Published in Proc. ICSSEA 2007

Nine design patterns to be used during functional design of a service-oriented architecture

Thierry MOINEAU
Capgemini
Coeur Defense - 110 Esplanade du Gal de Gaulle
92931 La Defense Cedex
thierry.moineau@capgemini.com

Abstract: IT systems of most large enterprise were built gradually during the last decades as a collection of independent silo software applications. This resulted in data duplication, data inconsistencies, overly complexity and eventually bad quality for the users, the customers and the company. To solve these issues, it is often decided to re-organize the IT system according to a Service Oriented Architecture centered on shared information repositories. This decision, which is often made at technical level only, has in fact numerous consequences at functional level. Not taking them into account generally results in the failure of the SOA program and to what can be called a SOA chaos : the silo are recreated, the services invocations transform themselves into implicit strong coupling and the IT system is even more complex and less agile than before.
We present here nine patterns to be used during the functional design of a Service Oriented Architecture in order to avoid this SOA chaos. These patterns result from our experience in several large enterprise wide SOA programs.

Keywords: Service Oriented Architecture, SOA, design pattern

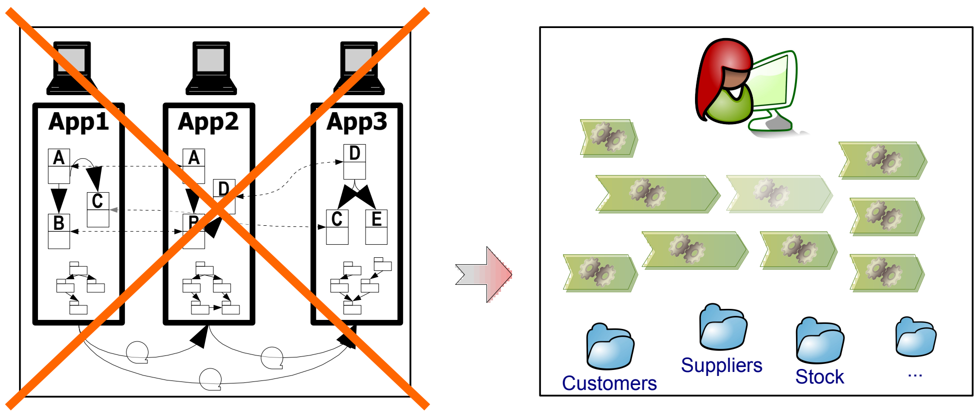
INTRODUCTION

The IT systems of most large enterprise were built gradually during the last decades as a collection of independent silo software applications, in which information are duplicated. This results in the 4 notorious mismatches:

- Application mismatch: data updates in one application are not are not applied in the other applications
- Identifiers mismatch: real life information is identified differently by various applications (e.g. a good is identified differently by the stock management system and by the order management system)
- Organization mismatch: even when they share application, each business unit has its own database and data are not shared (hence the same customer is known differently by different business units)
- Temporal mismatch: data synchronizations between applications (when they exist) take time (weeks or even months)

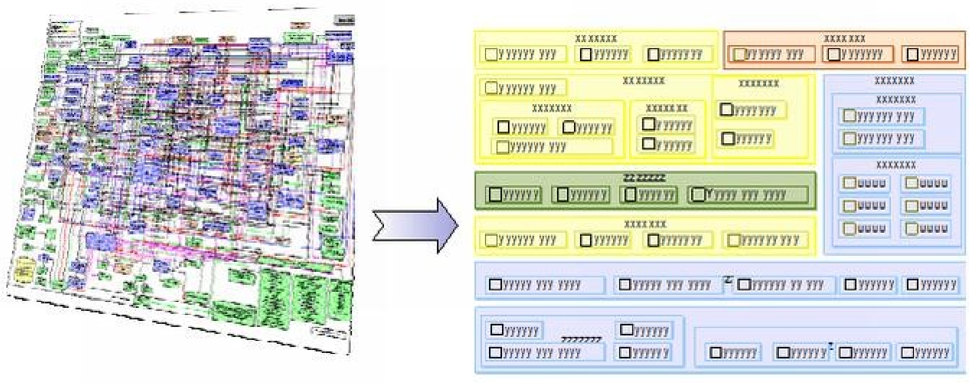
The consequences are multiple type-in, data inconsistencies, overly complexity and eventually poor quality for the users, the customers and the company.

To solve these issues, it is often decided to re-organize the IT system according to a Service Oriented Architecture centered on shared information repositories, which are used by all the IT applications.



This decision, which is often made at technical level only, has in fact numerous consequences at functional level. Not taking them into account generally results in the failure of the SOA program and to what can be called a SOA chaos : the silo are recreated, the services invocations transform themselves into implicit strong coupling and the IT system is even more complex and less agile than before. We present here nine patterns to be used during the functional design of a Service Oriented Architecture in order to avoid this SOA chaos. We don't claim any ownership on these patterns as we would rather consider them as common good sense – but our experience has demonstrated that they are worth being stated together.

Pattern 1: Modularity and encapsulation
The IT System is partitioned into sub-systems which are highly-cohesive and loosely coupled.



The rational beyond this pattern is double:

- Implementation complexity: it is impossible to build an enterprise wide IT system (hence large) into one piece – using the divide and conquer strategy, the system is split into smaller pieces easier to understand and to design.
- Maintainability: on a large scale IT system, it is mandatory to be able to modify one part of the system (new market needs, new regulatory requirements) without having to verify and validate the whole system.

Hence the IT system is partitioned into sub-systems which are highly-cohesive and loosely coupled:

- High cohesion: the data and processes inside one sub-system have a conceptual proximity
- Loosely coupling: a modification in a sub-system has minimal impact on the other sub-systems.

This structure of the IT system is purely based on functional aspects and does not take into any account technical aspect. Accordingly, we call these sub-systems Functional Components (FC). Functional Components are of 2 kinds:

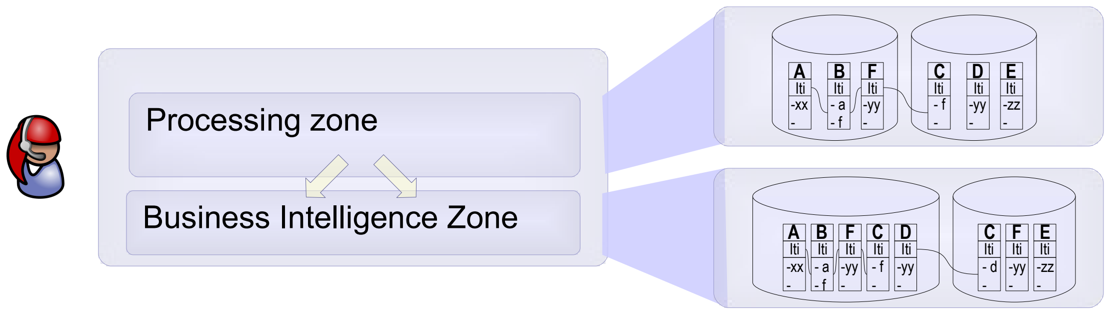
- Data Oriented Functional Components, which deliver services related to their data – they are also called Repositories
- Process Oriented Functional Components, which implement a set of business processes – they are also called Pilots.

In order to maximize maintainability of the IT system, it is critical to limit and control the impacts that a modification inside a functional service will have on other functional components. Hence Repositories must hide the internal implementation details and in particular the internal structure of their data:

> ***Rule 1a****: Repositories provide strongly data encapsulation and give access to their data only thru services which are explicitly described and published.*

Pattern 2: Unique management of information

Any information is managed by one unique component of the IT system. Multiple copies of the same information may exist for specific purpose (namely archiving or business intelligence), but these copies are not accessible neither by the repositories nor by the usual pilots.



The aim of this principle is to ensure integrity of up-to-date information. This principle is the basis, which enables to avoid double type-in and information mismatch.

It is important to note here the difference between information and data: a same data may correspond to different information. For instance the usual address of a customer and the delivery address for a specific order may be the same data but are different information – if the customer has ticked in the 'delivery at my usual address' box then this is the same information otherwise it is just the some data; this distinction is important, in particular if the customer has just submitted an address change request.

The second part of the principle recognizes the specificity of business intelligence activities:

- they are not part in a synchronous way of the usual activities of the enterprise,
- they need a specific organization of data to enable cross references among the data, and
- they do not need up-to-date information and use historical data (usually from previous months or years).

An important corollary of this pattern is the following:

> ***Rule 2a****: The data models of the repositories are disjoint.*

This, in turn, imposes the concept of Unique Internal Identifier:

> ***Rule 2b:*** *Each information in the IT system has a Unique Internal Identifier (UII) which obeys to the 3 basic properties:*
>
> *non-meaningful: it is impossible from the UII to guess anything regarding the related information*
> *non-reusable: the UII cannot be reused for another information*
> *non-mutable: the UII cannot be modified nor deleted*

Accordingly, a repository contains its own data and identifiers enabling to manage relationship with information belonging to other repositories. The unique internal identifiers are used for all the interaction between the functional components.

The *non-meaningfulness* property forbids including usable piece of information in the UII. For instance a customer identifier composed of its family name, its zip code and a sequence number does not fulfill this property. The need for *non-meaningful* identifiers is a consequence of pattern 2: meaningful identifiers duplicate the information and lead to data inconsistencies – for instance is the example below, a component could believe that the customer lives in a specific city, which may not be the case anymore and hence allocated the sales to a wrong department (we have all seen such examples, haven't we?).

The *non-reusability* property is essential to ensure the consistency of the enterprise information. Indeed, identifiers are stored in other repositories and reusing an identifier for another purpose than its initial purpose would just create havoc in the IT system, in particular in term of historical data (orders for Mr. Smith would be assigned to Ms Jones – imagine the consequences in case of a recall alert).

The *non-mutability* property is needed to ensure that information will be accessible in the future. Indeed, if the *Customer* repository decide to change the identifier of a customer, then it will not be possible anymore to relate the orders, in the *Order* repository, with this customer. *Non-destruction* of old identifier is essential to ensure traceability in the system.

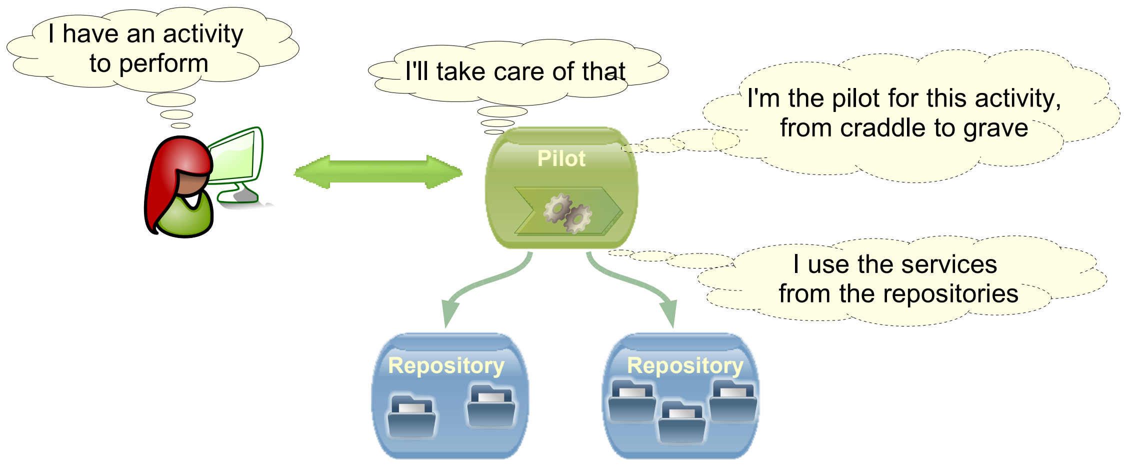
In order to avoid the above inconsistencies, one could imagine changing the customer identifier at each house moving. This would impose to be able to alert all the other repositories, either by maintaining in the repository a list of all the other repositories having information related to this customer, or by implementing broadcasting mechanisms – both mechanisms are very complex and would generate, at the level of an enterprise wide IT system, a huge number of messages. It is much simpler to have non-meaningful identifier and to request the appropriate repository to get up-to-date information.

It is important to distinguish here between internal identifiers used by the IT system and real life identifiers. Real life identifiers are usually ambiguous either because human beings need to remember them or for legal or cultural reasons (asking ones' customer for fingertips or DNA samples may not be acceptable for a commercial enterprise – and even for most government agencies).

Pattern 3: Unique management of activities

IT system usage is modeled as activities. An activity is a work unit, as seen by the end user, obeying to the 4 rules: unity of place, unity of time, unity of actors and unity of action.

Each activity is managed from cradle to grave by one Pilot, thru invocation of services provided by the repositories.


I have an activity to perform
I'll take care of that
I'm the pilot for this activity, from craddle to grave
Pilot
I use the services from the repositories
Repository
Repository

The first part of the pattern aims at normalizing the concepts used to model the IT system from a business point of view, in order to avoid a disparity which would be detrimental to the IT people and to the end users (ergonomics).

The second part of the pattern is the symmetric of the pattern 2 applied to processes and its rationale is similar: the need to have a modular IT system while avoiding strong coupling (note that the 4 unity rules already promote high cohesiveness). Indeed splitting the management of an activity among several pilots would require a hand over between pilots within an activity, which is seen by the end user as an homogeneous task. To avoid ergonomic and semantic mismatches, this imposes an in-deep mutual knowledge of the pilots, up to the basic interaction sequences, which leads to high coupling and its consequences (cost increase and agility decrease).

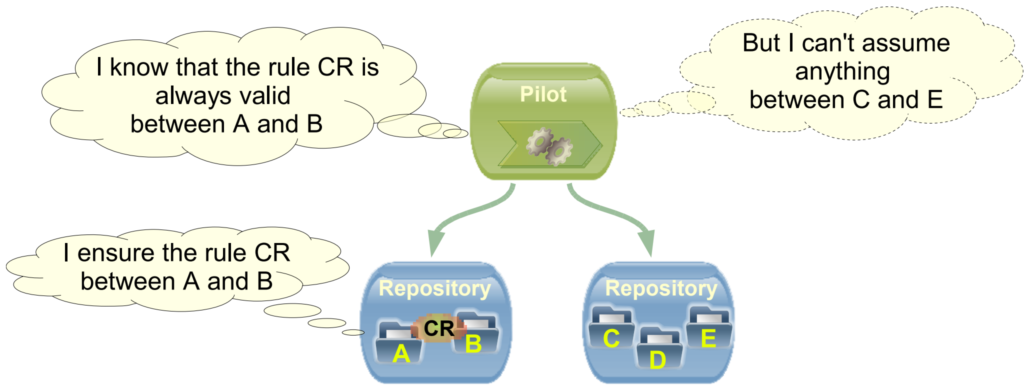
In addition, cascading hand-over from pilot to pilot results in overly complex management of non-nominal cases (in particular errors). Indeed, the experience proves that when an error occurs in the third pilot, then it is often needed to come back to the first pilot, which may have very limited understanding of the cause of the error and can't give much help to the user, except when a high coupling exists between the pilots – and we already know the drawbacks.

This pattern should not be understood as an exhortation to re-create a silo-based IT system: the pilot in charge of the activity shall of course use the services offered by the appropriate repositories.

Note that some processes can't be modeled according to the 4 unity rules, basically workflow oriented processes. Each of these processes also has to be managed by one pilot, whose duty is to manage the transition among activities without having an in-deep knowledge of the activities. Albeit such modeling requires a specific state of mind, it avoids high coupling and proves very valuable in the long term.

Pattern 4: Responsibility for functional consistency

Each information is owned by one Repository who is the only one in charge of guaranteeing its functional consistency.


I know that the rule CR is always valid between A and B
Pilot
But I can't assume anything between C and E
I ensure the rule CR between A and B
Repository
CR
A
B
Repository
C
D
E

Albeit this pattern may seem a simple consequence of pattern 2, it is in fact a change of paradigm which proves more subtle than it seems. Indeed, in a silo based IT system, the application manages all its data and knows everything regarding its history and the processing already applied to it. Hence the application designers may (and usually do) take advantage of this knowledge, which we call hidden interdependencies. When dealing at the level of an application this may be manageable and even source of economies. When dealing at the level of an enterprise-wide IT system, the risk to forget such hidden dependencies is huge and leads to significant cost and duration increase during the maintenance of the system (up to the point than some enterprise don't dare changing some parts of the system because they are not able anymore to predict impacts of modifications). Hence it is necessary to explicit these hidden dependencies and to store somewhere their consequences; the best place for that is a repository. Accordingly, repositories are in charge of the functional consistency of information.

Consequences of this pattern are numerous.

> ***Rule 4a**: No process may suppose, even on a temporary basis, that a consistency rule will be valid if this rule is not explicitly enforced by a repository.*

Indeed, as information are accessible by several processes, there is no guarantee that all processes will enforce this consistency rule, rule that they may not even be aware of and hence that they may break in an indirect way. If the rule is needed to ensure the integrity of the process or of the entire IT system, then this rule must be explicitly stated and enforced by a repository.

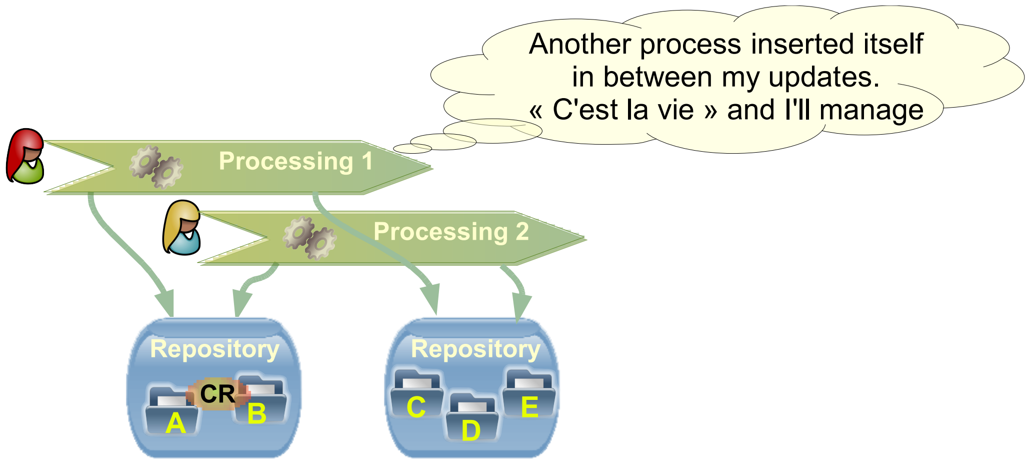
It is important to find the right balance between too few rules in the repositories, which may result in lack of transversal consistency, and too many rules, which may hinder the agility of the IT system. A rule of thumb is to include only rules which are fundamental basis of the business domain and to avoid rules which do not take into account the dynamics of the real world (e.g. a rule like "state B is forbidden if state A did happen before" may create complex side effects when forgetting that state A may result from an erroneous data entry).

> ***Rule 4b**: No process may assume anything on the history of an information, except when provided by a repository.*

This rule is very similar to the previous one and has similar rationale. One essential aspect of this rule is related to information validation: if a process is not authorized to assume any history on an information, how can it be confident in the validity of this information? The answer is that the fact whether an information has been validated or not is stored in the repository – which process has validated the information is of no interest for the processes using the data : these processes just need to know that the information has been validated (of course the enterprise architects have to decide which process will perform the validation).

Note that the 2 above rules apply to all pilots, even to the one which has created the information in the first place. This may seem obvious but our experience proves that this is not natural for IT designers used to silo-based IT systems.

Pattern 5: Unsynchronized pilots
Pilots may not assume that they will all be synchronized, in particular regarding information updates.


Another process inserted itself in between my updates. « C'est la vie » and I'll manage
Processing 1
Processing 2
Repository
CR
A
B
Repository
C
D
E

The rationale of this pattern is the fact that it is very complex (or even impossible) to synchronize all the processes within an enterprise wide IT system, without introducing high coupling. Indeed the various processes are initiated by many actors who have different time constraints in the real life – time constraints which are often outside the control of the enterprise (legal constraints or constraints imposed by external actors, like customers or partners).

For instance, a customer may have sent an order asking for the delivery to be made at his usual address. It may take some days for the enterprise to prepare the goods. In the mean time, the customer wants to change his usual address. How should the IT system behave? Should it send the goods to the usual address at order time? Should it send the goods at the usual address at shipment time? Should it forbid the address change because there is a pending order? Most IT system uses the third solution (maybe not in this basic example but in more complex cases), which in fact introduce a strong coupling between the processes. A good solution is to accept the address change but to send the goods to the address specified at order time – because this is what the customer asked for (this has to be of course clearly stated in the order form and it is a good practice to also provide a way for the customer to change the delivery address of goods not yet shipped).

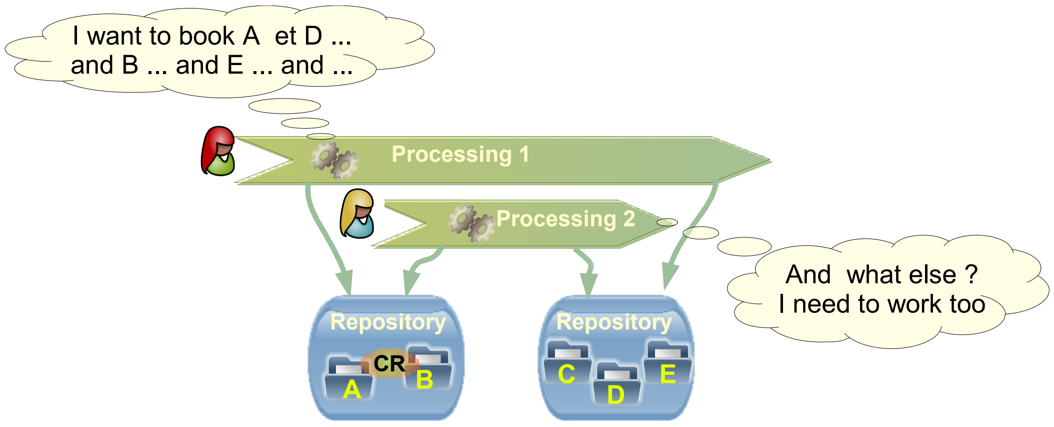
Implementing such solution imposes to be able to remember the usual address at the order time. This leads to the following rules.

> ***Rule 5a:*** *The repositories maintain the history of all information modification.*

> ***Rule 5b:*** *Information are not deleted but marked as obsolete.*

The later rule raises the issue of data proliferation and hence of archives and data purging. We claim that for most enterprises, this is a technical problem that technology progresses can handle. If data have to be physically cleaned up from the IT system (e.g. for legal reason), then a specific process has to be designed in order to perform the deletion which keeping the consistency of the IT system.

Pattern 6: Non-exclusivity of information
A pilot may not, even temporarily, lock the access to an information.


I want to book A et D ... and B ... and E ... and ...
Processing 1
Processing 2
And what else ? I need to work too
Repository
CR
A
B
Repository
C
D
E

This pattern is similar to the previous one, with an extra dimension related to data complexity.

Indeed, in silo based application, data concurrency control is usually managed thru locking mechanism: a process locks the data and nobody else can access nor modify the data until the process has unlocked it. Such a mechanism which was acceptable within one application can rapidly run out of control when applied at the level of an enterprise wide IT system.

The first issues to solve are: what needs to be locked and what is the semantics of the lock? Suppose for instance that the finance department wants to modify some customer data (eg. improve his credit rating). Should the system prevent all access to this customer data or should prevent only data modification? Should the system prevent only modification of the credit related data or should it prevent all processes related to the customer (including payments)?

The second set of questions is: how to implement such a locking mechanism? Obviously the lock has to be managed by the repository in charge of the data to be locked, otherwise this would introduce strong coupling among the pilots (i.e. all the pilots should know which other pilots could lock a data and ask them is the data is locked). But what to do if the data to lock is spread among several repositories, e.g. the customer repository and the order repository and the payment repository (changing the credit rating may have an impact on whether interests are applied or not for late payments)? We all know that all this easily leads to the notorious deadlock issue.

The next set of questions is: what to do if the lock is not released in a timely manner? If a lock is not released after some time, does it means that the activity which set it is longer than usual or does it mean that a bug occurred and that the lock will never be released? A usual way to solve the issue is to release all the locks every night, which is likely to lead to some data consistency issues, requiring some specific data cleaning processes. Another solution is to store the identifier of the user associated with the activity which set the lock and to send him/her a message asking to release the lock. The later case is quite complex to build (what if the user is on holiday? how to escalate to his/her supervisor? etc).

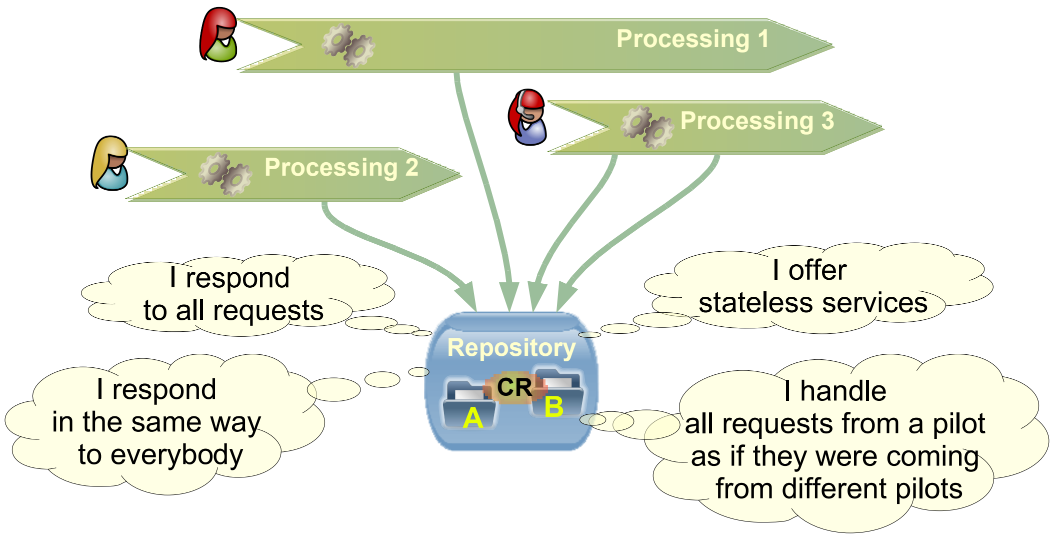
We see that what was easy and obvious at the level of a single application become very complex at the level of an enterprise wide IT system.

One may hence question the rationale of such locking mechanism in order to compare the value added with the cost generated. The rationales usually cited are either ensuring a functional consistency rule or enabling to temporarily break a functional consistency rule. In the first case, a pilot is trying to ensure a consistency rule that is not managed by a repository - we have already discussed the fact that this must be avoided (pattern 4). In the second case, a pilot is trying to store an intermediate state which is not acceptable by the repository in charge of the information - it is much easier to store such an intermediate state somewhere else and to update the repository when all the needed data is available.

Hence using usual locking mechanism does not provide a value which is in relation with the complexity it brings. It is much better, for enterprise wide IT systems, to use optimistic concurrency mechanisms.

Pattern 7: Stateless Services

Repositories provide access to their data thru stateless services, which always leave the repository in a coherent state.


Processing 1
Processing 3
Processing 2
I respond to all requests
I offer stateless services
Repository
CR
A
B
I respond in the same way to everybody
I handle all requests from a pilot as if they were coming from different pilots

The concept of stateless service is well known: a stateless service is a service that does not maintain state information between invocation and all the information needed is either passed in by the caller or stored in a persistent storage. The opposite mechanism, stateful services, requires much more resources (to maintain session data) and much more complex processing (to handle sessions, to recover broken sessions, to discard abandoned sessions, etc). It is hence worthwhile to question the value added by stateful services and to compare it with the added complexity.

The rationales given for stateful services are similar to the rationales given for locking mechanisms: to lock a set of data or to temporarily enable inconsistent data. We have already discussed these rationales for the pattern 6 and we have found out that, at the level of an enterprise wide IT system, the added value is not worthwhile the added cost.

The second part of the pattern is a consequence of the pattern 4 (functional consistency are ensured by repositories), of pattern 5 (pilots are not synchronized) and of pattern 6 (no locks). Indeed, if a service could leave the repository in an inconsistent state waiting for another service invocation to finalize a consistent state, then another pilot could activate a service in between and receive inconsistent information.

This pattern has an important consequence:

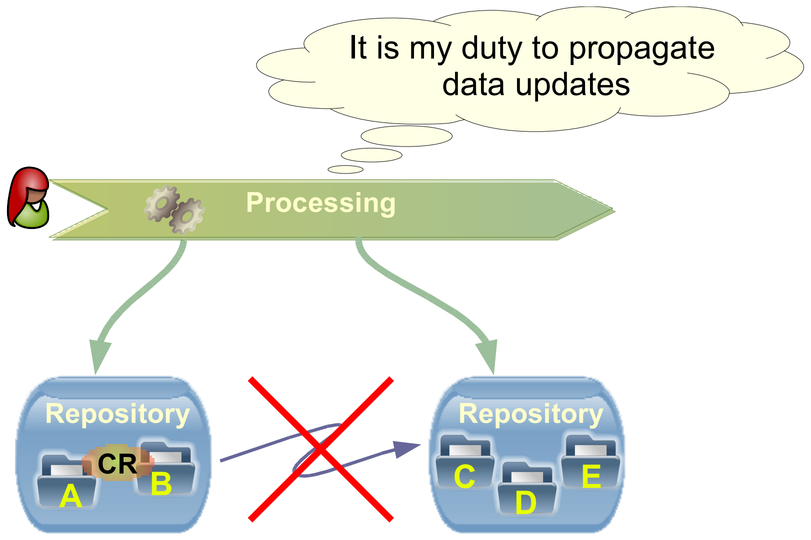
Rule 7a: *Services provided by a repository are atomic and of coarse granularity.*

Services are atomic in the sense that all the operations needed to provide the service are either performed or none is performed. In particular, if a service requires the update of several entities within the repository, then the repository must ensure that all entities are updated or that none is modified.

Services are of coarse granularity in the sense that fine-grained services would require several service invocations to ensure the consistency of the repository, hence breaking pattern 7.

Note that there is no contradiction between atomic and coarse-grained services: in an enterprise wide IT system based on shared repositories, services are 'large atoms'.

Pattern 8: Passive repositories
A repository is not in charge of alerting the other repositories when one of its information is modified.



Indeed, requiring a repository to alert the other repository would impose to defini distribution rules, in order for a repository to know which other repositories to alert. Indeed in an enterprise wide IT system, the amount of data update is much higher than in a silo based IT system (by several orders of magnitude: multiply by the number of applications and the number of geographies) and broadcasting to all repositories would consume a huge amount of network bandwidth, to distribute the alerts, and a huge amount of CPU, for each repository to filter the alerts relevant to him.

Such distribution rules would introduce a strong coupling between repositories, one way (the alerting repository has to know who to alert) or the other (in a publish-subscribe mode, the repositories have to know which information may have impact on their data). It is thus much simpler to apply pattern 3: the pilots are in charge of propagating data updates among repositories.

This pattern has the following consequence.

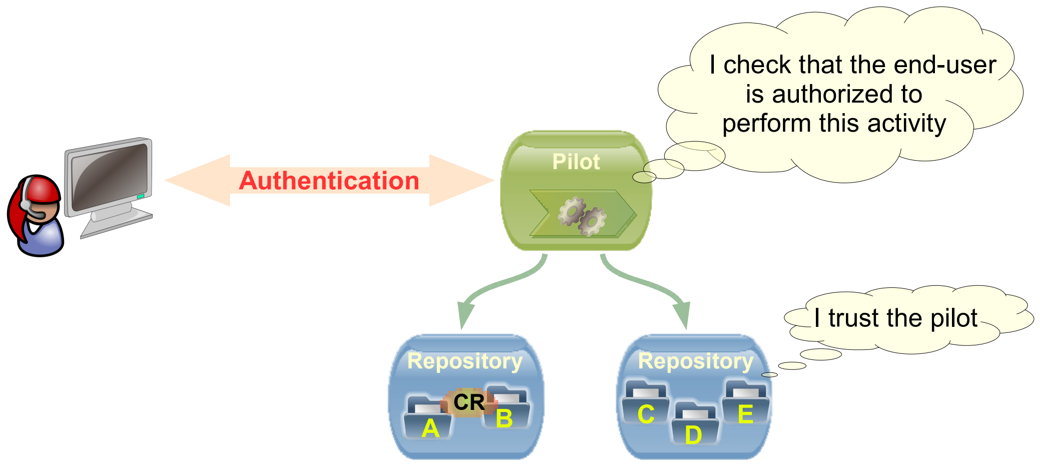
> ***Rule 8a:*** *Repositories must provide services enabling pilots to know which data have been updated within a given timeframe.*

Hence pilots are in charge of regularly requesting the repositories for data updates and of propagating these updates according to the business rules.

Note that implementing such services is quite easy thanks to rule 5a.

Pattern 9: Pilot based security

Security is based on a confidence infrastructure: end-user authorization is checked by the pilots and repositories only check credentials of the pilots.


I check that the end-user is authorized to perform this activity
Authentication
Pilot
I trust the pilot
Repository
CR
A
B
Repository
C
D
E

The rationale of this pattern is that access control has to be based on the activity to be performed and not on the information which is used. This enables the pilot to process and filter the information presented to the end-user and hence to provide processing based on information that the end-user is not authorized to visualize.

Acknowlegement : Many thanks to my colleagues and customers for all the discussions which eventually led to the ideas in this paper, in particular Jean-Marie Lapeyre and Denis Oddoux from the French Ministry of Finance. Many thanks to Jean-Paul Figer and Stéphane Girard from Capgemini for making the patterns REST and ATOM compliant.