sometimes, *tuples*). Every row in the table consists of pre-defined set of attributes, or *columns*.
Database consists of tables, table indexes or keys, relationships between tables (defined by means of their keys), and access and storage rules. Database Schema defines the structure of rows in the table, like type of the values stored in the column, order of columns in the table, indexes for the table etc.

It is good to know that Relational Databases and Data Models benefit from the theoretical rigours of mathematics and relational algebra. However, implementations of databases seek pragmatic compromises and do not follow the theory to the letter.
For instance, relational theory requires enforcement of single unique key in every table. In theory, any complex database can be designed in such a way that this constraint is satisfied (or, so theory says). Data Modellers see the mandatory single unique key as too restrictive in the real situations, and demanded from the database vendors not to enforce this constraint in their products.

As mentioned, definition of the relational table describes attributes in the rows of the table. Every attribute in every row can have a single value of the type defined in the Database Schema.
So, as long as you represent your data in the shape of such simple two-dimensional table, you can store your data in the Relational Database, even if your Data Model is a mess as far as application concerned. Database Management System, or DBMS, is inclined to trust you to infuse a good sense and semantical correctness in your Data Model.
Powerful database engine and seemingly simple Data Model does not relieve you from responsibility to capture the semantics of your Information Model correctly. Failure to do so will expose your

application to the real possibility of inefficient and convoluted software processes, and to the compromised data integrity in your database.

Process of designing the semantically correct Data Model is called *Data Normalisation*. First iteration in data normalization is representing your data in the two-dimentional tables, any way you like. We say this version of the Data Model is in 1st *Normal Form*, or 1NF.

Upon further analysis of the Data Model in its 1NF, we may notice that some attributes in the table row do not fully depend on the unique key of this table. This may cause some anomalies in the data update and data retrieval later. We strive to achieve that every attribute in the table fully depend on its unique key.
We may have to split the table in two in order to de-couple the attribute from the table key that is not really the key for this attribute.
If every non-key attribute in the table depends on the table key, we say that this table is in the 2nd *Normal Form*, or 2NF.

Furthermore, some attributes in the 2NF table may depend not just on the table key, but on other attribute as well. Or, in other words, we may have transitive relationships between attributes in the table. Such relationships in the table also can cause anomalies in data management later in the life cycle of your application. Again, we remove this anomaly by splitting the table in two, so that any non-key attribute depends on the table key attribute, and only on key attribute. In this case, we say that this table is in the 3rd *Normal Form*, or 3NF.

As a rule, Data Modeller should strive to design the conceptual Data Model in 3NF.

Theory and process of data normalization, as well as data management anomalies that may happen if we neglect to normalize our data, are well documented in numerous database textbooks.
Data normalization may sound simple, but may be very difficult to achieve, and many databases settle for practical compromises. That is perfectly fine, as long as benefits and importance of data normalization are well understood, and the best attempt to achieve the 3NF has been made.

Correct, efficient and robust Data Model is one of the most rewarding parts in your overall Software Architecture design. And the opposite is true – sloppy Data Model can create problems and costs blowout from the inception of your application, and throughout its life cycle.

Backup, Restore and Recovery

Your application data are stored in databases, tables and table spaces – in terms of the relational databases.
Or, you store your persistent data in a File System of your Operating System – in file directories and files.

Backup implies that you copy your persistent data for safekeeping from the primary repository to your secondary storage (possibly cheaper, slower and with substantially greater capacity).
Sometimes backup is done by copying data onto similar media, just to create another spare copy of data. For instance, backup could be a copy into another directory on the disk, or mirror database on the Business Continuity Volume (BCV) on the Storage Area Network (SAN).

What is the use of the backup, if you cannot re-create your stored away data when you've lost an original? Every backup procedure must have a corresponding *restore* that brings back data as it was at the moment of backup.
Backup and restore can be lengthy and resource-hogging processes. They are largely non-functional, i.e. may be invisible to the business. These Data Management processes may be perceived as an overhead, or even disruptive for some database business activities.

However, think about the risk of loosing your data altogether, and what it will cost your business in the likely worst scenario.

In short, your backup strategy is the exercise in trade-off, and is function of the cost of backup, how much of recent database updates business conceivably can afford to lose, and how long and disruptive the restore, rollback, or recovery of the database.
Necessary database housekeeping may cause downtime and disrupt some business functions.

Recovery of the database means repair without the need for the complete restore. Database Management Systems are capable of recovering the Data Integrity of your databases with the minimal loss (if any) of the latest updates, without the human intervention, and in the shortest possible time. Now we take this small but mission-critical miracle for granted. Often, this is just a quiet achievement of the Database Administrator who wisely did his homework up-front. DBA attracts attention in the Enterprise only when something goes wrong with our precious data.

Database recovery relies on mechanisms 'under the hood' in the DBMS that are described in the next section.

Rollback, Checkpoint and the Database Transaction

Database Transaction is the process of manipulating the persistent data that possess ACID properties.

Transaction is denoted by, and is contained in *begin* and *commit* brackets – this is the way of telling DBMS to enforce ACID properties for the process within the brackets.

As with non-database transactions, *two-phase commit* (2PC) is used to enforce the Data Integrity in the transaction.

If the Database Transaction encounters trouble, DBMS is responsible for the *rollback* to the beginning of the transaction, or to the explicitly set *checkpoint*, whichever happened last.
Long running transactions (like batch database updates) may set database checkpoints at a reasonable intervals between *begin* and *commit* transactional brackets, so that when batch update fails we do not have to roll back all the way to the start of the batch – only to the latest checkpoint, and re-commence the batch process from there. This is provided the batch program was build with such resilient restart logic, to the much joy of the support personnel.

Ensuring the Data Integrity may seem to be simple. However, consider simultaneous access with updates to many different pieces of data by many processes. Data Integrity may be corrupted even when single process is behaving seemingly well, but concurrent updates from two processes clash on the same data.
In order to ensure integrity of updates, the client process must lock the data for the duration of update transaction. Even if Data Integrity is not compromised, poorly planned data locking strategies may cause lockdown of processes and whole applications ('system hangs').

Locking strategy is very important both for Data Integrity of updates, and for smooth timely execution of multiple processes simultaneously, possibly clashing on the same pieces of data (rows, tables, indices etc.) and database resources (threads, connections, devices).
Good DBMS engine may be quite forgiving in handling multiple competing transactions, but never forgiving enough to relieve the application developer completely from diligent and respectful use of shared resources. Application developer should have a good understanding of what operations will require locking of resources, possibly implicitly 'under the hood', without the clear hints in the source code about the locks being taken.

Transaction process must do its job in managing the state of data in the first place.
At the same time, transaction must be a good citizen in the company of other transactions, so it does not hog resources unnecessarily. Apart from delays in transaction caused by reasonable waiting for other transactions to release required resources, complex transactions may create a deadlock that would never be broken by simple waiting – some drastic measures need to be taken to break the deadlock.

Transaction processes or, more likely, DBMS engine can be clever enough to break the deadlock by sacrificing one of the competing transactions (eg, by enforcing timeout).

Otherwise, we resort to even more drastic measures, like when we lose control over mouse and keyboard on PC, and pull the plug to reboot it (hoping, that no data will be lost or corrupted in the process).

Figure depicts the scenario where two simple transactions, both trying to use two of the same resources, can get in the deadlock situation.
Transaction A started off by successfully acquiring the lock on Resource X, with intent to update it later, or ensure that the state of resource does not get changed by someone else while Transaction A is doing its job. Likewise, Transaction B successfully acquired lock on Resource Y.
Now, in order to continue it's processing, Transaction A tries to acquire the lock on Resource Y as well. But remember, Resource Y is already locked by Transaction B. Transaction A has to wait until B finishes.
Fair enough, Transaction A patiently waits for B to finish and to release Resource Y eventually. However, Transaction B needs resource X now, and cannot get it. We've got a deadlock. Both transactions wait for each other to finish, but neither will.

Only way out from the deadlock is to break some of the locks by sacrificing at least one transaction, so that another transaction can complete successfully.
In our case, DBMS or Transaction Manager (whichever controls our transaction processes) may decide to kill Transaction B. Transaction B will rollback all partial updates it made after the *begin*, and release all locks it has taken. Finally, Transaction A succeeds in locking the Resource Y and happily reaches the *commit* point.

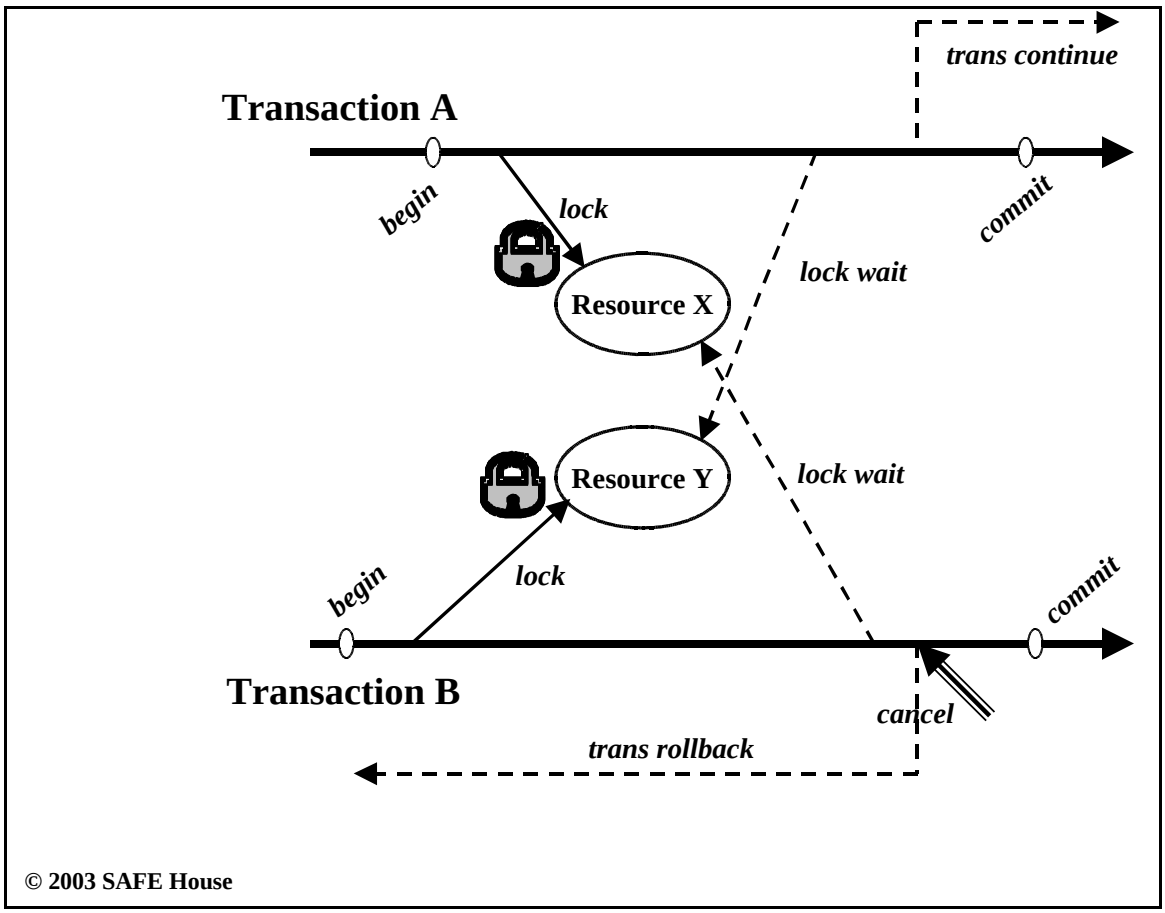

Figure 2.2. Transaction Deadlock

We walked through the deadlock scenario with two transactions and two resources. Real applications may encounter deadlocks involving many transactions and resources. Such complex deadlocks may be difficult to pinpoint and replicate in the testing environment. System may appear to hang under workload at random intervals, and under different circumstances.
Application developers can save a lot of grief by adhering to the guidelines of designing the well-behaved transactions.

Rules of thumb for avoiding transaction locking (or, rather reducing the probability of deadlocks) in the application development:

- Acquire locks to resources strictly on 'need to have' basis
- Lock resources as close as possible to the point of actual use of resource in the transaction. Otherwise, as close as possible to the beginning of the transaction
- Lock as fewer resources as possible, not more resources than you really need. Sometimes, locking lots of fine-granular resources is too expensive – you may have to go for bigger lock. Eg, locking thousands of rows in the table (and indices along the way) may be impractical – you might judiciously consider locking the whole SQL table instead, and run this transaction when business is slow
- Hold locks for the shortest possible period of time, not any longer than you need them
- Unlock resources in the reverse order of locking them
- Make sure you promptly and, preferably, explicitly release the locks in your code

As far as DBMS concerned, all business transactions translate into chronological sequence of database access requests. In case of relational databases, SQL statements will represent all DB access requests. DBMS executes the sequence of SQL statements, does actual database updates, and diligently records all what it has done into the activity log, in chronological order with the timestamp attached to every record.
Let's outline the essence of the database procedures that ensure data integrity. Design, development, and support of these procedures constitute primary responsibility of the Database Administrator (DBA).

DBA sets up procedure for creating the backup of the database, in case business suffers the loss or corruption of the database, or requires to ship data elsewhere.
First, DBA has to decide how often backups need to be taken. Having the full backup of the database for the certain point in time implies that we can restore our database to the state as it was back then if need be. Restore means complete overriding the current database with the data from the backup. Ideally, we want our data to be restored without any loss of previous updates.

If database size is reasonably small, and full backup of the database image does not take too much time and space, and not too disruptive to the business, then taking the full backup from time to time is all that DBA needs. Frequency of backups will depend on intensity of update activities in the database.

Frequent full backups of the database may be impractical or impossible, especially if the database is large and constantly in use. DBA can choose to perform series of incremental backups that store only latest batch of updates, or some part of the database.
Either way, when it comes to restore, DBA will have to produce the full backup of the database image first. DBA procedures can merge the incremental backups into the latest full backup, and produce the version of the full backup for the later point in time, corresponding to the latest incremental backup. DBMS facilitates housekeeping of backups by keeping track of them in the database dictionary.

If DBA wants to restore database to the some point in time between the two backups (rather unfortunate situation to be avoided), DBA restores the database from backups first, and then instructs DBMS to rollback or, accordingly, roll forward updates from the restored state by re-tracing the updates recorded in the activity log.
Note that activity log may grow very large very quickly for active databases, and requires periodic archiving and cleanup. If your recovery procedure relies on activity log, make sure your transactions have not been archived yet.
This is all in the day's work of the Database Administrator.

In general, DBA deals with following database entities in managing the data integrity: database itself, full backup of the database image, incremental backup of the latest updates, and database activity log with chronological record of all activities against the database.

DBMS provides the DBA with complete set of utilities for every procedure that helps to ensure the data integrity. DBA puts together procedures that are aligned with the business requirements for the database integrity, performance, and availability.

As backup is the snapshot of the database state at some point in time, DBMS requires database to be in the consistent state at that moment.
Guaranteed consistency of the database is achieved by taking the checkpoint. Database checkpoint means that there are no unfinished transactions in the system – all outstanding transactions reached their commit, and no new transactions were allowed to begin. Checkpoint may take time, but there cannot be guaranteed data integrity of the backup otherwise. In essence, checkpoint is the lock on the whole database.

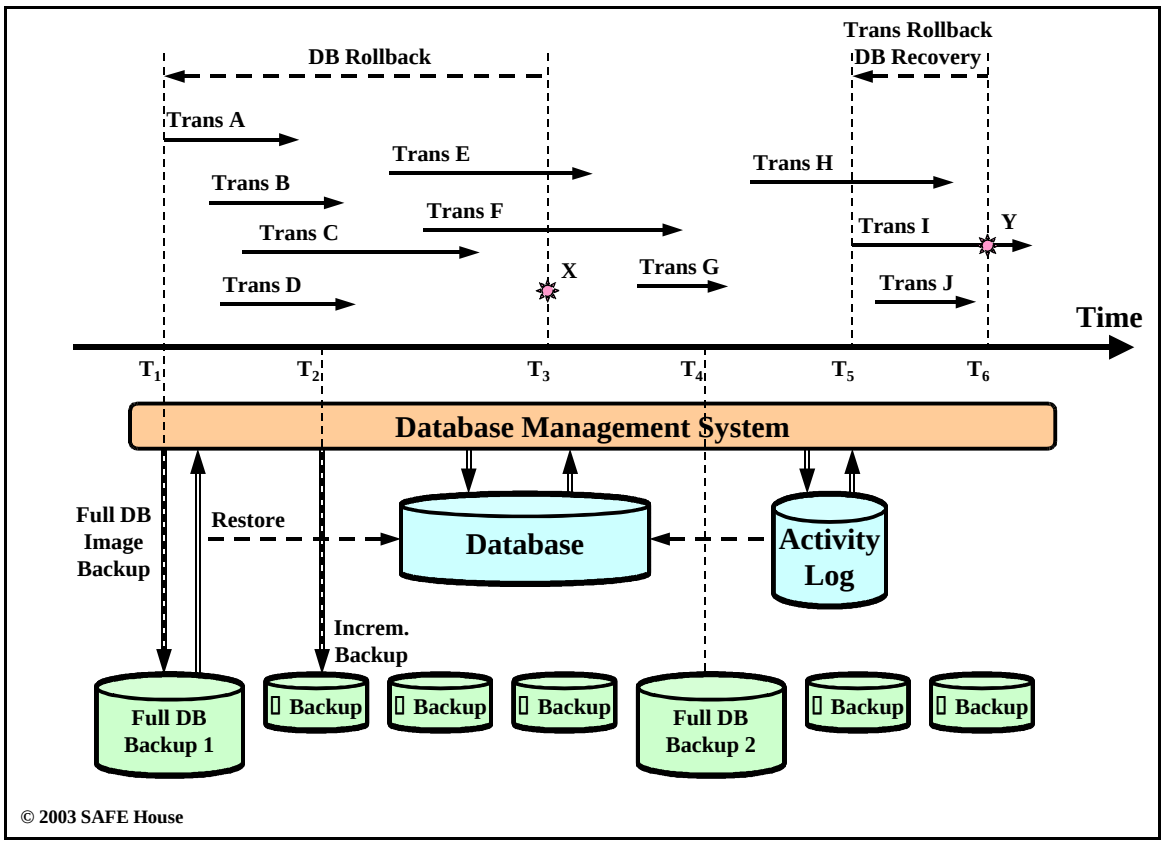

Figure 2.3. Database Rollback and Recovery

Figure 2.3 provides the high level view on database entities and processes that take part in development and execution of DBA procedures.

Database failure X at the point in time T_3 necessitates database rollback to the point in time T_1 when we performed the Full DB Backup 1. Database rollback is achieved by restoring the whole database from the available backup.
All updates performad by Transactions A to F are lost. We may attempt to salvage some database updates that happened between T_1 and T_3 by rolling forward incremental backups and the records from the activity log, by strictly following the chronological order of updates.

Even if we manage to restore database to the point X (or, T_3), updates from Transactions E and F will be lost, as they did not reach the *commit*.

Transaction I failure at the point in time Y (or, T_6) caused the rollback of transaction to it's *begin* (or, T_5).
Transaction I did not reach the *commit*. All partial updates are stored in DBMS buffers or in the activity log. Due to the ACID properties of the transaction, rollback of Transaction I did not affect Transactions H and J.

Replication and Propagation

Database replication process creates a full copy and another instance of the database.
Propagation process maintains one or more copy of the database by shipping or publishing the incremental updates and applying them to the replicas.

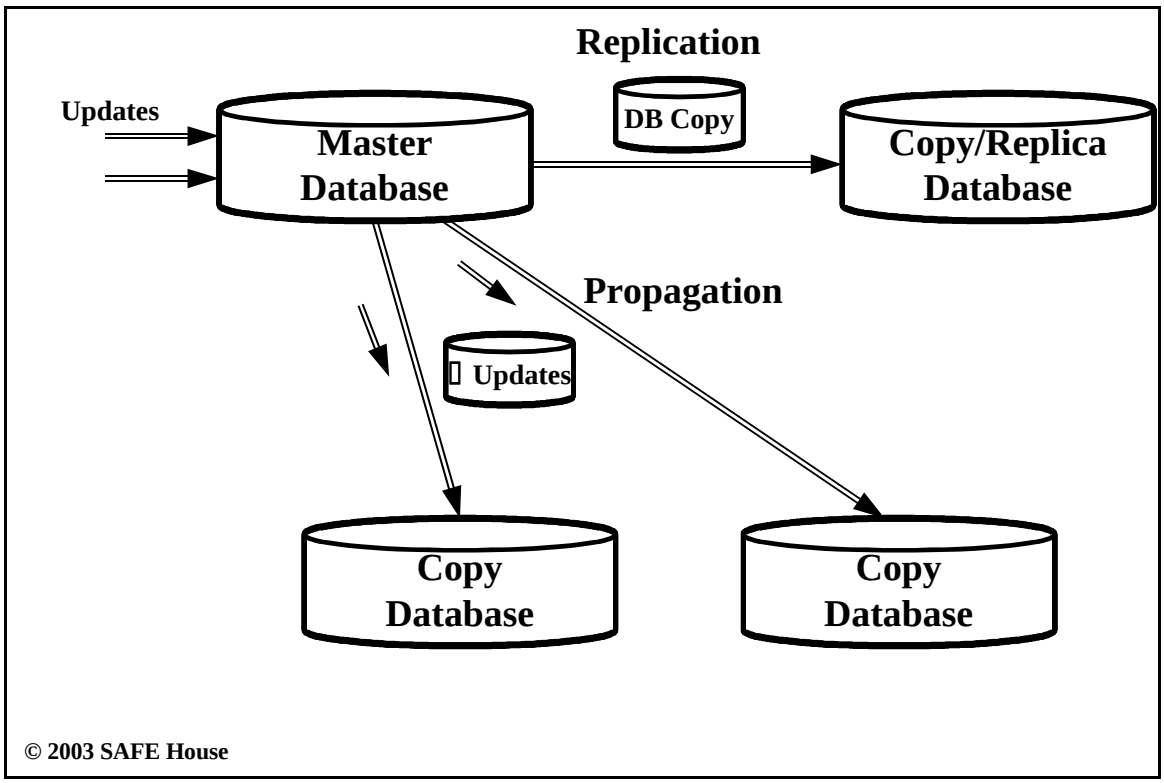

Figure 2.4. Database Replication and Propagation

Creating and maintaining the synchronized copies of databases is an important part of building the distraibuted architectures – common challenge of the modern enterprise with geographically dispersed components and access points, and high level of mobility of users and customers.

Application Life Cycle - 'miracles of life'

"All happy families are like any other in their happiness, each unhappy family is different..."
Leo Tolstoy, "Anna Karenina"

Every application evolves through its distinct life cycle – from the birth (inception) to the death (or decommissioning).

The historically important Waterfall Lifecycle Model was developed around 1970. Waterfall Lifecycle Model, for many years mandated by DoD Military Standard 2167, introduced the sequential stages (hence, 'waterfall') of the software life cycle – System Requirements, Software Requirements, Analysis, Program Design, Implementation, Testing, Operations.

In 1988, Barry Boehm introduced the Waterfall Spiral Lifecycle Model. Spiral Lifecycle Model recognised the repetitive and incremental nature of the software development process. Boehm observed that not just stages, but certain lifecycle activities might be performed several times in sequence to arrive at the final product. For instance, risk analysis, reviews, specification design with finer detail, project planning are repeated on every spiral of the lifecycle.

Various methodologies will argue about stages and phases, and how to name them. Likely, you will have a customisation of some methodology accepted as a standard for your Enterprise.

However, plain common sense will keep you on the safe ground. Typical phases of the life cycle of some stand-alone application unfold as follows:

- Business Idea
- Viability and Feasibility (technical, business and financial)
- Design
- Development, Procurement, Customisation and Integration
- Testing (Unit Testing, System Integration Testing, Stress and Volume Testing)
- Implementation, Deployment, and Migration of deprecated functions, products, databases
- Support and Maintenance
- Deprecation and Decommissioning

Various enhancements or support activities may have a complete life cycle of their own.

Several applications on different stages of the life cycle may compete for the same resources and infrastructure in the Enterprise. Teamwork and the smooth collaboration of all Enterprise units and stakeholders on every stage are of paramount importance.

There is no universal methodology that can suit perfectly every project in every enterprise at any time. However, any reasonable methodology is better than none. And any good methodology can be taken to the extreme when it may become counter-productive and start 'spinning the wheels'.

Some of the most common methodologies for building the Enterprise Architectures will be covered later in a book.

Integration Interfaces – 'are you talking to me?'

Enterprise Architecture consist of many, seemingly vastly different, components and products. Every component communicates with the surrounding environment following certain well-defined rules, and environment (other components) should follow the same rules. You cannot expect the whole system behave well, if rules of communication between components are broken.

Ideally, we should not care what is inside component, as soon as it behaves well and does its job. We call such component a 'black box' – we do not see what's inside, just shape and surface.

The same with components – we publish the interface by which component communicates with the outside world. We do not want any surprises and 'undocumented features' – talk to component through published interface, and only interface, and, ideally, do not care what is behind the interface, as long as we get what we are asking for.
Interface defines our rules of engagement with the component. We integrate complex systems by snapping components one to another – like plug and socket, or peg and hole, if you like. Obviously, it would be very frustrating to try pushing the round peg into square hole.

Hence - Integration Interfaces. Quality, completeness, openness and stability of Integration Interfaces are our primary concern in building the Enterprise Architectures.

Public Interface or Contract

Public Interface or Contract – very clear notions that can be easily understood in layman terms. However, these concepts came directly from the rigorous Object-Oriented methodology and will be explained there in more detail.

Public Interface of the component represents it's publicly stated obligation to provide capabilities and ways to make use of them by clients or other components.

Stable and open Public Interface of the component is the very foundation of well-put Enterprise Architecture, with good separation of concerns and re-use.
Really, if outside world knows about the black-box component through its Public Interface only – we can isolate this component and replace it with another black box without the impact on other components. We may wish to consider such substitution because other black-box component may be cheaper, better (possibly, not available to us initially, e.g. newer release or another implementation altogether), or more suitable to our overall architecture.

Published Public Interface better be stable. As soon as you open it up to the world, and dozens of applications start using it, you won't be able to easily deprecate function that is in use already, and many mission-critical applications may rely on it to be there – good or bad.

Rigorous definition of interfaces between components forms the very core of the Object-Oriented approach in building computer systems.

Bertrand Meyer elevated the consistent and uncompromising methodology for formal specification of interfaces to the *Design by Contract*™ theory. These ideas were implemented in *Eiffel* – programming language, as well as development and execution platform [WWW Eiffel].

Taken to the extreme, formal specification languages can produce the executable code. Practical commercial implementations require some rational compromises.
Still, ability to enforce traceability from the business requirements, to formal specifications, and further to the executable code, can be appreciated by the IT professional in any application domain.

Quest for the bug-free Object-Oriented software, with the ability to *prove* and *enforce* the correctness and the predictable high quality of the software, requires more formalism than the average commercial enterprise application is prepared to sustain. However, if you build the highly reliable mission-critical control system for the nuclear reactor or space program, you will pursue all rigour you can get in order to reach the required level of comfort in quality assurance.

More likely than not, your average target platform or methodology will not *enforce* formal specifications all the way through the application life cycle, or in every component of the framework. Even when your deliverable is a paper-based design, you will benefit immensely in building the component architectures from the understanding of fundamental notion of the Public Interface, and from ideas and motivation behind the *Design by Contract*™.

Application Programmer Interface (API)

Application Programmer Interface (API) is the more detailed design and the specific implementation of the Public Interface or Contract.
Programmer can manage communication with component or program by setting parameters or variables, and invoking methods or functions defined in the API.

API should be complete and stable, and handle abnormal situations in well-behaved fashion.
Errors or abnormal conditions (as far as intended behavior of the invoked function concerned) are handled in the API through the definition and use of return codes, error codes, and exceptions.

API does not have to be Object-Oriented, and it won't be if it is written for non-OO programming language.

Examples of APIs are all around us. Open any Developer's Guide or System Programmer's Guide of any software product, and you most likely will see its documented API.

Separation of Concerns and N-tiered Designs - 'divide and conquer'

Old rule of medieval politicians still applies – 'divide and conquer'.

Enterprise Architectures are very complex beasts with many intertwining parts and layers. And you do not want to build all of it from scratch every time when putting together the solution for your particular business problem.
This is where old proven approach of separating parts and layers, and attacking them one by one, comes handy. And you may start looking around for pre-fabricated parts that will do at least some job for you.

We attack complexity of the Enterprise Architectures on two fronts – reduction in volume of entities and relationships we have to look at together and at the same time, and raising the level of abstraction as high as we can get away with.

Separation of Concerns starts with identifying the components by de-coupling them from other components and the system context, so that we can focus on them in our further analysis without external influences and disruptions.
No component or entity of the real world exists in the complete isolation from other entities. However, we try to define the boundaries of components as best as we can. We call components loosely coupled if they have less, and more simple ties between each other.

Human brain has a limited capacity for holding many concepts in the memory at the same time (some studies mention the magic number seven). Therefore, the ultimate litmus test for the quality of design is how many entities, relationships, and their attributes or properties we have to consider in any part of the design.
Keeping our analysis on the higher level of abstraction, with coarser level of detail or lesser fidelity, also helps in reducing the cardinality in the view of the system. We then gradually drill down by turning the fidelity level up a notch, and by tracing the control flow through the more detailed landscape. Every consequent level of detail, or step in the control flow, corresponds to another *layer* or *tier* in the architecture.

We succeed in studying, designing and managing complex architectures by the systematic and recursive application of the most common patterns in the human intelligence – *Separation of Concerns* and *Layering*.
We shall return to the discussion of Separation of Concerns and Layering in the Chapter 3 – Quality Measures.

Security and Access Control - 'your ticket, please'

User Identity and User Profile - 'who are you?'

> *"In these days of computers and conformity life is becoming more and more depersonalised. People feel the need for self-identification..."*
> *Michael Reynolds, heraldic arms tradesman,*
> *The Australian Women's Weekly, May 29, 1968, p.13*

Every customer or user should become known to the Enterprise and get some kind of unique User Identity or Id, before customer allowed to conduct business transaction.

This Id (Principal) allows us to recognize this customer for the purpose of conducting the business transaction with him, her, or it. We need to know the customer profile, depending on kind of business we conduct with the customer, and attach this Id to his or her profile (so that we find it when required).

Authentication and Authorization, Security Standards

Authentication is the process by which we comfortably identify who the customer is and if we talking to the right person. Depending on criticality of the business transaction that we are going to conduct, we may require various levels of comfort, reliability or the strength of authentication.
If your kid knows the PIN on your credit card – bank ATM would not have a clue that this was not you.

Let's introduce some terms for access control in light of ISO 10181 (or X.800 series) standards. These standards underpin many seemingly different security frameworks and help to find commonalities between them, despite the differences in implementation and local terminology.
For instance, it is useful to see Java or .NET Security Frameworks, as well as dedicated security products like Netegrity SiteMinder, in light of X.800 (even if product documentation makes no explicit mention of security standards).
We explain a rudimentary access control request and components that come into play.

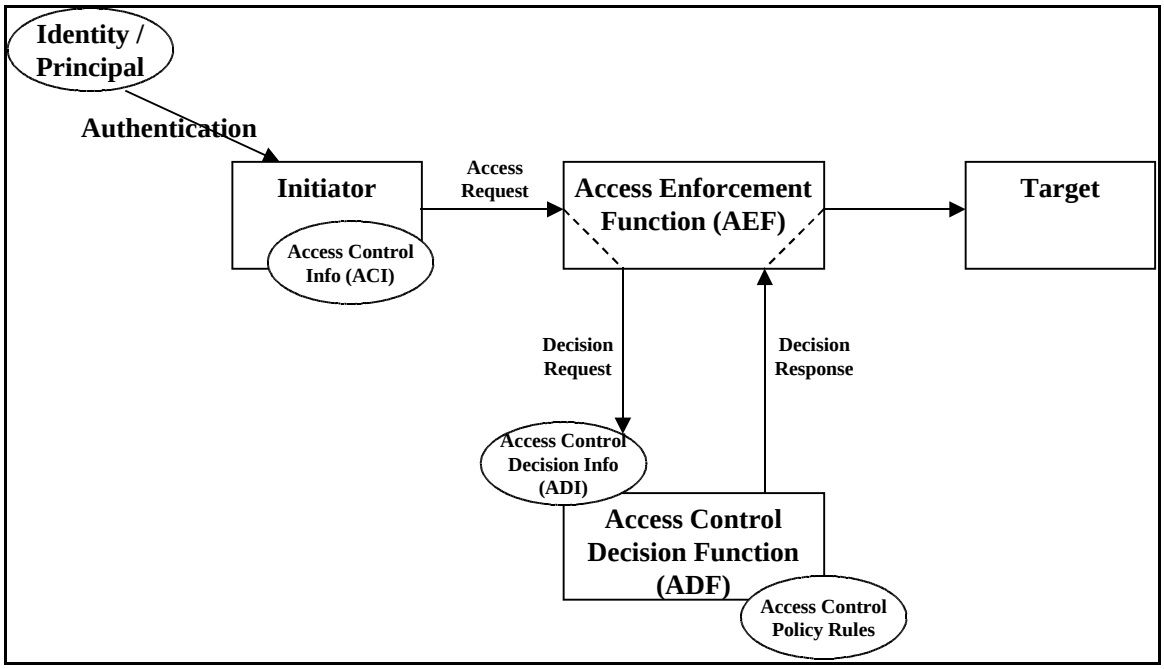

Figure 2.5 Access Control Terms and Concepts

In the end of the day, whatever the authentication procedure is, upon successful authentication Enterprise attaches known Id or User Identity to the User or client application. You *are* your Id, as far as Enterprise concerned.
If User is not authenticated (or may be not logged-on as yet), we assign this user the Id *Anonymous*, and may allow such a user to see some publicly available information.

Enterprise may define Roles and Privileges to manage the Access Control to valuable Enterprise resources and applications.
Every User Id will have some Roles allocated to it and some Privileges granted.

Authorization is the process of granting Access Privileges to the user (through Roles and User Id), and managing the Security and Access Control to protected resources and applications according to these Privileges.

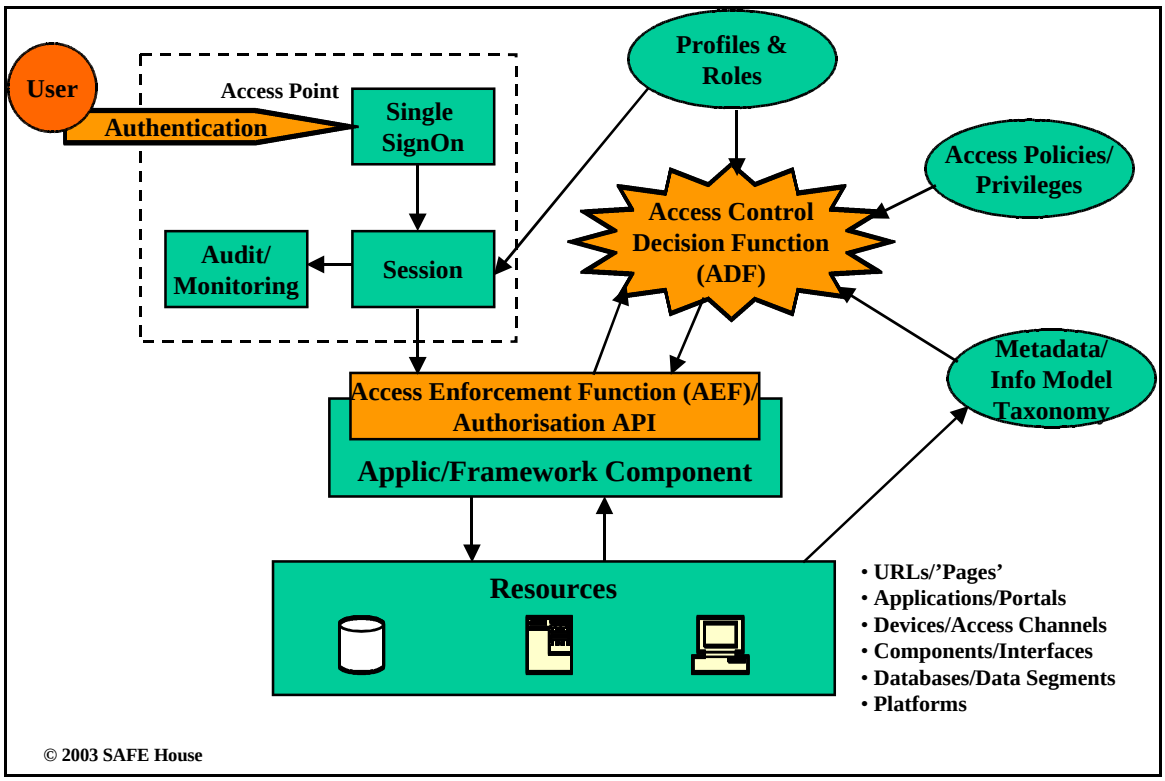

Figure 2.6 Access Control Reference Model

In a distributed system, control and data flow between the client process and the target resource transcends geographically remote locations, network devices, protocols, and software products.

Enterprise Architect has number of choices on where Access Control Decision Function (ADF) and Access Enforcement Function (AEF) should be applied along the way from the client to the protected resource. Security and Access Control products may have made this choice for you. Still, it is important to understand the rudimentary basics of the Access Control procedure in a single distributed access transaction.

As Figure 2.7 shows, Access Decision related to authorisation and privileges of the Client in conducting the requested transaction with Target/Resource can be performed in the business logic of the Client process or the Target process itself, or in their context environment, on either end of the wire (e.g., through the use of ORB interceptors or other components in the encompassing framework).

Things can get complicated quickly, as we consider complex access transactions when Target needs to perform further operation. We may have the chain of basic access control transactions, with Target becoming the Client for the next transaction, and so on.
For instance, if Target performs the database access on behalf of User in the Client process, who is the Principal then, and what are his Privileges for the database access transaction?

Transparently to the User in the initial Client process, overall framework will chain each leg in the complex access control transaction by the Delegation of Principal's Identity and Privileges to the next

transaction. Delegation of Identity and Privileges does not necessarily mean passing all access rights straight through, with the Identity and Privileges of the original Principal.

In our example with the further database access from the target process, database engines do not get User Id of the original Client. As a common practice, target process usually establishes connection with the database under some generic User Id, probably even not unique to our Target process (which is a Client now, as far as the database engine concerned). Such use of the database security features is another way of saying that client processes do not rely on database security features in enforcing fine granular access control for Users. Database APIs, like JDBC, still require passing the User Id and the password as parameters for connecting to the database, but this is becoming a nuisance, and does not perform the original access control function.
Single database connection from the Target to the database engine will perform many database transactions, on behalf of many different original Client processes, or Users.
Clearly, in this case, Target process should accept responsibility for the security and privacy of the original User in the database access. Incorrect business logic for the database access in the Target may, potentially, expose sensitive data to the unauthorised User.

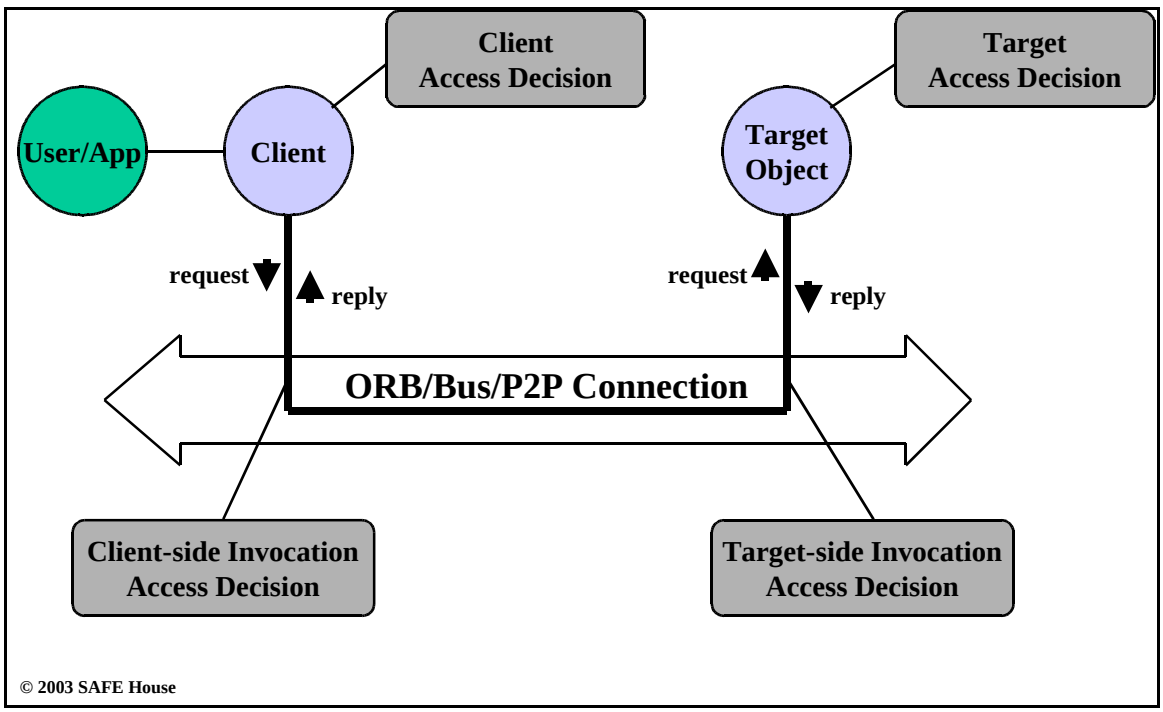

Figure 2.7 Access Control Model For Distributed Object Systems

Some of the more influential international security standards are:

- ***Security Framework in Open Systems, ISO/IEC 10181:1993, or X800 series.*** Defines data elements and operations for security services, but not protocols. Consists of:
  - ***Part 1, ISO/IEC 10181-1:1996.*** Overview
  - ***Part 2, ISO/IEC 10181-2:1996.*** Authentication Framework
  - ***Part 3, ISO/IEC 10181-3:1996.*** Access Control Framework
  - ***Part 4, ISO/IEC 10181-4:1997.*** Non-repudiation Framework
  - ***Part 5, ISO/IEC 10181-5:1996.*** Confidentiality Framework
  - ***Part 6, ISO/IEC 10181-6:1996.*** Integrity Framework
  - ***Part 7, ISO/IEC 10181-7:1996.*** Security Audit and Alarms Framework
- ***OSI Security Architecture, ISO 7498-2:1989.*** Defines security services and associated mechanisms in the OSI reference model

- ***Common Criteria (CC) for IT Security Evaluation, ISO/IEC 15408:1999.*** Defines the assurance criteria against which a system or a product will be evaluated. Criteria are grouped into rates from E1 to E6, and allow evaluating the confidence in the correctness of security functions. Consists of:
  - ***ISO/IEC 15408-1:1999.*** Introduction and General Model
  - ***ISO/IEC 15408-2:1999.*** Security Functional Requirements
  - ***ISO/IEC 15408-3:1999.*** Security Assurance Requirements
- ***Data Elements and Service Definitions, Security in Open Systems, ECMA 138:1989.*** Defines services and protocols for security in open systems. Provides a public key-based specification for distributed authentication
- ***US DoD Trusted Computer System Evaluation Criteria (the "Orange Book"), DoD Directive 5200.28-STD, 1985.*** Defines the criteria against which a multi-user operating system will be evaluated. Criteria are organized into classes from D to A1 according to the degree of trust that can be assigned to a computer system:
  - ***D*** – no security
  - ***C1*** – discretionary security protection. This class provides separation of users and data. It incorporates some form of credible controls capable of enforcing access limitations in an individual basis
  - ***C2*** – controlled access protection. Provides more fine-grained discretionary access control than C1. Makes users individually accountable for their actions through login procedures, auditing and resource isolation
  - ***B1*** – labeled security protection. Data labeling and mandatory access control over subjects and objects
  - ***B2*** – structured protection. Access control enforcement extended to all subjects and objects. Authentication mechanisms are strengthened. Stringent configuration management controls are imposed
  - ***B3*** – security domains. Reference monitor mediates all accesses of subjects to objects, and is tamper-proof and auditable
  - ***A1*** – verified design. Formal design specification and verification
- ***Trusted Database Management System, NCSC-TG-021, 1991.*** Interpretation of the "Orange Book" for the databases
- ***Trusted Network Implementation, NCSC-TG-005, 1988.*** Interpretation of the "Orange Book" for the networks
- ***Single Sign-On, A one-Time Password System, RFC 2289.***
- ***Directory Authentication Framework, ISO/IEC 9594-8:1995 or CCITT X.509.*** Describes two levels of authentication – simple authentication using a password, and strong authentication involving credentials formed using public-key cryptographic system
- ***IPsec, Security Architecture for the Internet Protocol, RFC 2401.*** Describes interoperable, high quality, cryptographical security services at the IP layer
- ***Digital Signature Standard, NIST FIPS 186-2.*** Specifies algorithms for generating digital signatures, including Digital Signature Algorithm (DSA)
- ***Secure Socket Layer (SSL), version 3.0, Netscape 1996.*** Provides communication privacy over the Internet by preventing eavesdropping, tampering, or message forgery
- ***Standard for Interoperable LAN Security, IEEE P802.10:1998.***
- ***Generic Security Service Application Program Interface (GSS-API), IETF RFC 2078, 2743.*** Defines high level API independent of cryptographic algorithms
- ***Generic Cryptographic Service application Program Interface (GCS-API), 1996.*** Defines API from the Open Group consortium for integrating the cryptographic functionality into the application
- ***Cryptoki, PKCS#11, RSA 1999.*** Part of the RSA's Public Key Cryptography Standards (PKCS). Describes access to personal cryptographic tokens
- ***Secure/Multipurpose Internet Mail Extensions (S/MIME), RFC 2632, 2633.*** Describes method for sending and receiving secure MIME messages using X.509 Public Key Infrastructure Certificates (PKIX)

Privacy - 'say no more…'

Security and Privacy are closely related, but not the same. Computer system may grant you a privilege to access some information, but these data themselves (or the way they are presented) may violate someone legal privacy rights or just hurt someone's feelings.

Security, Identity Management and Access Control represent the first bastion in defending the person's privacy.

Global reach of the Internet, combined with cultural and social differences in various countries and communities, creates a potent mix that may threaten (or, be perceived as threat to) the privacy of the individual or community. Opposite is also true – blocking the free flow and exchange of information (like chat, email, or access to the website), beyond reasonable security imperatives, may infringe on a basic human right of privacy.
Privacy breach may happen even without the malign intent, due to insensitivity or misunderstanding.

Single Sign-On (SSO)

How many User Ids and Passwords you have to remember to find your way around computers? And what about your PINs on your credit cards and mobile phone?
Would not it be great if you are reliably authenticated just once and could get on with your job without such a nuisance?

Ability to logon into Enterprise IT solution and to navigate from one of its application to another without being challenged with logon screen again, called Single Sign-On (SSO)
Security has its price. However, it is possible, and indeed desirable, to implement SSO in a controlled environment like the Enterprise with many separate applications, systems and platforms.

Identity Management spills from the arena of the particular Enterprise, to customers and business partners in the B2B scenario, to the society at large, where all aspects of our social and business life are becoming more and more reliant on services we receive over the 'web'.

In order to receive the personalised services, we have to start by revealing who we are, and to allow tracking our behaviour from transaction to transaction, from application to application, from service to service.
Ask yourself, how many changes you have to make in billing, banking, and other procedures, when you change address or the last name? You wish that you could do it once, in one place.
On other hand, you may not want to mix your public profile data and activities with your very private ones, like confidential medical records, or your personal social life.
Convenience comes into direct conflict with security and privacy, aggravated by complex technical, legal, and social chalenges.

SSO is just one, albeit very important and visible, manifestation of the more general issue of Identity Management in the access control.
In order to maintain SSO, Enterprise should enforce the common User Identity across all applications and components. This may prove to be impractical in the large Enterprise, due to the complexity and broad scope of the customer databases, or the history of the evolution of various components. Beyond the single Enterprise, with the partners in B2B transactions, establishing the common User Identity becomes even more problematic.
Second best thing to the single common Identity is the ability to link identities of the same customer, or to provide the Federated Identity.

Ownership of the infrastructure for provisioning the Federated Identity of customers and partners to the business (and to the society in general) is in the core of the battle for supremacy in IT industry, and, possibly, in many technological, business, and social aspects of our life at large.

Notable efforts in the area of Federated Identity include Microsoft's .NET Passport and Liberty Alliance's Project Liberty.

Federated Identity

Enterprise Identity and Access Management systems strive to maintain orderly and manageable databases with valuable profile data on all players in the enterprise business transactions.
Whole core business revolves around satisfying their customers with offered services better, and getting more business with the same or new customers in the process.

Primary driving force in Identity Management of the customer database is the enterprise ability to enumerate every customer one by one, and assign to them some unique Id that would provide the key into the enterprise corporate information model, and easily identify them in every business transaction.
This is still the basic intent of the Identity Management.
However, modern enterprise shatters such a simplistic view by presenting the real-life challenges. Single namespace for the whole of the customer base is becoming all but a dream for the large enterprise.

Enterprise's size, diversity of customer base and types of business, global reach, lack of centralised control over the customer data in different parts of business or on the business partner's site, all hinder our attempts to keep Identity Management simple.
And, even if we could resolve technical issues of consolidating the identity namespace of all players in the business transactions, there are privacy and over sociological constraints that prevent us from creating and maintaining the single identity namespace.

For instance, it would be technically feasible to identify all individual customers by their unique Social Security Number, and consolidate all snippets of their profile data from various, seemingly unrelated, customer databases. Someone could be a car buyer in one customer database, and the broadband cable connection user in another, and a furniture designer in yet another database, and so on. Common identity key would greatly facilitate cross-references between all these databases.
Technical convenience comes into direct conflict with privacy and basic human rights. Traceability of various aspects of person's profile and business activities fuels the fears of Big Brother. Even if we resolve all technical and social issues of single identity in particular country or enterprise, there will likely be different identity namespaces in other countries or partner's enterprises.

Out of necessity, there comes the idea of the de-centralised Federated Identity, when we recognise the local control over the identity namespace, but establish cross-linkages of mutual trust between separate consumers and providers of customer identity and services.
Project Liberty [WWW, LibertyAlliance] is one example of establishing collaboration mechanisms for loosely coupled businesses through the use of Federated Identity.

Client and Server

In computing, notion of *Client* and *Server* (or *Supplier* of the service) relates to the roles that two entities play to each other in conducting conversation or transaction.
The same entity may be at the same time Client to some, and Server to other entities, or both. Client requests for some service, Server performs the service.

If you pull apart your business transaction, you should be able to identify who is Client and who is Server.

Distributed and Remote

Distributed and *Remote* – both concepts related to the geographical separation of communicating components, but are different.

If components or servers of the IT solution are physically deployed in different geographical locations – we say that this is a *distributed* solution. Our Enterprise Architecture can be spread across rooms, floors, different buildings, cities etc. etc.

We say that connection is *remote* when two interoperating components (peer-to-peer) are far from each other, or when user tries to access our application from far away.
Note the difference – Enterprise Architecture itself may be *distributed* or not, but user connects to some access point to this infrastructure *remotely*.

Real-time and Batch - 'I'll be back …'

Real-time refers to the processes in the computer system being able to respond to the governing business processes in timely fashion. It does not necessarily mean lightning response – just in time so that business process does not have to wait for it. So you see – notion of 'real-time' is pretty relative.

Batch refers to some bulky processes when we submit many smaller transactions together as one transaction. We say that batch processing can run 'off-line', i.e. not real-time. Business processes are prepared to wait for it or forced to work around the delays caused by batch.
Funny thing is, that business does not always need real-time, and batch processing may achieve much more efficient use of computing resources overall.

Online and Off-line

Online in a computer-speak means 'up and running' and ready to serve request from the client process – be it user at the terminal or some client program.
User's terminal may not be necessarily PC, by the way. And the client program may be the ATM software, expecting bank's server to be ready for the processing of withdrawal.

Offline means 'down' or unavailable to serve immediate requests from the client.
This is not necessarily bad or abnormal state of the server. We may take server or database *offline* to run some batch processing, backup or planned maintenance.

Note that if we are aiming for 24x7x365 availability, even downtime for the planned maintenance counts.
Salesperson will not accept 'planned maintenance' as an excuse for the downtime if he or she won't be able to check the credit reference of the buyer at the counter, before the large sale transaction on credit card is finalised.

Synchronous and Asynchronous

Synchronous means that some process follows the pace of the workflow in the business transaction. Also, if workflow is sequential, process consists of sequence of steps – every step waits for the previous step to finish. Or, we say that this process is single-threaded and its steps are *synchronous*.
For example, method or subroutine call is *synchronous*, because the calling program stops until method returns control and the execution (or control flow) of the calling program resumes.

Conversely, *Asynchronous* process implies that it starts the life of its own, in its own thread or execution environment. Start or end point of the process may be aligned (synchronised) with other processes by some business logic, external to the process itself.
For example, we can submit a scheduled report overnight (asynchronously to user's activities) and sent e-mail to the user to logon and to get the report online when it is ready. User does not have to wait for the report at the terminal. Waiting may not be an option at all, if we know beforehand that we have to produce a complex report and it will take time.

Sessions - 'where were we?'

Session is the context of Enterprise communication with the User from the moment when user logs on into system until user logs off. Generally, session context represent the context of business transactions that user conducts while logged on.

Session context keeps the state of business transaction from one step of user activity to another, and from one basic transaction in the conversation to another.
For instance, session context may keep User Id after user's successful logon, user's name that was retrieved once from the profile database, last page visited etc.

Keeping state of the conversation becomes even more important on the Internet due to the stateless nature of HTTP.
Peer-to-peer conversation over the Internet may consist of many basic steps represented by the HTTP request/response. Every HTTP request/response transaction proceeds as if there was no history of previous requests. That is unless the server goes into trouble of storing in the Session context the state of where the conversation left off last time.

Maintaining the Session context for every logged-on user may be a costly undertaking when many Internet users hit the website at the same time.
Also, in case of several load-balanced servers handling user requests, website must ensure that the context of HTTP Session is propagated, or otherwise is made available on every server. Common technique for sharing the Session context between Servers is the persistant storage of the Session in the database. Although, you may decide to store the Session context in the database even when there are no other servers, for the purpose of Session recovery in case of failure for instance (see *Persistant* and *Transient*), or for scalability.
Note that Server in this case may be a separate physical box, or separate process on the same computer.

In the case of the load-balancing configuration with several Servers and local Session context, we must guarantee that the next HTTP request in a Session will be directed to the same server, which maintains the Session context (session affinity).

Components and Business Object

Enterprise Architect is always in search of pre-fabricated parts that can be easily plugged-in into his or her solution. Also, architect wishes for these parts to be easily replaceable and separately manageable.

A complex enterprise solution consists of parts, or Components, Business Objects. We define Components so that we decrease complexity and cardinality of the system, and complexity of interfaces between parts.

Components may represent some part of the whole. Whole system represents the aggregation of components. We identify components through system de-composition.
Altenatively, components may determine the level of abstraction or specialisation in the system.

<<< … >>>