White Paper
“Simplify Legacy Systems”

Abstract

This white-paper is made of five chapters and is inspired by five case studies conducted with CEISAR's sponsors (Axa, BNP Paribas, Michelin and Total).

Chapter 1 deals with measuring complexity and its impact on the business.
Chapter 2 proposes a method to conduct legacy simplification projects.
Chapter 3 proposes a process to actually implement legacy simplification, through Transformation or Replacement.
Chapter 4 answers frequently asked questions about legacy simplification.
Chapter 5 is a short abstract of the case studies that have been conducted to write this white-paper.

Chapters 1, 2 & 5 can be read by any IT executive (CIOs, Architects, Business people), while chapters 3 & 4 are targeted towards more technical people (Architects, Project leaders, Engineers) involved into legacy simplification.

Measuring Legacy Complexity and its Impact on Business

As they grow older, legacy systems become more complex with a negative impact on the business: **decreased agility**, **decreased quality of service**, **decreased ease of use**, **increased costs** and **increased risk** due to loss of knowledge.

To simplify your System, define your Enterprise System scope and simplification rationale with **offensive** (reduce time-to-market, offer new Functions…) and **defensive** (replace old technology…) **objectives**. Choose between different **execution strategies** (mandatory, self-standing, opportunistic or architecture renewal program).

To justify the need for simplification, **measure complexity** and measure how it impacts the business. Complexity comes from the Business System, Organization System and IT System.

- **Business System** complexity can be measured by the number of Business Entities, Business Processes and Business Functions and by volumes (data and process instances).
- **Organization System** complexity can be measured by the ratio of Organization Processes over Business Processes, Activities over Processes and Organization Functions over Business Functions, and by the number of actors.
- **IT System** complexity can be measured by the number of assets (hardware, software, middleware), number of interfaces, databases, tables, number of operations and development technologies, and by system and static code analysis.

Complexity can be reduced by a good architecture and **architecture level and quality should be measured** as well, but there is no established framework for that. Nevertheless CEISAR proposes tracks of what could be an architecture level evaluation: common Business Processes and Functions, Common Organization Functions, Common Software Services, Shared Block Cartography, Level of parameterized settings…

Proposing a Method to Simplify Legacy Systems

Once the framework has been set to justify the need for simplification, CEISAR proposes a method to simplify legacy systems:

- Describe the **old System**
  Old Enterprise System, Block Cartography and Technical architecture.
- Describe the **new System**
  New Enterprise System, new Block Cartography and Technical architecture.
  Explain **how they relate to the simplification** objectives.

- Decide **which blocks to keep/transform/replace**
  Based on Block interdependence, **refine the scope** to take into account propagation.
  Based on Block decomposition and Block evaluation (with respect to complexity and old/new Enterprise System gap analysis), choose between **keeping, transforming or replacing** each Block.
- Define simplification **Governance**
  For strategic governance, set-up a **small governing body with business people**, have **metrics** and dashboards, **train** IT people, enforce **architecture committees** and empower domain managers
  For project governance, keep the subject on the steering committees agenda, choose simplification **ordering** logically if possible (input blocks first), but favor **progressive phasing** with landing points and **intermediary business benefits**.

Proposing a Process to Implement Transformation and Replacement

To actually deliver simplification, CEISAR describes in more details two simplification strategies: the Transformation Strategy and the Replacement Strategy.

- First, **Isolate the blocks** that are within the scope from those which are outside. This should be done with **minimum impact on the blocks outside of the scope**, to avoid scope propagation.
- For **Transformation**:
  - Build **Data Access Services** and reengineer blocks to use them; migrate data
  - Isolate reusable Business Functions and expose them as **Software Services**
  - **Re-interface** blocks and remove dead parts
- For **Replacement**:
  - Replace block but maintain **ascending compatibility** in the interfaces (which implies maintaining some legacy constraints during a transition phase)
  - **Upgrade peripheral** blocks to use the new interfaces
  - **Renew peripheral** blocks themselves to achieve full simplification

  Choose between two execution approaches:
  - In the **Big-Bang approach**, migration is **easier and faster** but transition is more brutal and must be **thoroughly prepared**. It requires precise understanding of the functions of the legacy and proper testing with a **pilot phase** on a product or organization subset of the Enterprise.
  - In the **Parallel Execution**, the new System **runs in parallel and in sync** with the old System during a transition phase. It requires **great care in the architecture** and costs more but **reduces risk**, improves progressiveness and often allows earlier business benefits

Answering Frequently Asked Questions

Finally, the white-paper answers frequently asked questions:
- How do you extend an existing System and expose Services?
- How do you handle Entity Model changes?
- How do you modify Block Structures?
- How do you deal with data replication?
- Which middleware and tools can be useful?

1 Present Status and Objectives

Legacy Systems have been built progressively by successive add-ons.

As for any complex system, entropy increases with time, and comes a time when the Legacy Systems deserve to be simplified.

When the IT system is composed of tens of thousands of programs, consistency becomes a crucial problem: checking that a modification in a Block will not impact other Blocks can cost much more than the modification itself. This is why so many IT departments spend more than 70% of their energy just maintaining legacy systems.

Example: BNPP in France manages about 400.000 parts (piece of code, data description, JCL, …) in its IT System.

But simplification can be a huge investment. To be sure that it is really useful, we must:

- Define the global Enterprise System **scope**
- Measure **complexity** of the Enterprise System
- Measure how complexity **impacts** the business
- Measure how much it **costs** to simplify the System.
- Measure **ROI** before making a definitive decision, and define scope of Blocks to simplify

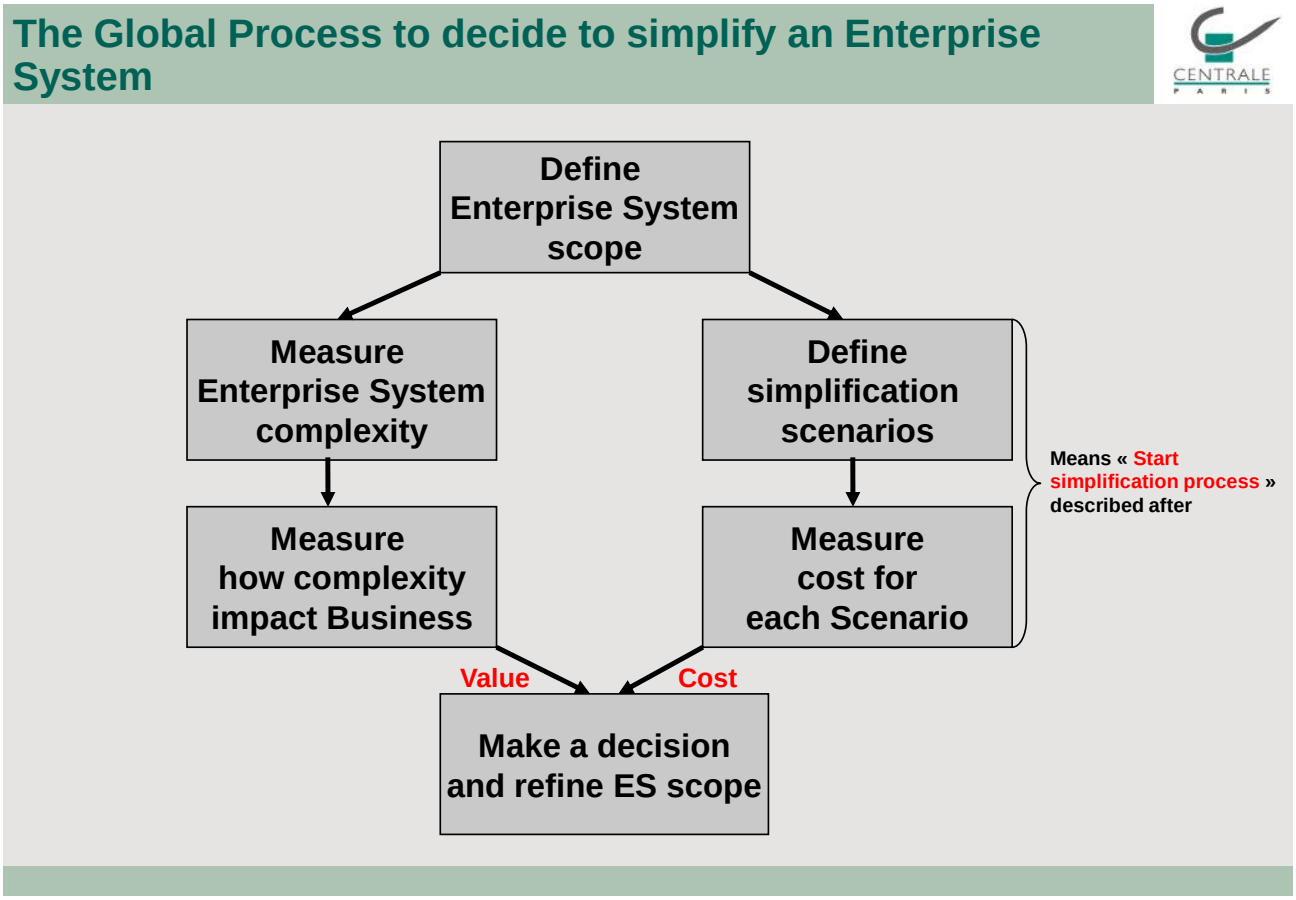


Figure 1: Global process to decide simplification

1.1 Define Enterprise System Scope

You first have to define what must be simplified.

Even if it is difficult to describe with precision what you will simplify, you must define a rough scope based on offensive or defensive objectives.

Offensive rationale:

- Decrease **time to market** for new products
- Offer **new functions** which cannot be delivered by present system, like vision of customer summary, more dynamic reporting functions, …
- Give **direct access** to partners, prospects and customers (e-business)
- Streamline and optimize Business **Processes**

Defensive rationale:

- Replace **old technology** which will not be maintained any more
- Maintain **knowledge**: if legacy systems experts progressively leave the company, a big documentation effort is required on the legacy if it is still maintainable, or a new documented software must replace the old one
- Increase IT **Service quality**: reliability, time to correct a defect, response time

Scope can be built with four major execution schemes (Figure 2)

Execution Strategies

Mandatory Discontinued technology, new regulations - No ROI issue (must be done) - Focus on mandatory objectives - Minimal changes. Avoid new functions	**Self Standing** Simplification can exhibit an ROI by itself - Remediate long-lasting system with rampant cost - Consolidate system instances - Phase-out expensive technology…
Opportunistic Piggyback on large business projects - Have clear vision of new Enterprise Architecture - Adapt to Business priorities (needed for ROI) - Share simplification objectives with Business	**Architecture Renewal** Strategic corporate program - Profoundly improve Enterprise processes and operations - No ROI or commitment issue - Scope and stakes complexity

Figure 2: Execution strategies

- **Mandatory execution** happens when a technology (hardware or software) is discontinued and **not supported** anymore or when new regulations cannot be supported by legacy evolutions.
  In this case, there is **no real ROI issues: the job must be done**, generally quickly (before a given deadline at least), and one looks at **minimal changes**.

  > *It is tempting to look at this opportunity to simplify further the system or to bring new user benefits. One should be cautious going this way, as the **added complexity can jeopardize the deadline for the mandatory changes**.*

- **Self-standing execution** happens when **simplification itself can exhibit an ROI**. It is rare when it concerns only IT simplification because savings are usually low and lead to slow ROI while they usually present business risks (bugs, downtime, change management).
  However, targeted and tool-assisted remediation can sometimes exhibit self standing benefits when an application or product line's lifecycle has been extended while maintenance costs are rampant.

- **Opportunistic execution** happens when a business project has **enough business benefits** for simplification efforts to piggyback on it. This is the best solution to find an ROI for a simplification project, and opportunities should consequently be seized.
  Opportunistic execution **should always be done with a vision**: the new architecture, migration and simplification principles should be clear and thoroughly analyzed beforehand (which means efforts must have been put on the subject independently of projects). The final simplification objective should be accepted and committed to by the business as well as IT. Otherwise, it is very unlikely that a sum of opportunistic simplifications leads to a real global simplification.

- **Architecture renewal execution** is the most ambitious. It consists in a **strategic corporate program** to profoundly change and modernize the Enterprise System in order to give a key competitive advantage to the company: support new service offerings (e-business) or processes that will be unique in the marketplace, profoundly reduce product time-to-market, profoundly improve operations and productivity (like in the Credit du Nord example).
  Architecture renewal is a strategic move with top executive support. ROI and commitment is not problem, but complexity arises from the width of the scope and the stakes and expectations of the program.

1.2 How to Measure Present Complexity?

Complexity must be measured to evaluate improvements.

A general Rule is that:

- Complexity increases if **too many "HAS"**: too many relations in all directions, no encapsulation, no reusability (no Block Model)
- Complexity increases if **few "IS"**: every product is specific (no Product model), every Process is specific (no Process Model)

To be more precise, complexity must be measured on the 3 dimensions (Figure 3):

- Complexity of **Business System**: like complexity of offered products
- Complexity of **Organization System**: like complexity of Organization Processes
- Complexity of **IT System**: like complexity of Software

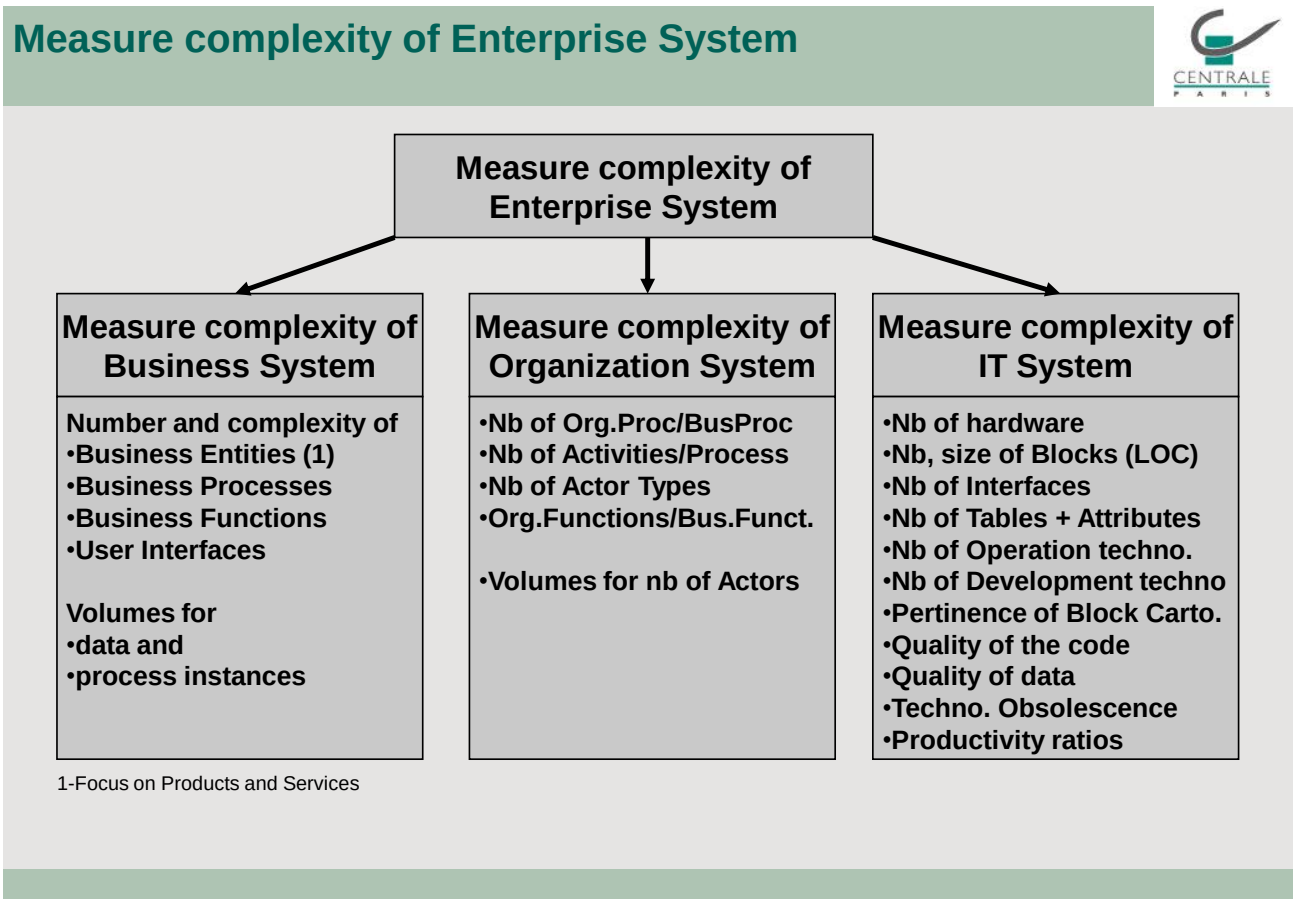


Figure 3: Measuring complexity of an Enterprise System

1.2.1 Complexity of the Business System

It is very difficult to measure complexity of offered User Functions. A popular method is Function Points evaluation based on inputs, outputs, inquiries, files and interfaces. But it is not based on Business Processes and Business Entities, which limits the obtained value.

There is a need for a common approach to measure complexity of offered Functions. This method should take into accounts:

- Number (of classes and not instances) and complexity of Business **Entities**
- Number (of classes and not instances) and complexity of Business **Processes**
- Number and complexity of **Functions** like computations (for pricing, commissions)
- Number and complexity of **User Interfaces** for updates and inquiries
- **Volumes** of Activities

The **difficulty** will come from:

- Entities, Processes and Functions must be **listed** and documented, which is not the case in most Enterprise Systems
- Which **weight** should be given to each item and each level of complexity in each item?
- If 2 Entities are **very similar** and inherit from the same parent Entity (like similar Products): do we count them for 1 Entity (the parent Entity), 2 Entities (the 2 sibling Entities) or 3 Entities (the parent + the 2 sibling entities)? Same thing if 2 Processes are very similar

1.2.2 Complexity of the Organization System

Complexity of the Organization System should be based upon:

- Number of Organization Processes **by** Business Process
- Number of **Activities** by Organization Process: a small number of Activities means that the work is done by a limited number of Actors

- Number of **Organization Functions** compared to Business Functions: if the proportion of Organization Functions is high, then it means that the Organization is not optimal
- Proportion of Activities done by Persons compared to **Automated Agents**: if many Activities are performed by Automated Agents, then it means that level of automation is high and Organization Complexity is low

1.2.3 Complexity of the IT System

The IT System should match the Business and Organization Systems, which means that IT complexity is **directly related** to Business and Organization System complexities. But part of IT complexity may also come from the way it was implemented, which involves architect talent.

Complexity of the IT System is measured by the number of **IT assets**:

- Number of **hardware** assets: clients, servers, networks…
- Number of **software** assets
  - Number and complexity of **Blocks** (or applications/packages/programs in the absence of Block cartography): complexity can be measured by the number of lines of codes
  - Number and complexity of **Interfaces**: complexity can be measured by the number of attributes and number of calling Blocks by Interface
- Number of **Data** assets
  - Number of Databases
  - Number and complexity of tables in the databases: number of attributes and relations
- Number of **Operation** technologies: they require as many skill sets
  - OS
  - DBMS
  - Middleware (EAI, ETL, applications servers, …)
  - Networks
  - Operating toolsets (scheduling, monitoring, back-up and recovery solutions)
- Number of **Development** technologies: they require as many skill sets
  - Development tools
  - Languages
  - Frameworks

And by **quality indicators**:

- Quality of Block Cartography and Structure (compliance with IT architecture principles: urbanism, tier separation…)
- Quality of Code (comments, rule violations, dangerous constructions, cylomatic complexity…)
- Quality of Data (respect of integrity rules, level of recurrent manual cleansing…)
- IT Productivity indicators (average cost for developing a Function, average lead time, average defect ratio…)

Methods (McCabe, Halstead, SEI…) and **tools** (See Chapter 4.8) have been conceived that help to automatically measure complexity, maintainability, and flexibility of a software system.

1.2.4 Measure Architecture Level

Architecture Level reflects what is **shared** in the Enterprise System.

If the number of shared pieces increases, then the total number of pieces decreases, which also **decreases global complexity**.

It would be useful to define a **synthetic measure** of the “architecture quality and maturity” of a legacy, which will influence the advised strategy (Figure 4).

Measure Architecture Level which decreases unnecessary complexity

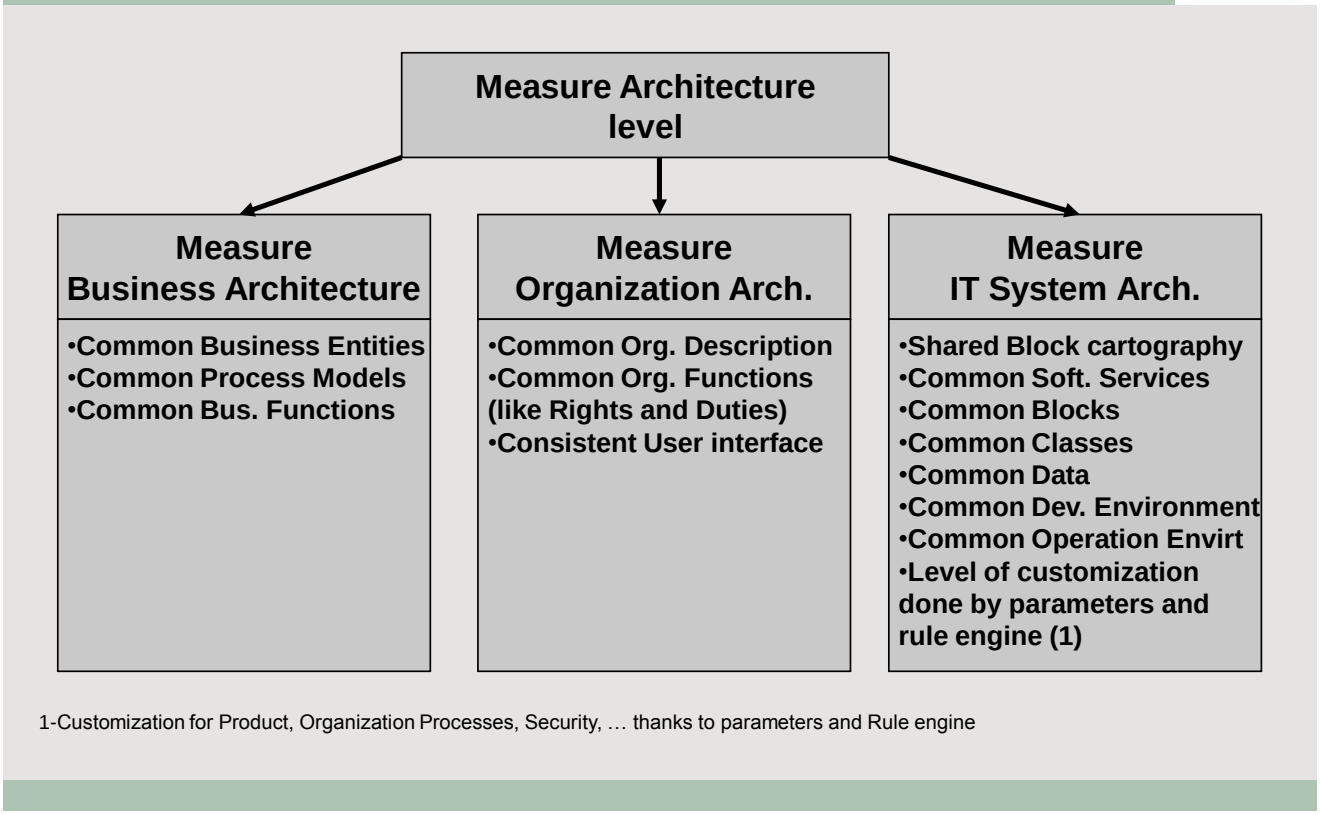


Figure 4: Measuring Architecture maturity

Sharing common **Software Services** is key. Some people talk of "**Component Reuse**" or "**SOA**", which is also about reuse.

Some examples of component reuse:

- Exchange Services between Blocks: synchronous or asynchronous (like accounting entries or decision data), updates or inquiries
- Data access to protect data evolutions from software evolutions
- Inputs
- Outputs
- Security
- Error reports
- Rule engine

1.3 How does Complexity Impact the Business?

Complexity leads to sub optimal IT systems, by driving up development and evolutions lead times, total cost of ownership and risk of failure (lower quality).

- **Decrease of Agility**
  - **Slow evolutions**: average time increases for all kind of standard modifications (from single attribute modification to launching new Products)
- **Increase of Costs**: not only increase of IT costs, but also increase of all other related costs
  - User costs: **productivity** is impacted by
    - User Interface inconsistencies
    - Recapturing information because of inconsistent IT systems
    - Error cost because of lack of controls or asynchronous controls

- IT Development costs:
  - If the System is too complex, most of the energy is spent just testing how modifications impact it. One criteria is % of man days spent on:
    - **Developing** the new functions: requirements and analysis, design, programming, individual tests, user documentation
    - **Integrating** the new functions into the IT system: interfaces, software integration, non regression tests
  - If several Development Environment exist, it implies specialized teams which cannot help each others in term of workload balance or sharing common components
- IT Operation costs: specialized operation systems means increase of costs for hardware, software, network and operation staff

- **Decrease of IT Service quality**
  - More complexity comes with less **reliability** and more time to identify where a defect comes from
  - It also comes with more difficulties to solve **performance** problems because it is harder to identify where the IT energy is spent
  - Enterprise System **administration** becomes more complex (Identification, Rights, User interface parameters, Products definition) because IT Systems are different
- **Decrease ease of Use**: if user Functions are more complex
  - No standard user interfaces means more difficulty for employees to move from one job to the other in the same Enterprise
  - Discontinuity because of lack of cross-blocks Processes
- Risk of **lack of knowledge**: an Enterprise with a complex Enterprise System depends on a small number of people who understand this complexity. As it takes time to train new employees, dependency may become dangerous

1.4 Why are IT Architectures so Complex?

1.4.1 On the Demand Side

- **Scope** of IT has exploded over the last 30 years for at least two reasons:
  - **Internal productivity**: scope of automated activities has grown exponentially to cover most functions in the organization
  - **Business opportunities**: information technology can provide the foundation for new products and services (e-economy)
- Companies evolve and increase their business process maturity: benchmarking and sharing best practices impact the way organizations operate. Continuous improvement is key to competitiveness. This generates demand for **change** in existing IT systems and creates a **need for new systems**.
  - For instance, after developing operational systems, companies develop monitoring and decision support tools
  - Accounting systems: several generations of systems, from custom developments to standard packages (ERP)
- Companies adapt their strategies to optimize their competitiveness: vertical or horizontal **integration and diversification** (through mergers and acquisitions), or on the contrary concentrating on their core businesses which generates most value (through business process outsourcing). Moreover, companies tend to open their boundaries to external partners (**extended** Enterprises). IT landscapes are widely impacted by these major changes in the configuration of the organization.
- Increasing **regulatory constraints**: each industry has its own set of regulations which have greatly developed in the recent years (example of banking (Basel II), pharma (FDA CFR 21 Part 11), aerospace, automobile, …) + cross industry regulations (Sarbanes-Oxley, …). These

regulations directly or indirectly prescribe the way to develop and manage IT solutions used to support critical business activities.

1.4.2 On the Supply Side

- **Moore's law**: Information Technology is a young industry. IT is still evolving exponentially after 30 years of existence. The pace of change is high and existing solutions are quickly going out of date/**obsolescing**. Since we cannot replace all our existing systems each time a new generation of technology is developed, this introduces a great variety in the portfolio of existing solutions.
    - Cobol
    - 3270 applications
    - Client server
    - ERP
    - Internet
- IT expertise:
    - Skilled resources have developed rapidly but remain scarce compared to the demand for new IT systems. This results in the recruitment of **under-skilled resource** on some projects which deliver sub-optimal solutions (In some cases, end users have even thought they were able to develop their own system by themselves, which has been made possible by the personal computer). Errors arise all along the life cycle of the solution: not choosing the right functionality in the first place, wrong analysis, wrong conception, low quality of code, leading to poor maintainability, high administration and operation costs, high failure risk, low scalability, …
    - IT development and operating processes have kept evolving for the last 30 years. **Best practices** are slowly emerging as this industry matures. Proprietary frameworks leave way to open standards (like IT security standards ISO 27000, CMMI, UP, ITIL or COBIT). Hence, each generation of systems might be built and operated in a different way.
- The combined effect of the great variety of technologies and the lack of skilled resources: It becomes increasingly difficult to maintain knowledge on all technologies. Suppliers (hardware and software vendors, service and consulting firms) cannot maintain their investment on older technology. This pushes all customers to turn to new technologies when implementing new systems. The IT landscape grows in diversity. It also creates a need for technical migration projects (with no direct business benefit), which mobilize budget and resource on initiative with sub-optimal return on investment

2 How to Get There?

As the topic is complex, let's start with an example to identify some elements of the simplification approach.

2.1 An Example: Credit du Nord Simplification

Credit du Nord is a French retail Bank which was losing money and required huge productivity gains.

The global objective was to:

- Deliver high service quality to customer
- Increase time-to-market
- Decrease internal staff: from 11,500 to 7,500 people
  - In centralized Back Office: from 6,000 employees to 2,000
  - In Branches: remain steady with 5,500 employees

The solution was to keep the Business System (same products, same customers, same Business Processes), but deliver a new Organization System and new IT System which allowed

- to simplify Processes and help Users by standardisation and ease of use
- to transfer workload from Back Office Centres to branches close to the customer.

To simplify the Organization Processes:

- few activities for each Process, which means a minimum number of Actors for each Process
- all checks on line,
- user interface consistency,
- workflow facility

To simplify the IT System:

- one mainframe instead of 4
- huge reuse of Software Services: 50% reuse rate

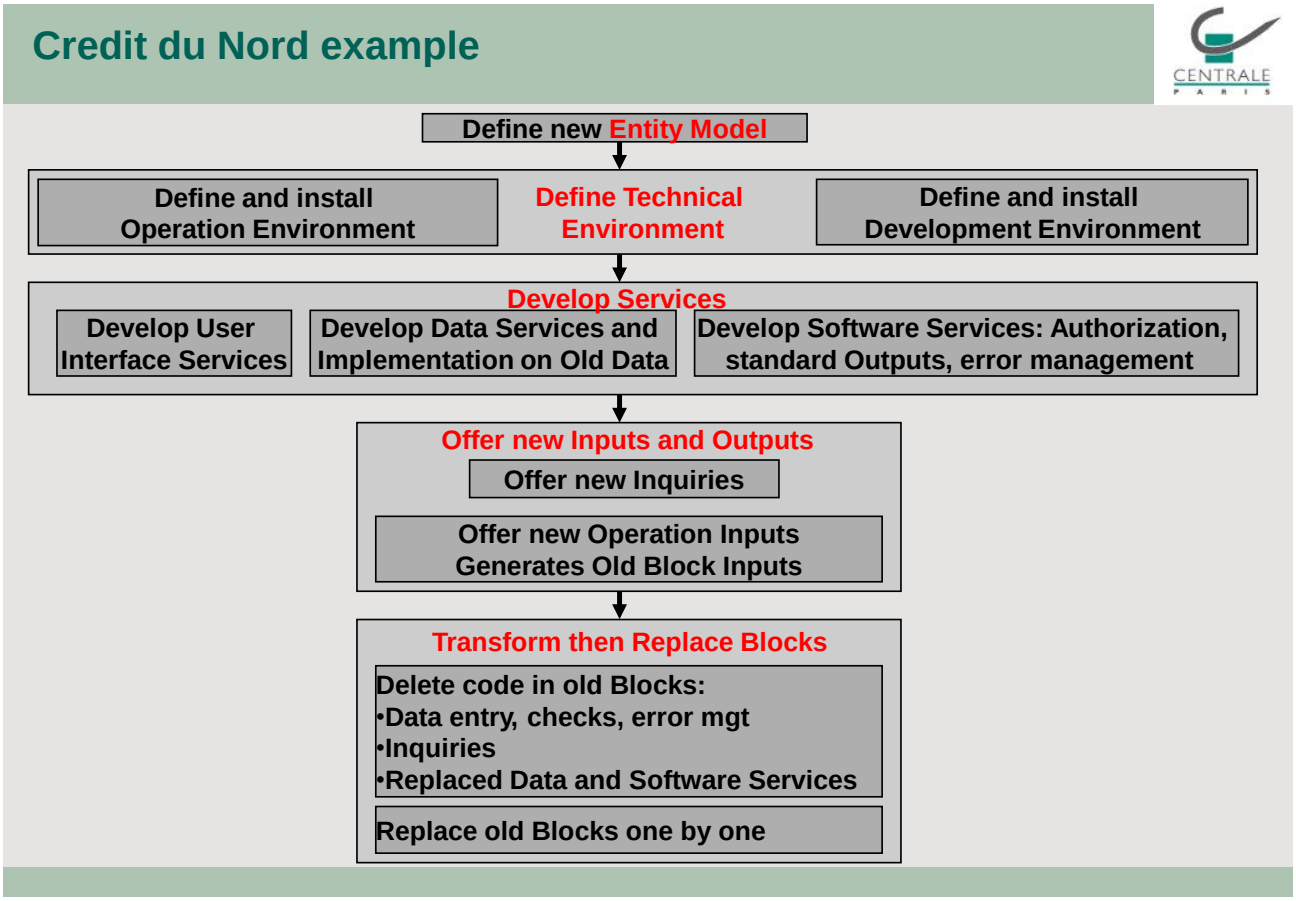


Figure 5: Credit du Nord example

From this example we notice that:

- Simplification is a progressive process
- It requires a global view of the System: old System and new System
- It requires shared Services: Data Services, Software Services, Input Services and Output Services
- It is a mix of progressive Transformations followed by Replacement when each transformed Block is well structured
- It involves a choice of technologies

2.2 CEISAR Simplification Process

But is there a global approach which could help us find the good path and take the good decisions?

We have tried to develop a first version of this approach (Figure 6).

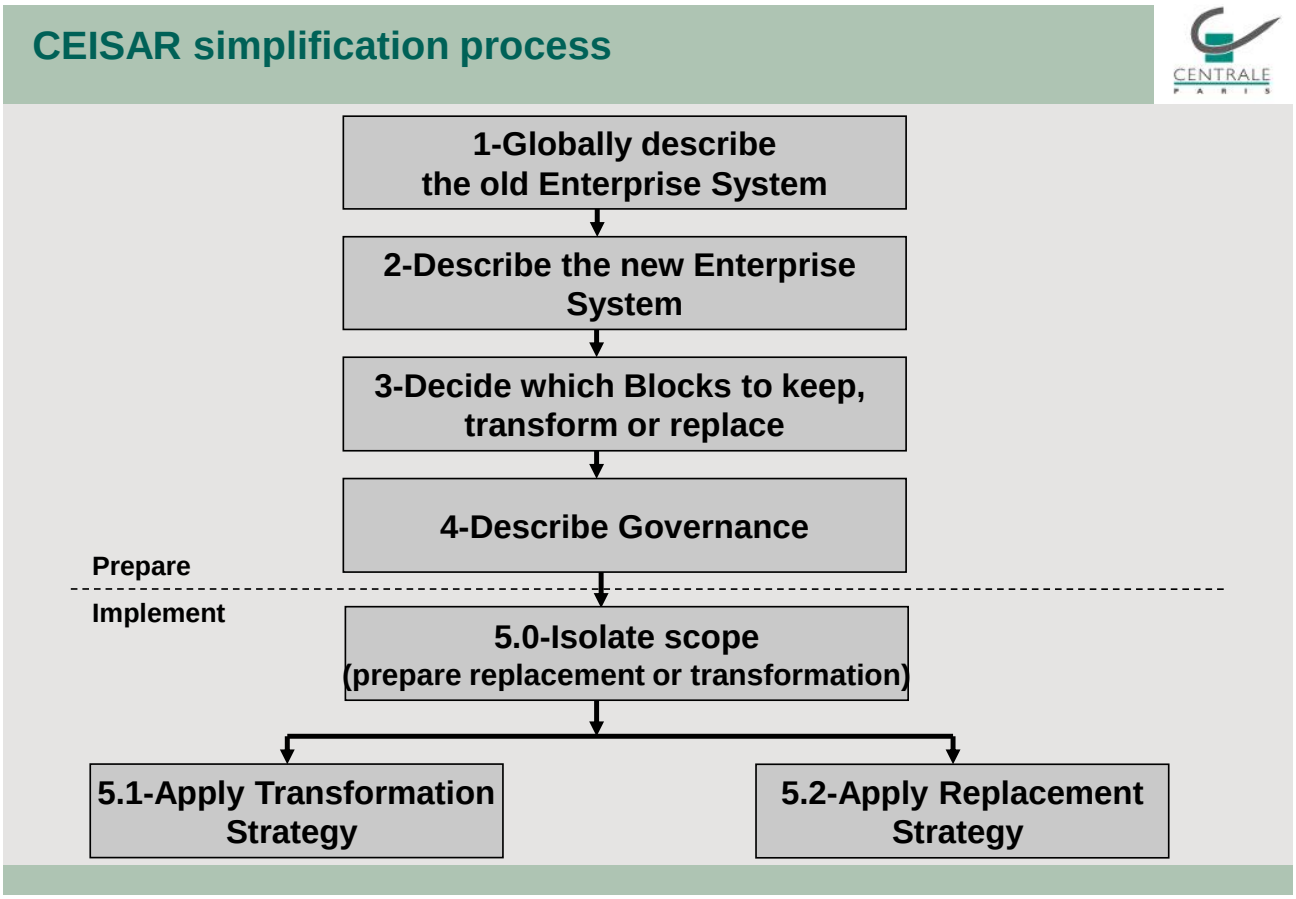


Figure 6: CEISAR simplification process

2.3 Globally Describe the Old System which Must be Simplified

Whatever the objectives you are pursuing, whether you want to value the old IT System and keep it (modernized) or whether you want to phase it out, you will have to understand it (Figure 7).

Documentation of an old system is not easy to do because it usually has not been updated, or because people who know it are not here anymore. Tools can help you understand your system (See Chapter 4.8).

The description of the old System must be reduced to 2 topics:

- "**Which IT Services are offered to IT Users**" requires describing the Business System and the Organization System.
- Which **interfaces** with external IT Systems requires identifying exchanges like Data Services or Software Services. At this stage, it is not required to detail the decomposition of the legacy System into Blocks.

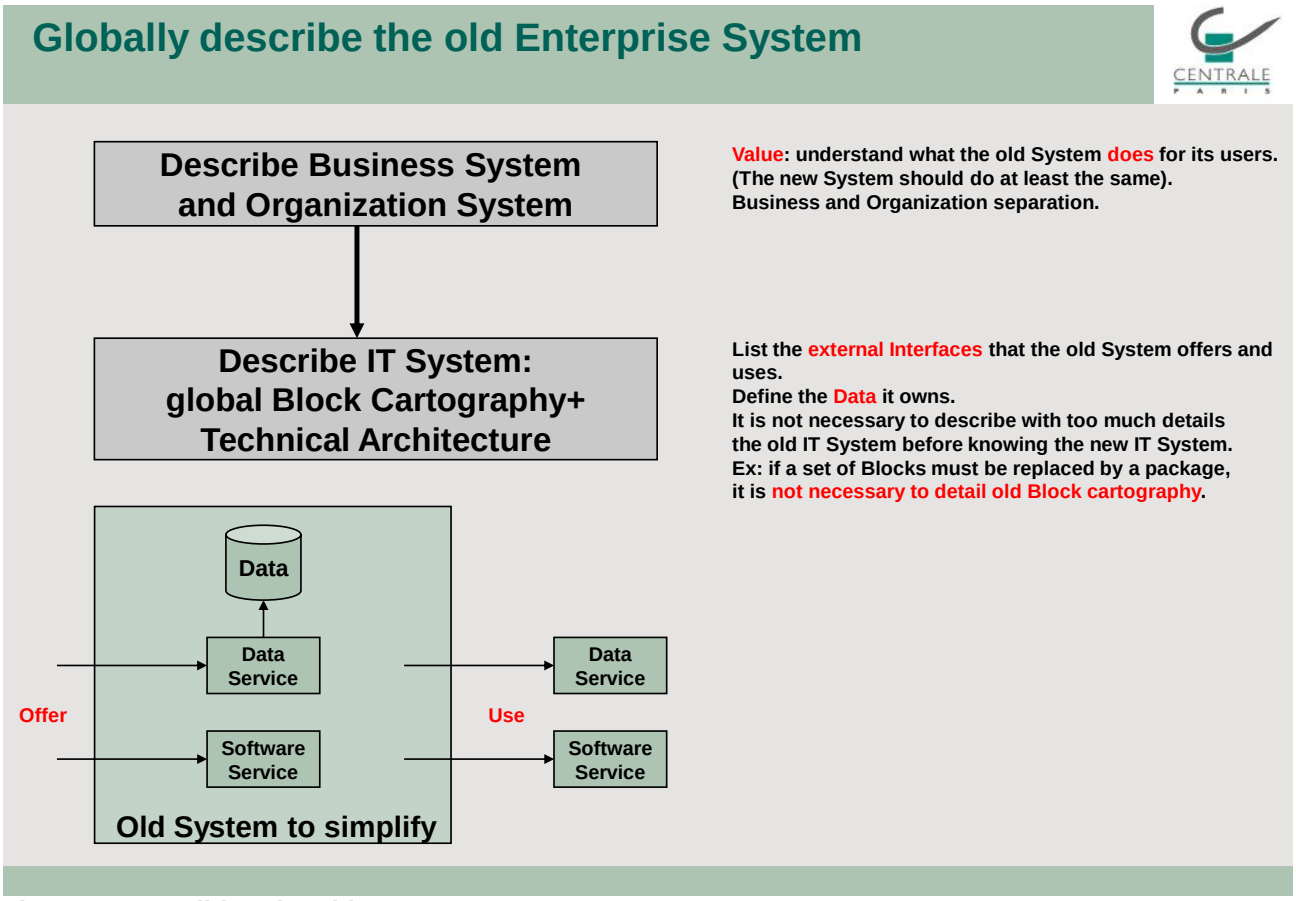


Figure 7: Describing the old System

2.4 Describe the New Enterprise System

Before defining your simplification strategy, you should globally describe the new Enterprise System (Figure 8).

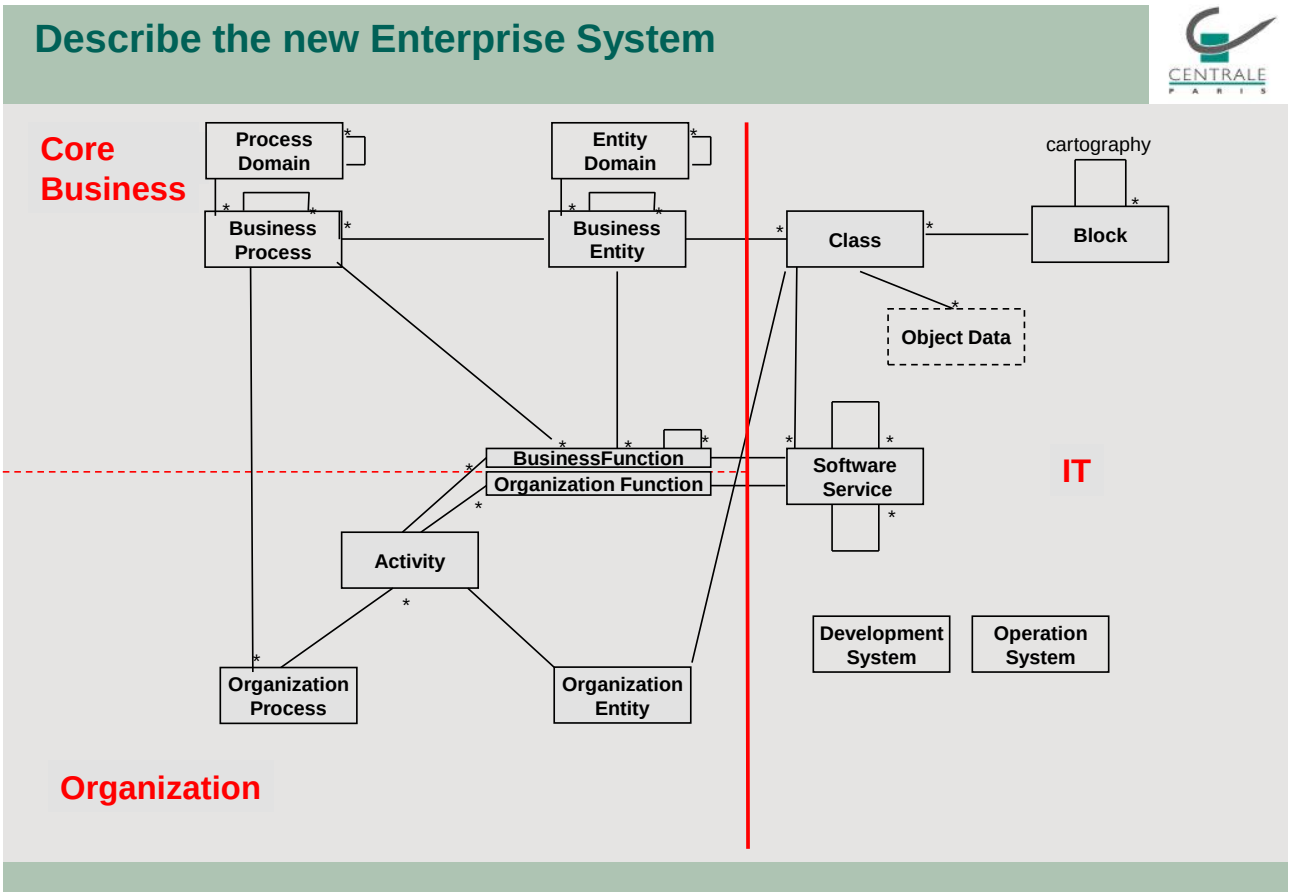


Figure 8: Define new Enterprise System

Obviously, the new system should obey good architecture practices which can be found in the other CEISAR white-papers. For the IT System, you should think of:

- **External Block Cartography**: hierarchical decomposition of Blocks and definition of exchanges between Blocks (see Figure 9 for exchanges between 2 Blocks)
- **Internal Block Structure** (Tier decomposition) (Figure 10 which comes from another white paper on "Block Structure")
- **Operations System** (In particular, intermediation middleware: EAI, SOA, ESB…)
- **Development System** (In particular development standards for each tier…)

Define Block cartography of the new System

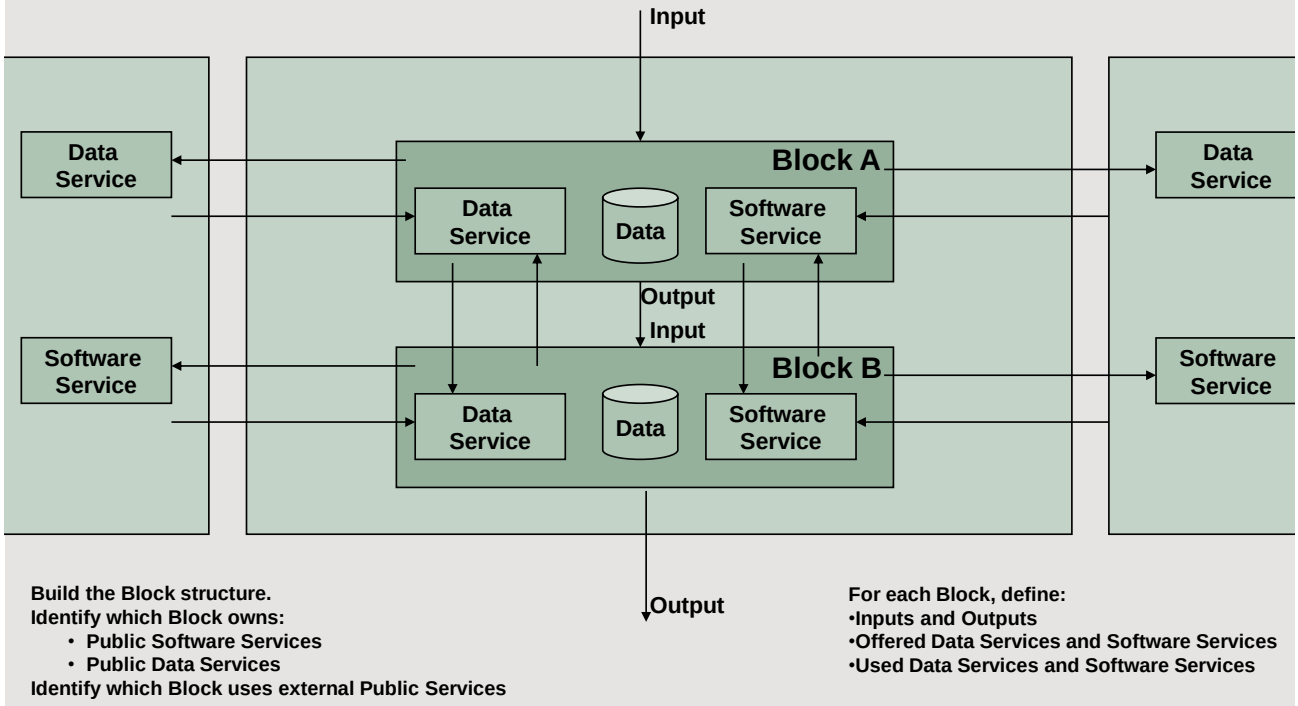


Figure 9: New Block cartography

5 Tier Block Structure

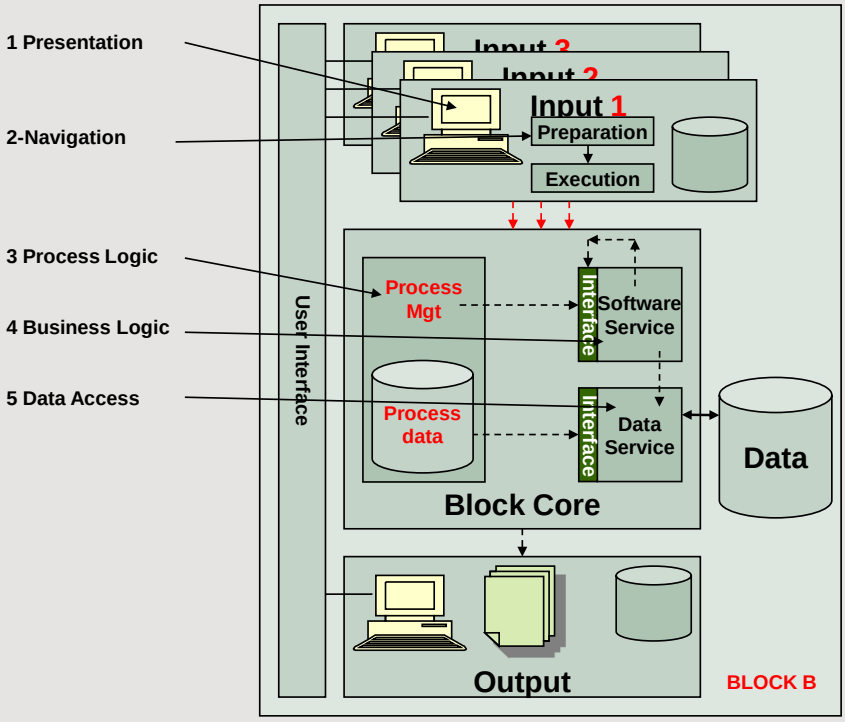


Figure 10: Tiered architecture

When describing the different views of the new System, one should always **relate them to the simplification objectives** that are pursued (See Chapter 1.3):

- How does the new Enterprise System **improve agility**?
  - Good separation between Business Processes and Organization Processes
  - High reuse rate. Simplified development environments
  - Remove silos between systems and support transversal Business Processes. Support real-time Processes
  - Changing Functions are isolated from stable ones and implemented with flexible solutions (rule engines, workflow engines)
  - Modern Development Environment: integrated tools for all development steps, Object Oriented approach, powerful features (versioning, transaction, sophisticated relations, …), reusable Software Services
- Does it **reduce Organization costs**?
  - Cross Processes over different IT Systems
  - Workflow facilities
  - No redundant data entry
- Does it **reduce IT costs**?
  - Every thing which improves agility also decreases costs
  - Reduce the number of operation technologies (OS, DBMS, Middleware)
  - Simplify hardware and network infrastructure
- Does it **improve the IT Service quality**?
  - Global simplicity
  - Change process simplified and well mastered
- Does it **improve the ease of use**?
  - Standard GUIs (portal, presentation, navigation)
  - Common Functions (authorization, printing…)
- Does it **improve knowledge**?
  - Clear structure (global Block Cartography + individual Block structure) is more important than a specific language knowledge

2.5 Decide which Blocks to Keep, Transform or Replace

Deciding which Blocks to keep, transform or replace is a recursive process. In particular, transformation of a Block often consists in a mix of Transformation and Replacement of sub-blocks (Figure 11).

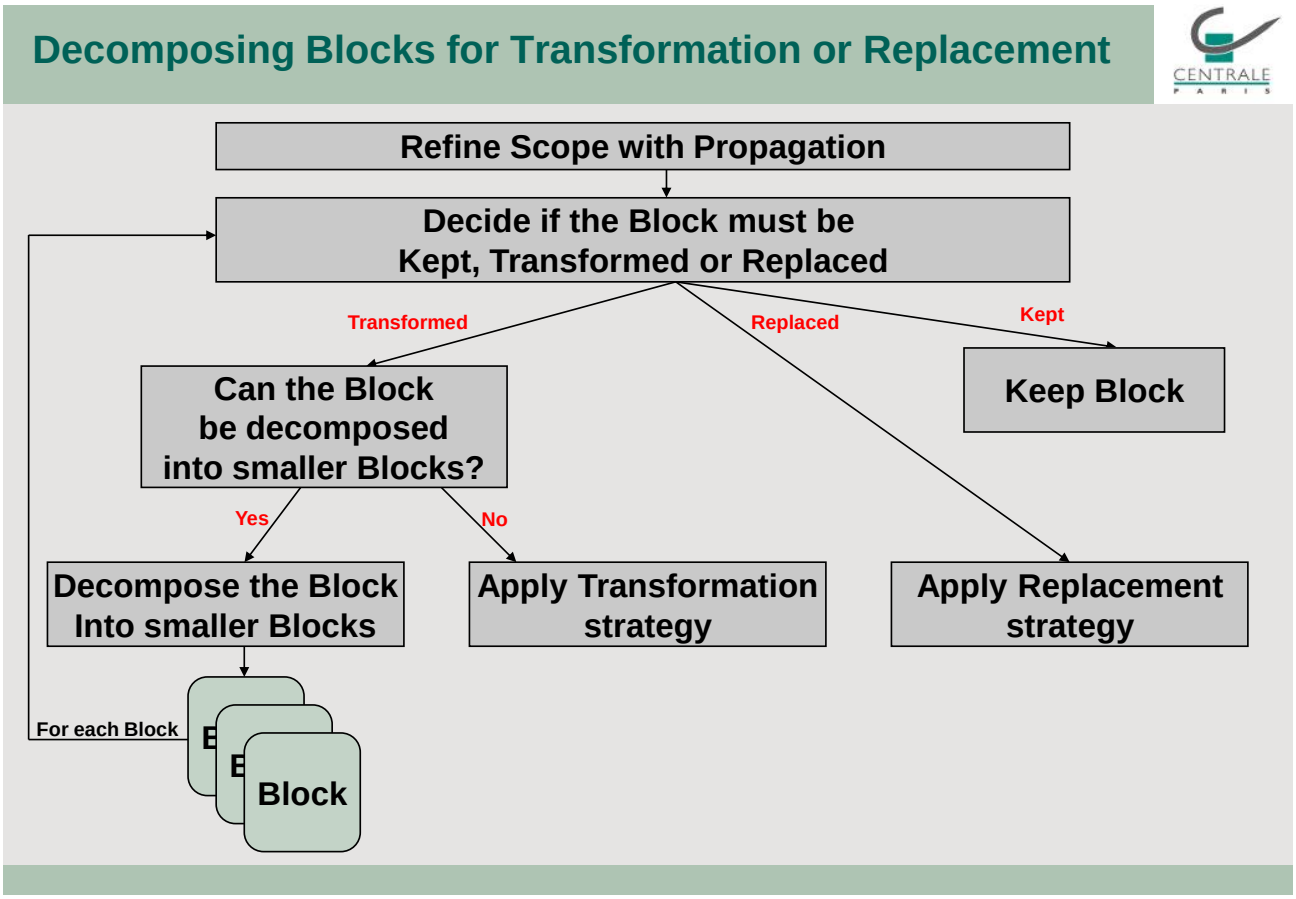


Figure 11: Recursive keep/transform/replace analysis

2.5.1 Refine Scope with Propagation

When a Block outside of the scope is tied to a Block within the scope, one faces two alternatives:

- Either the Block outside of the scope can be lightly transformed to provide interfaces that the Block within the scope can use (once transformed or replaced).
- Or the Block outside the scope cannot be transformed to provide these interfaces.

In the first case, simplification propagation is marginal as it is stopped by the light transformation of the peripheral Block. In the second case, simplification propagation will include the peripheral Block and the scope will need to be more heavily extended (Figure 12).

Chapter 3.1 describes techniques to avoid scope propagation.

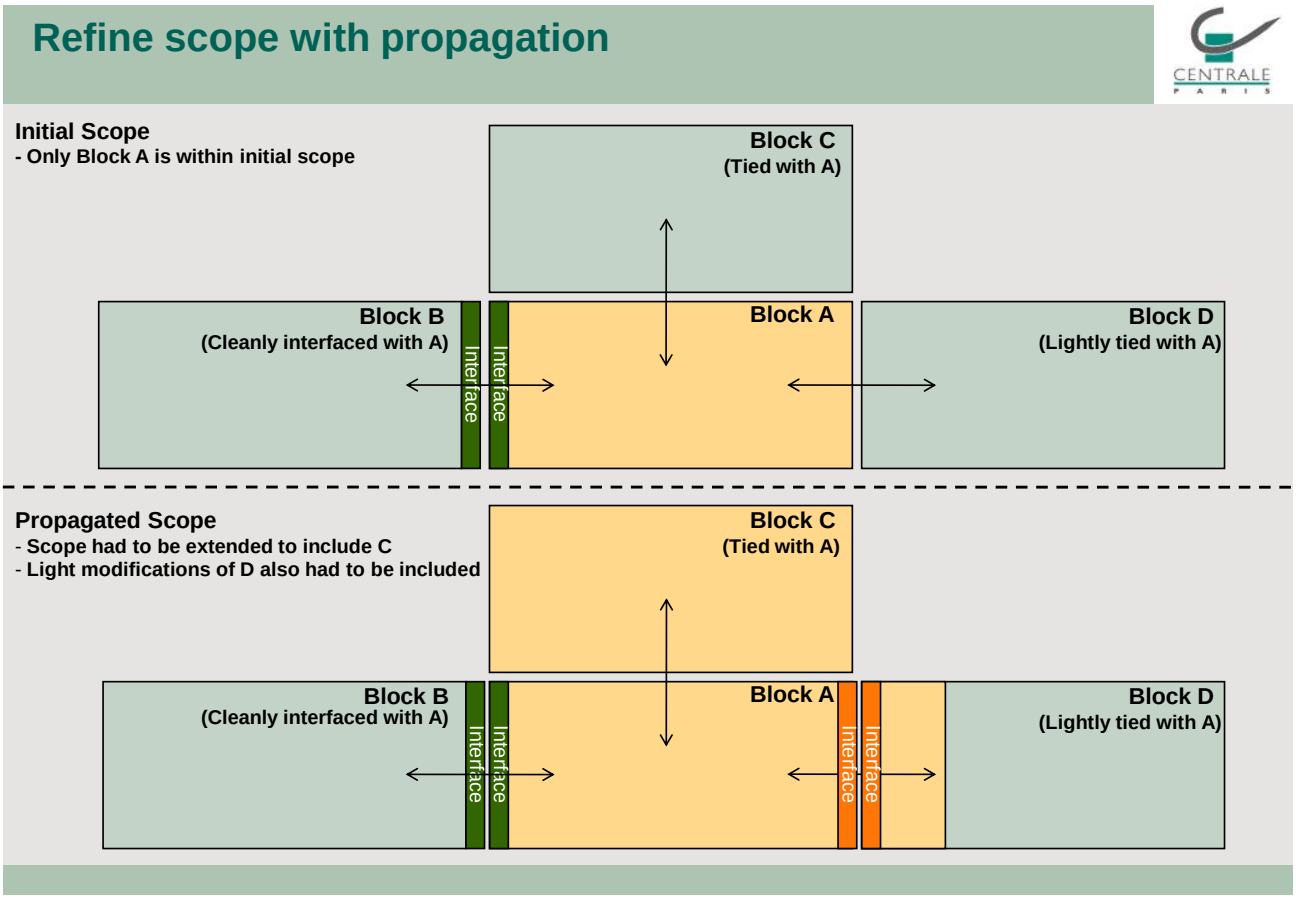


Figure 12: Refining scope

2.5.2 Deciding to Keep, Transform or Replace a Block

Blocks which comply with the new Business, Organization and IT System (new Block Cartography, new Development and Operation Environments) and have acceptable complexity levels should be kept. Others should be transformed or replaced, based on the simplified decision map (Figure 13).

How to decide Transformation or Replacement?

	Transformation **Progressive improvements on existing software structure**	Replacement **Supression of an existing software structure**
Advised When	Asset value is strong	Asset value is weak
	Average or good software structure (To document)	Bad software structure (Do not document in details)
	Knowledge exists (Possible use of retro-documentation tools)	Knowledge does not exist
	Technology is maintained	Technology is obsolete
		New System is a package
Properties	In-house development only	
	Progressivity: **lower Risk**	Higher risk: big change for Users, Partners, Developers, IT Operations
	High global cost, but faster results and spread-out on successive budget cycles	**Lower global investment**: one big change costs less than progressive changes

Figure 13: Decision map for transformation vs replacement

2.6 Describe Governance

Proper Governance is key in legacy simplification programs. Simplification is a long journey which is made difficult by all the ongoing events: new projects, new urgent needs for evolutions, urgent corrections, etc.

2.6.1 Strategic Governance

Set-up a **small governing body** to follow simplification, preferably with people who have **operational responsibilities**, and not only architects (project managers, business analysts, etc.). It should even include **business people** so that they become familiar with the process, and see their business benefits (ultimate benefit is business agility). Best people should spend more time to build or improve solutions than checking what it is done by others.

Have **metrics** to make sure you stay on course. Having simplification metrics on your IT strategic dashboard, or on each project dashboard increases your chances to reduce complexity.

Reducing complexity should be a state of mind of all IT professionals. It requires appropriate change management techniques. **Train IT people** to integrate this state of mind.

Except for companies whose enlightened business leaders have grasped the criticality of having an optimized IT architecture asap (and mobilized their entire IT workforce on it), reducing complexity is seldom a priority.

A tactical solution is to include the following rule in your governance framework: **each major project should devote a minimal % of its budget to simplification** of the existing IT landscape.

An alternative is that **each project** goes before an Architecture Committee where it **must explain** (among other things) **how it makes the Enterprise System simpler** in the end (possibly showing actual

complexity metrics before the project and projected complexity metrics after the project has been delivered).

If the Information System is organized by Business Domains, the **domain managers** (Business and IS) should have **simplification in their objectives and be held responsible**: they have to build the vision and roadmap and make sure it is executed through their portfolio of projects and maintenance work. They can be **assisted by domain architects**

2.6.2 Project Governance

Projects that are concerned with simplification (directly or indirectly through the "opportunistic" execution) should **share their simplification strategy with their business users** and ask for commitment.

Prefer a **formal decision**: "no immediate return" means "risk of changing mind".

Steering committees, even for business projects, should keep this topic on the agenda to make sure it does not disappear behind day-to-day priorities (especially with the Opportunistic Execution).

Define project phasing:

- Decide to implement **read-only** queries or services first because they do not destabilize the System
- Then define in which order the Blocks will be simplified: prefer **input Blocks before output Blocks**
- If two blocks are tied, **start with the Block that is called less**

But **business priorities** may override these rules. By delivering business value earlier, one **improves the ROI** and cash flow of the project, gets **user feedback and acceptance**, gives **visibility** on the control of the project and **builds confidence** with the business.

Even though the final cost and time needed to complete the simplification project will be higher, **CEISAR advises to have a phased approach** with frequent releases, **intermediate business benefits** and landing points where a stop-and-go is sustainable.

A rule of thumb is that each phase should last less than a year and cost less than 10 m.y if possible.

Define **Planning** and Budgets and isolate transparently the part (with benefits!) which is devoted to simplification. Share it with users, so that they buy into the project, understand its goals and see their business benefits.

3 How to implement?

In this chapter, we will start diving into more technical details as we describe the actual **implementation process** to deliver simplification. It is targeted towards more technical people, like architects and project managers.

To make it simple, we modelled the information system as a pair of Blocks that use each other in any possible way (A uses B services and outputs, B uses A services and outputs).

What is shown on this pair of blocks can serve as a model and be extended to any combination of blocks.

3.1 Isolate Scope

If A and B are not cleanly interfaced, which means A and B access each other's data and programs without service interfaces (Figure 14), it is advised to **do some preparation first to uncouple A from B**, before they can be replaced or transformed separately.

> *Sometimes, scope isolation is more easily done in parallel to the transformation or replacement work. However, principles presented in this chapter remain.*

In this chapter, we transform Block A and look at avoiding scope propagation to B (which is outside of the initial scope). Consequently, we look at **minimal reengineering of B**.

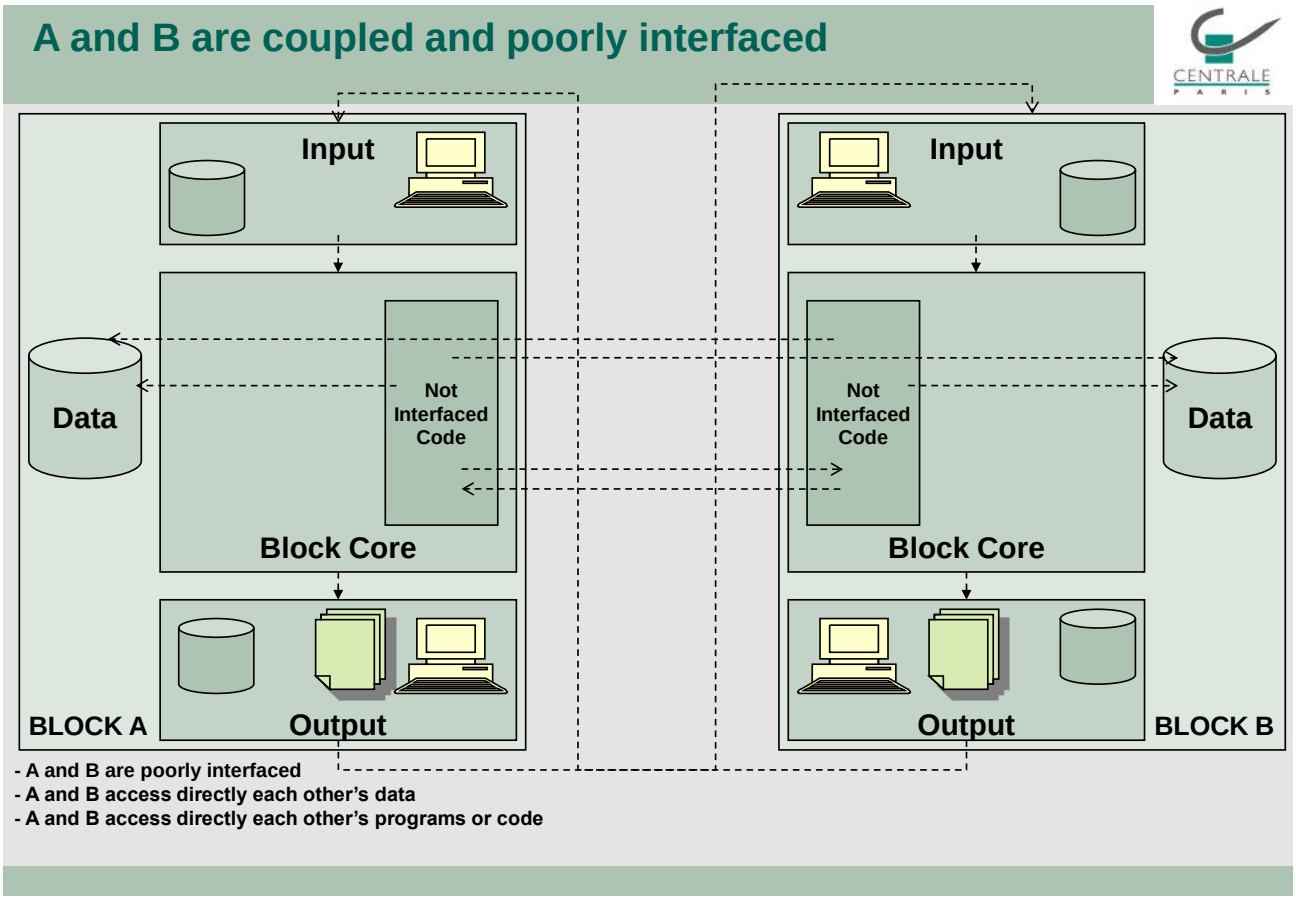


Figure 14: Coupled blocks

Step 1 consists in **adding interfaces to B** (Figure 15). This minimal action will allow to have the new Block A access B only through proper interfaces and consequently avoid being tied to it again.

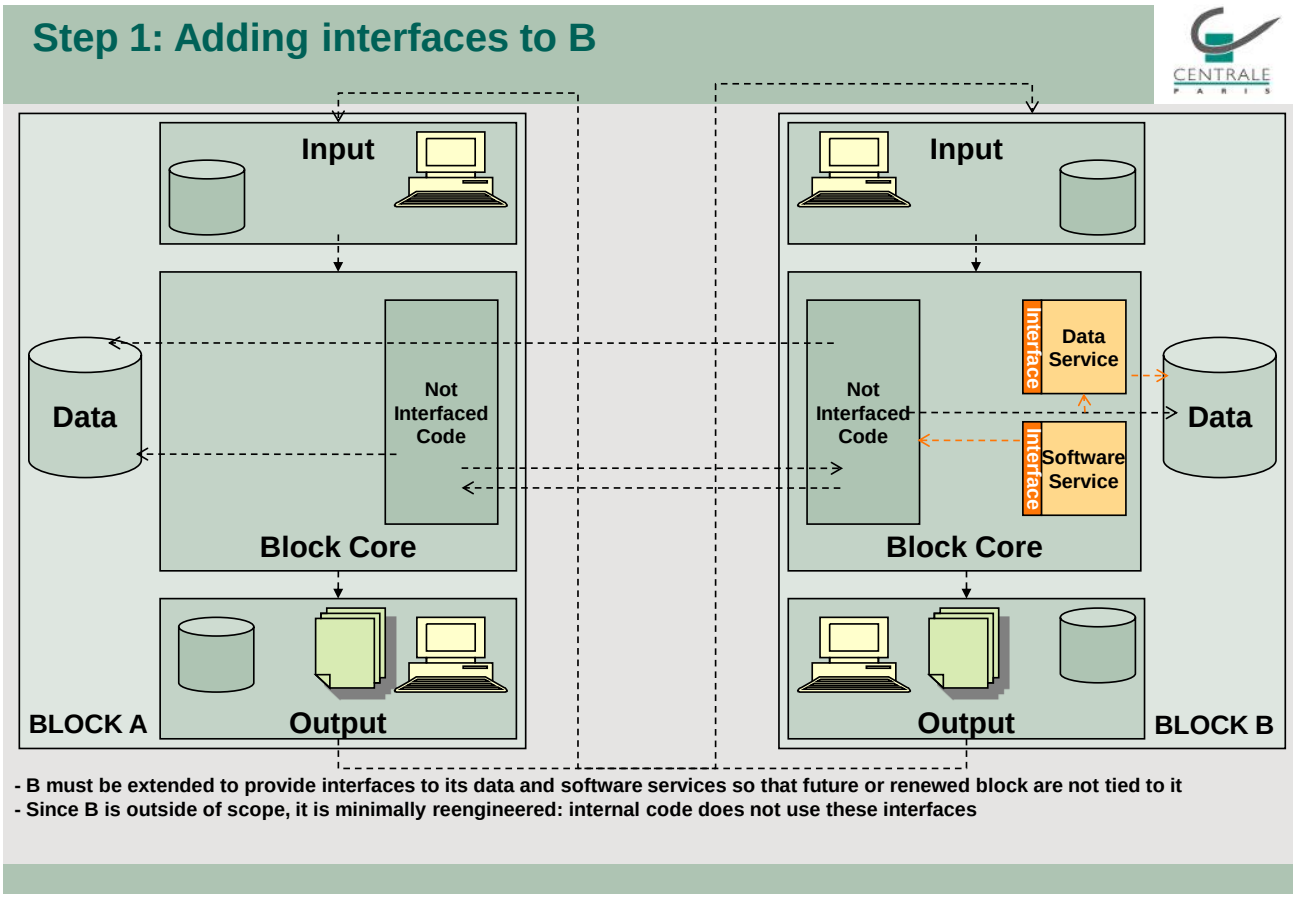


Figure 15: Adding interfaces to B to avoid future coupling

Since we look at minimal reengineering of B, **B's new interface will likely be an additional wrapper's code** that will directly access B's data and encapsulate B's existing programs or transactions, without further impact into B's core code (in particular, B will not be transformed to use this new Service layer).

> ***In some cases**, in particular if B's data is unstructured, organized as silos, based on an obsolete database technology (hierarchical) or needs transformation, **one will replicate B's data** (through replication techniques ; see Chapter 4.4) to a clean and modern target database **and will build B's new Service layer onto this target** database.*

Step 2 consists in **transforming or replacing A with minimal impact on B** (Figure 16).

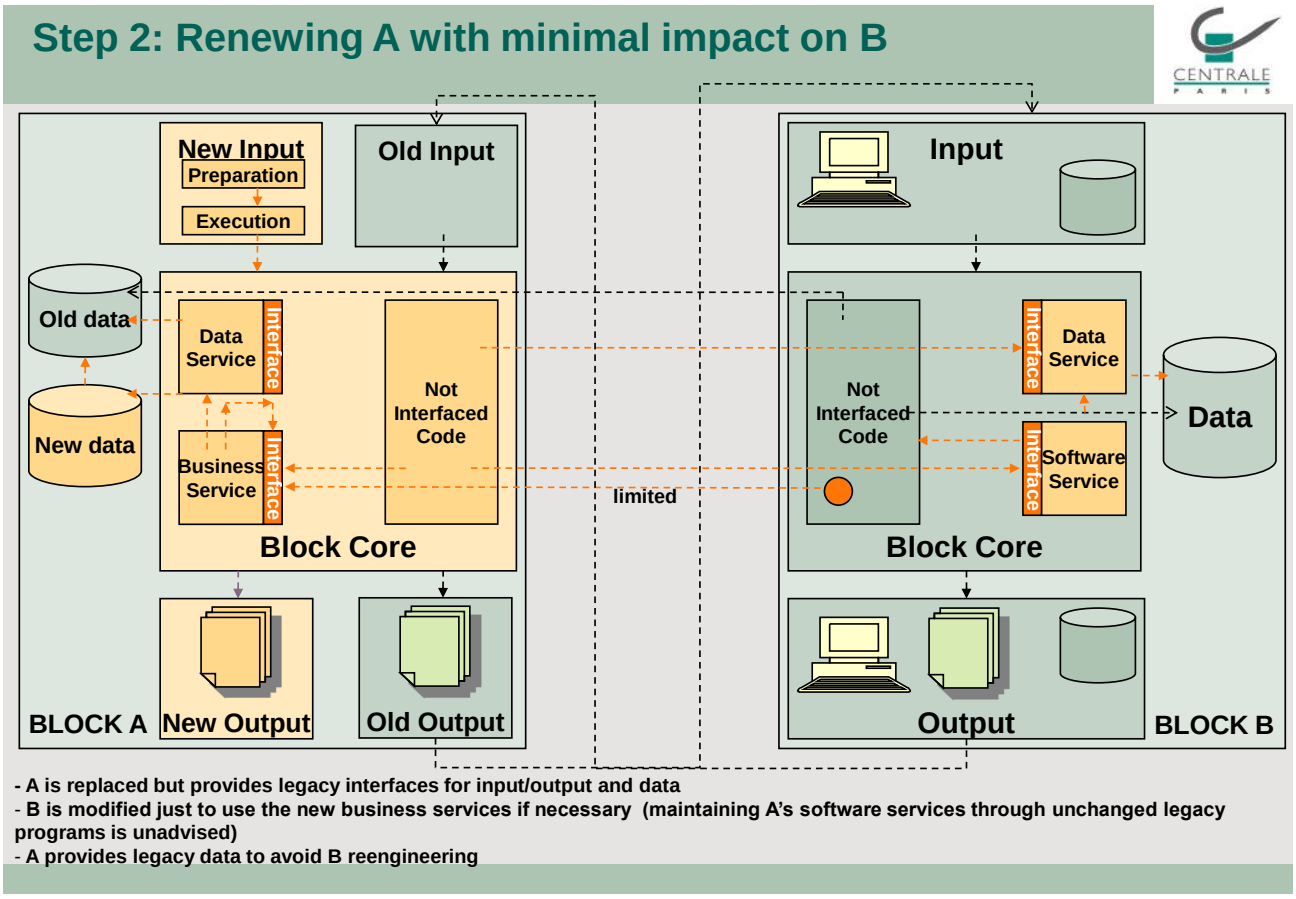


Figure 16: Renewing A with minimal impact on B

If we look at minimal transformation of B, the less impacting solution is that **A maintains legacy inputs/outputs and maintains the legacy database** (through data replication or A's data access layer).

On the other hand, **CEISAR does not advise to give access to A's software services through legacy programs or code**. Legacy programs or code are too specific to A's legacy technology or code structure and providing compatibility would put too much constraint on its architecture. Consequently, B will have to be transformed to use A's new software services.

If B's access to A's code are too numerous or too complex, then A and B have to be transformed or replaced together.

We are now in a situation where A is uncoupled from B and scope propagation has been limited. However, since we looked at minimal reengineering of B, B remains coupled to A through the direct access to the legacy database. In some cases, it can be worthwhile to go one step further.

Step 3 consists in **modifying B so that it accesses A's data only through A's interfaces** (Figure 17). This additional step reduces the coupling of B using A. It removes the constraints that A has to maintain a legacy database. On the other hand, **it should be done only if B has limited and well know access to A's data.** Otherwise, it becomes a full transformation, which **means that B has actually been added to the scope**.

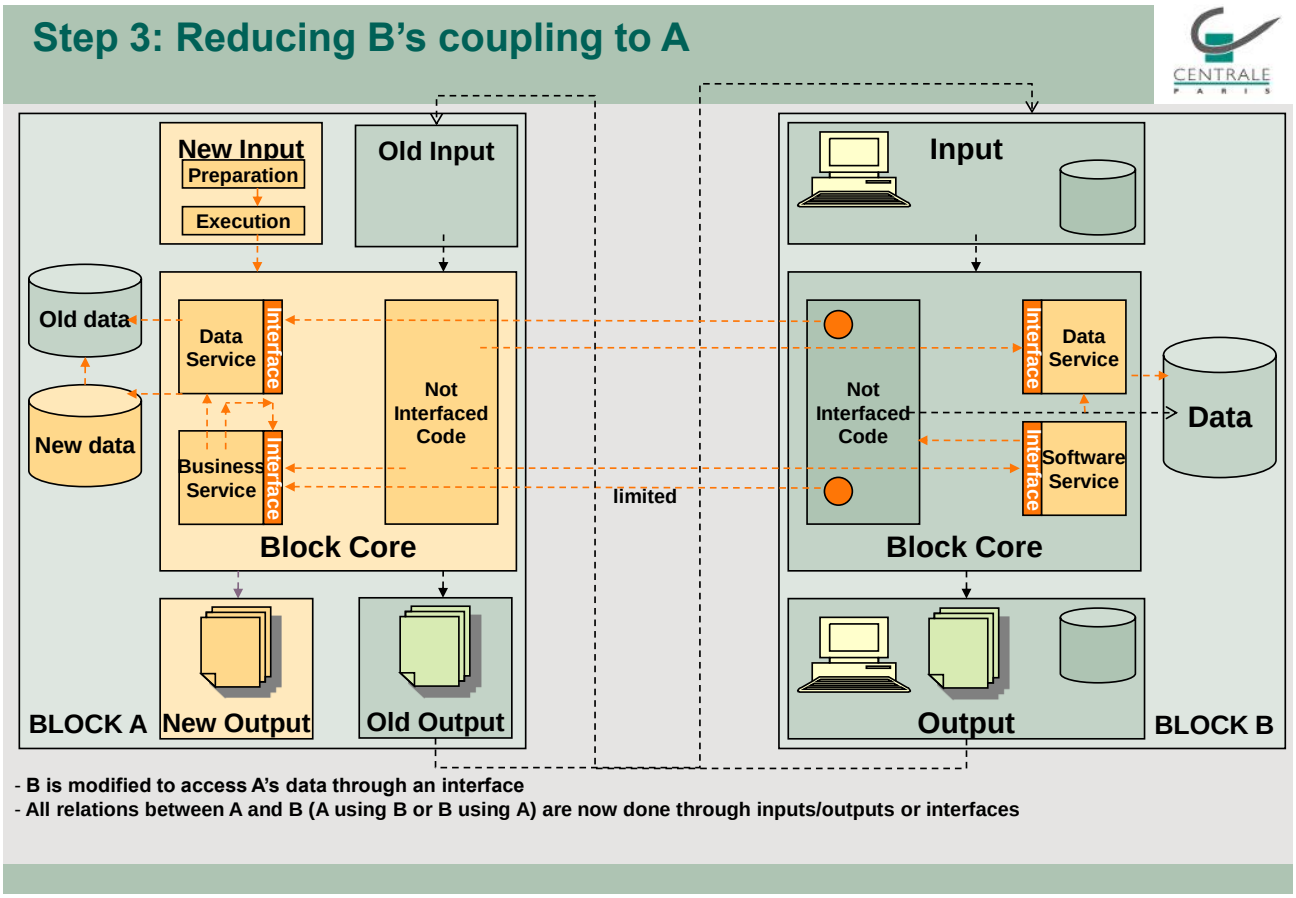


Figure 17: Reducing the coupling of B using A

3.2 Apply Transformation Strategy

The Transformation Strategy consists in preserving some parts of the old System (code and data) which are considered valuable, but to progressively modify it to fight obsolescence or to improve agility. In general the objectives are to (by order of ambition):

- Clean-it up and structure it better (cleanse data and code, share controls, etc.)
- Make the user interface web-compliant
- Re-organize it into separate and properly architectured tiers (UI Logic, Process Logic, Business Logic, Data Access, Data) which improve agility and favor reuse (See Chapter 2.4).
- Isolate reusable data access and business functions and expose them as Services to external applications
- Slim down the old System to the sole reusable Functions (and extract Process logic in external applications)

Figure 18 describes the different steps of the Transformation Strategy.

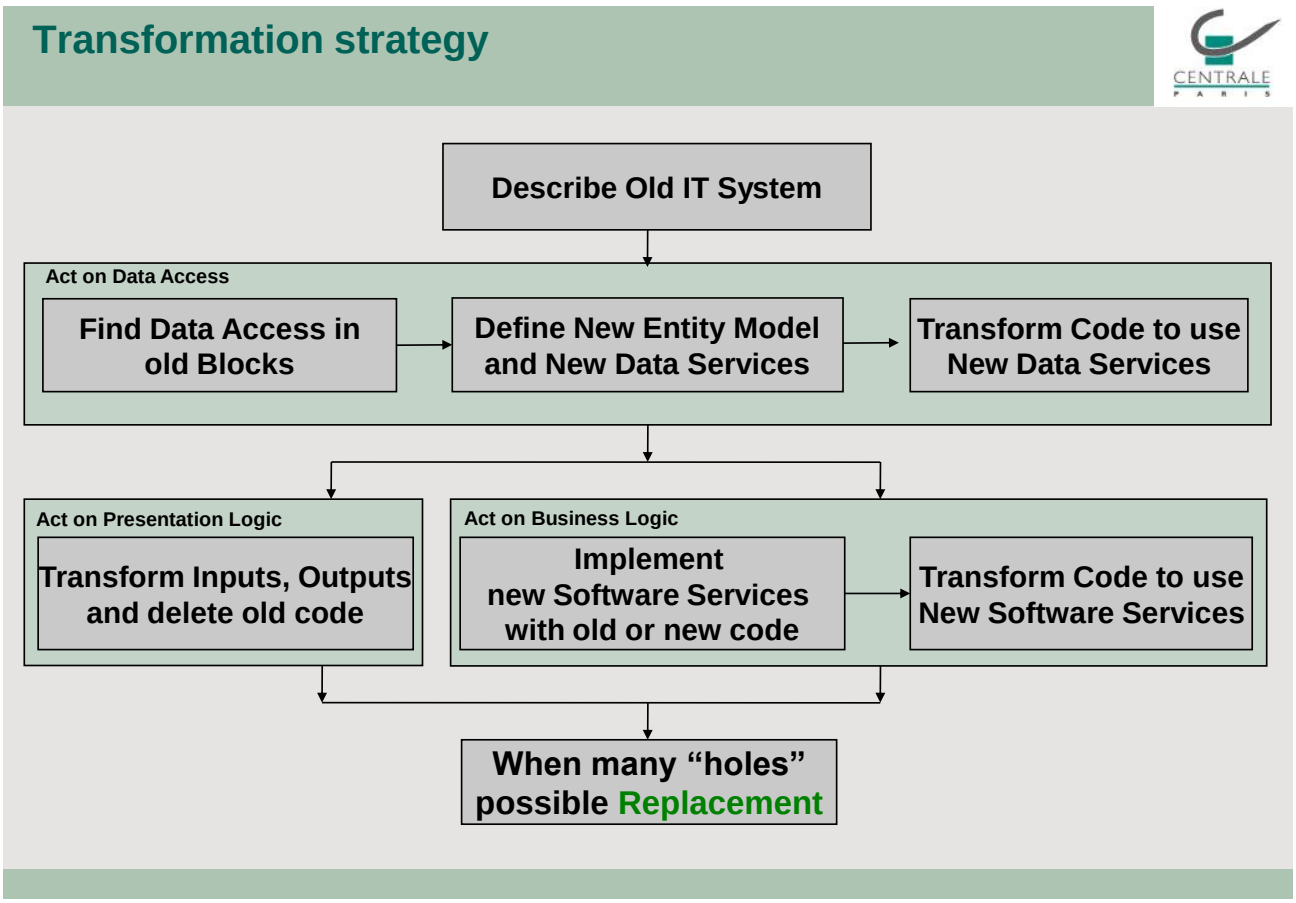


Figure 18: Applying Transformation Strategy

In this chapter, we transform Block A only (and minimally Block B to adapt to A's changes). If in the scope, Block B transformation is symmetric and can be conducted in parallel or in sequence.

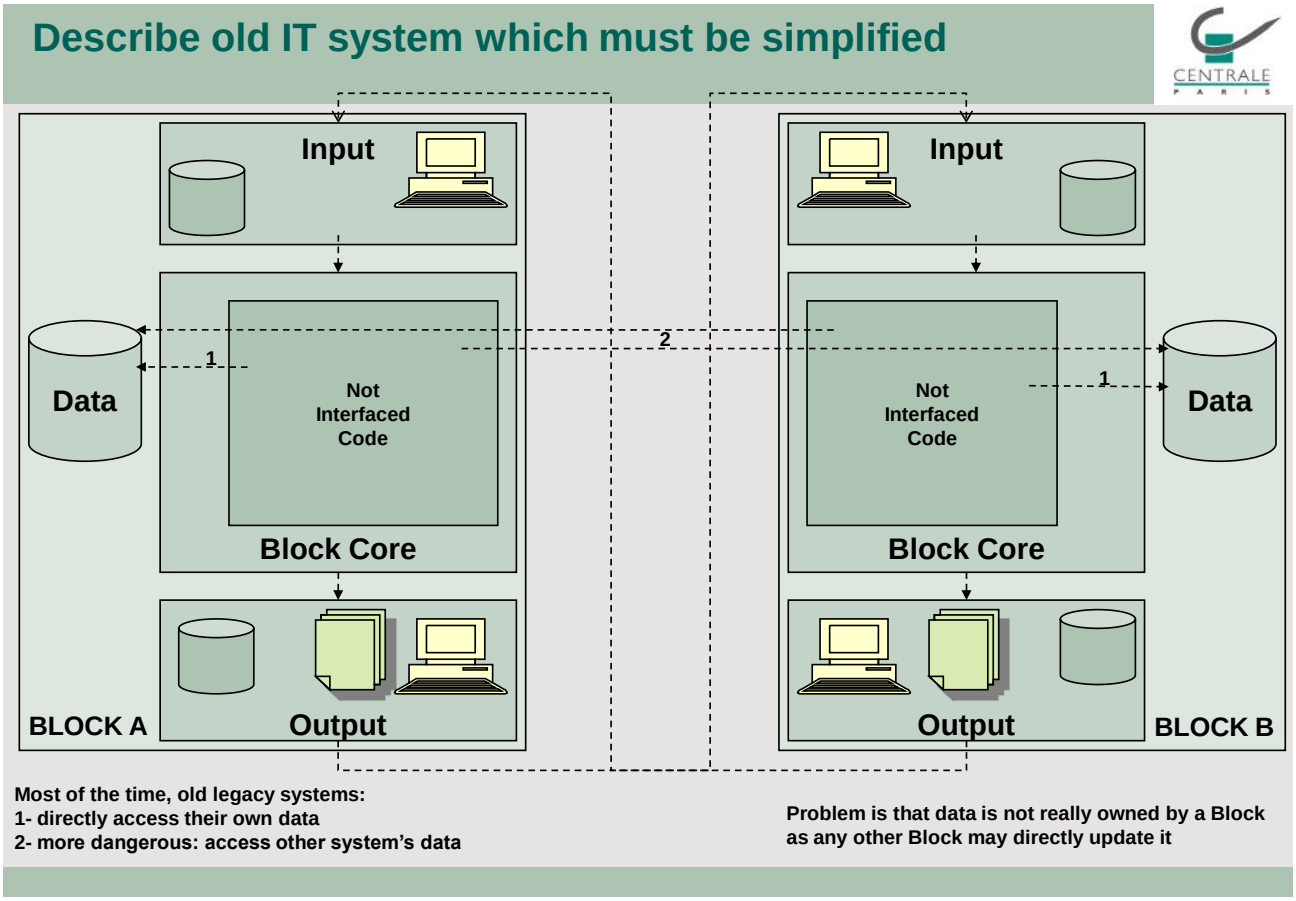


Figure 19: Description of the old system

Start by creating **Data Access Services** and transforming the code to access data only through these Services (Figure 20).

- Volume of data access code is often 50% of total code: huge investment
- For performance purpose, in some old Blocks, a record includes data from different Business Entities: a single data request must then be decomposed into several data Services and be optimized
- Code analysis tools can help find-out where data is used and accessed within programs (See Chapter 4.8)

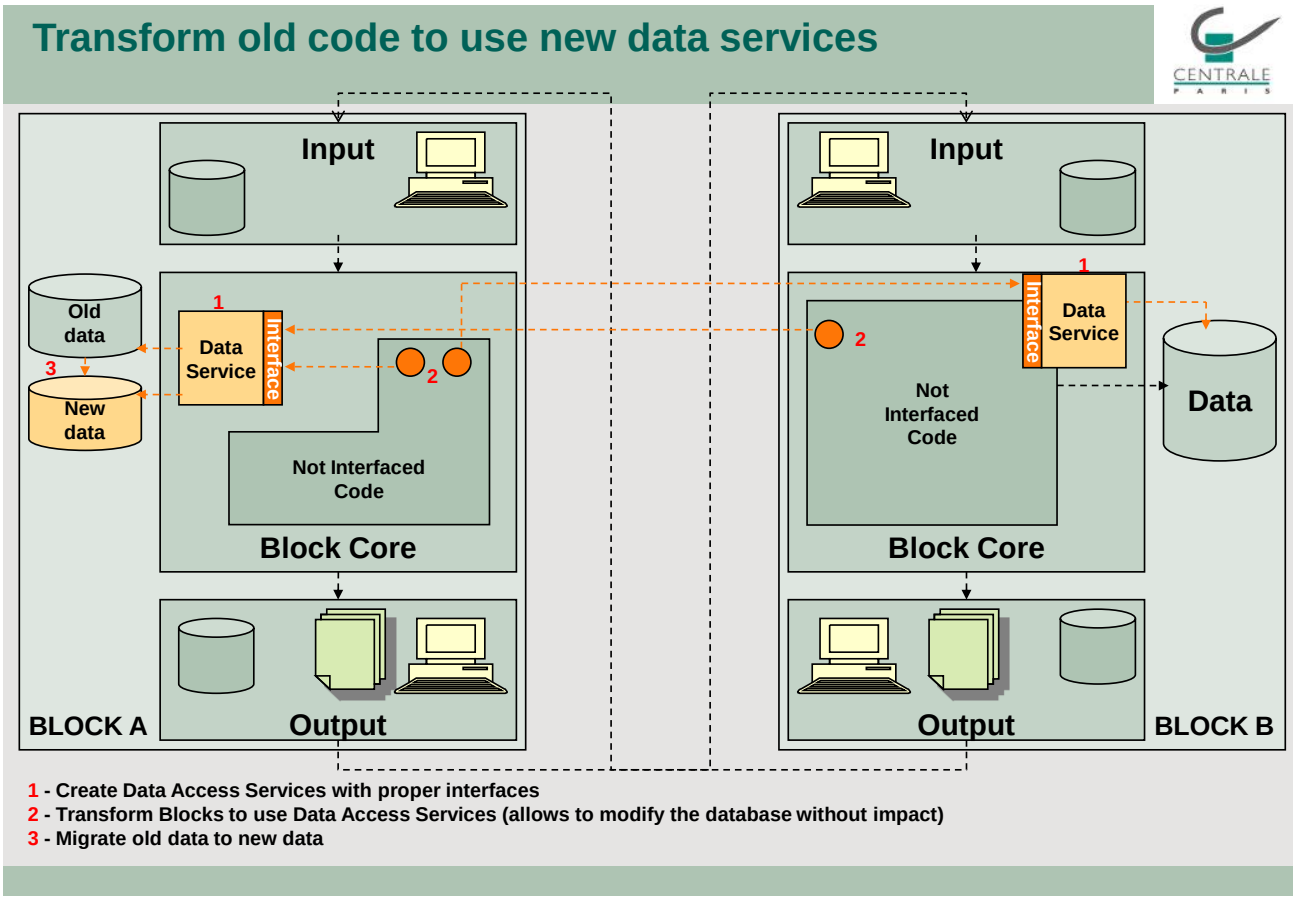


Figure 20: Creating and using Data Access Services

"New Data" may include many different changes:

- New Business Entities
- New Identifiers
- Include Versioning
- New DBMS (like going from a classical File System or hierarchical database to a relational database)
- Less databases
- More Attributes by Class
- Replication mechanisms with subscription Services if duplicated data

Then isolate the reusable Business Logic and organize it as interfaced **Software Services**. Modify inputs/outputs to **isolate presentation an navigation** logic. Transform the old System to use them (Figure 21):

- **Create new Services** and delete equivalent code

  Even if you are just wrapping the legacy *with new technology or exposing services,* ***you need to understand it****. We have witnessed projects that have tried to wrap and reuse legacy functions as black boxes because they knew they had value but could not understand them anymore. These projects fail miserably because they end up facing unexpected behaviours of the System that they cannot explain.*

- **Call Public Services**
- **Change Input**
  - Identify input records
  - Isolate input code from core application code
  - Change input functions: light client, on line controls, workflow, ...
- **Change outputs**
  - Identify output records (including flows towards new Services)
  - Isolate output code
  - Change output functions: light client, printings, archives, complex requests, …
- **Change Block Core** if necessary: the volume of code should be reduced by former actions

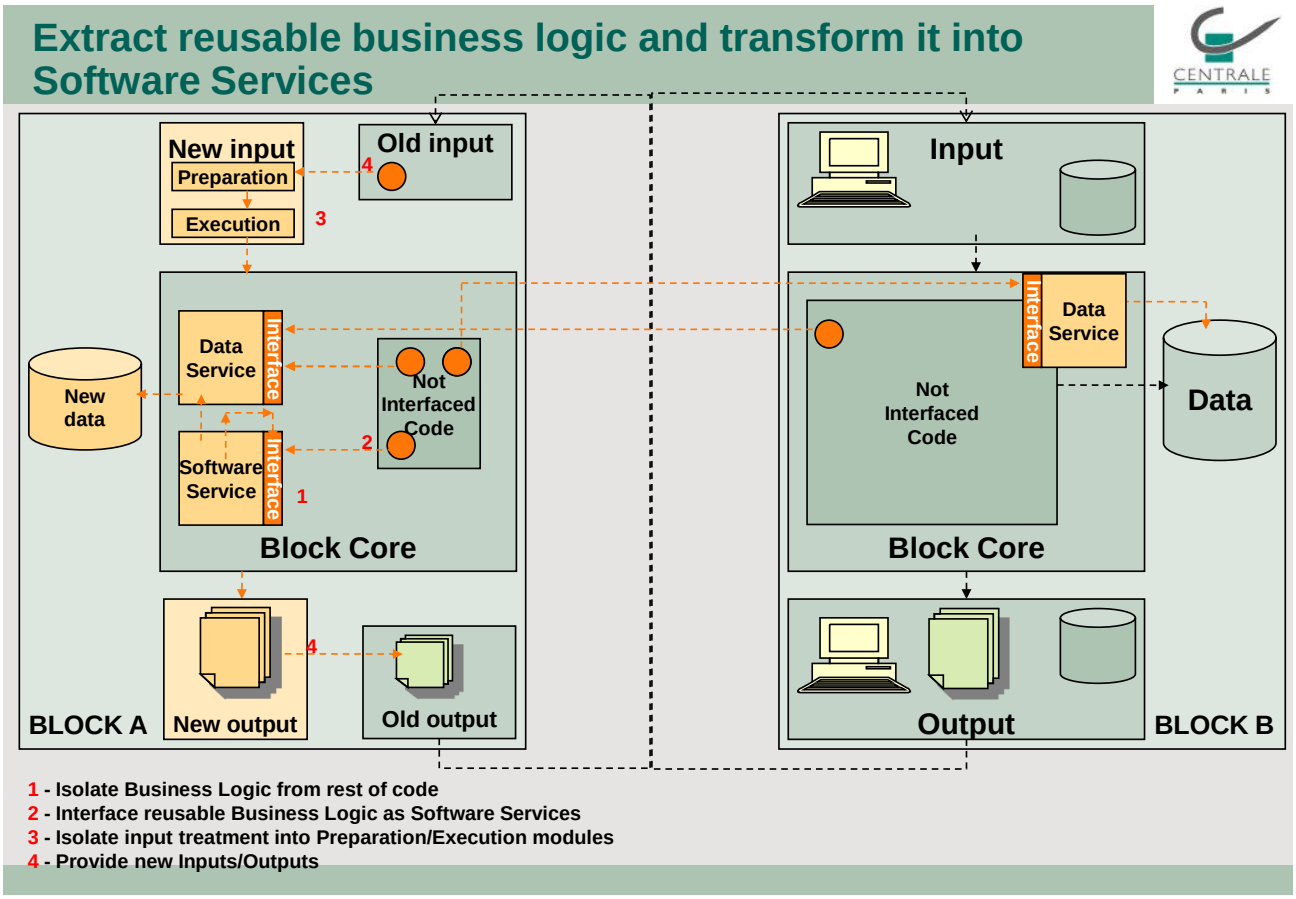


Figure 21: Extract reusable business logic and transform it into Software Services

Once you have reached this stage, you can even slim the old System down even further if it is to be used only as a data and service provider (with process logic in external applications or execution engines).

3.3 Apply Replacement Strategy

The straight replacement of a whole legacy is very rare because it is often unmanageable or too risky. However, the replacement of a big block in one piece, instead of small block by small block, is sometimes necessary (See Chapter 2.5.2).

In this chapter, we will describe the **replacement process**, i.e. the different steps that should be followed during a replacement strategy (Chapter 3.3.1).

We will then describe two execution strategies the "**Big-Bang**" approach when one switches over from one system to the other in single shot (Chapter 3.3.2), or the "**Parallel Execution**"[1] strategy, where the two systems (old and new) run in parallel and in sync for a while (called the transition phase), thus mitigating the risk (Chapter 3.3.3).

3.3.1 Replacement Process

Figure 22 shows the replacement process.

[1] Sometimes people use the term "Parallel Execution" to describe the tuning phase where the new System is deployed separately from the old System (like in Big-Bang) and users do dual inputs in the new and old System for a while, to make sure it behaves as expected. In this white-paper, we use the term "Parallel Execution" only for solutions where the two Systems are kept in sync automatically

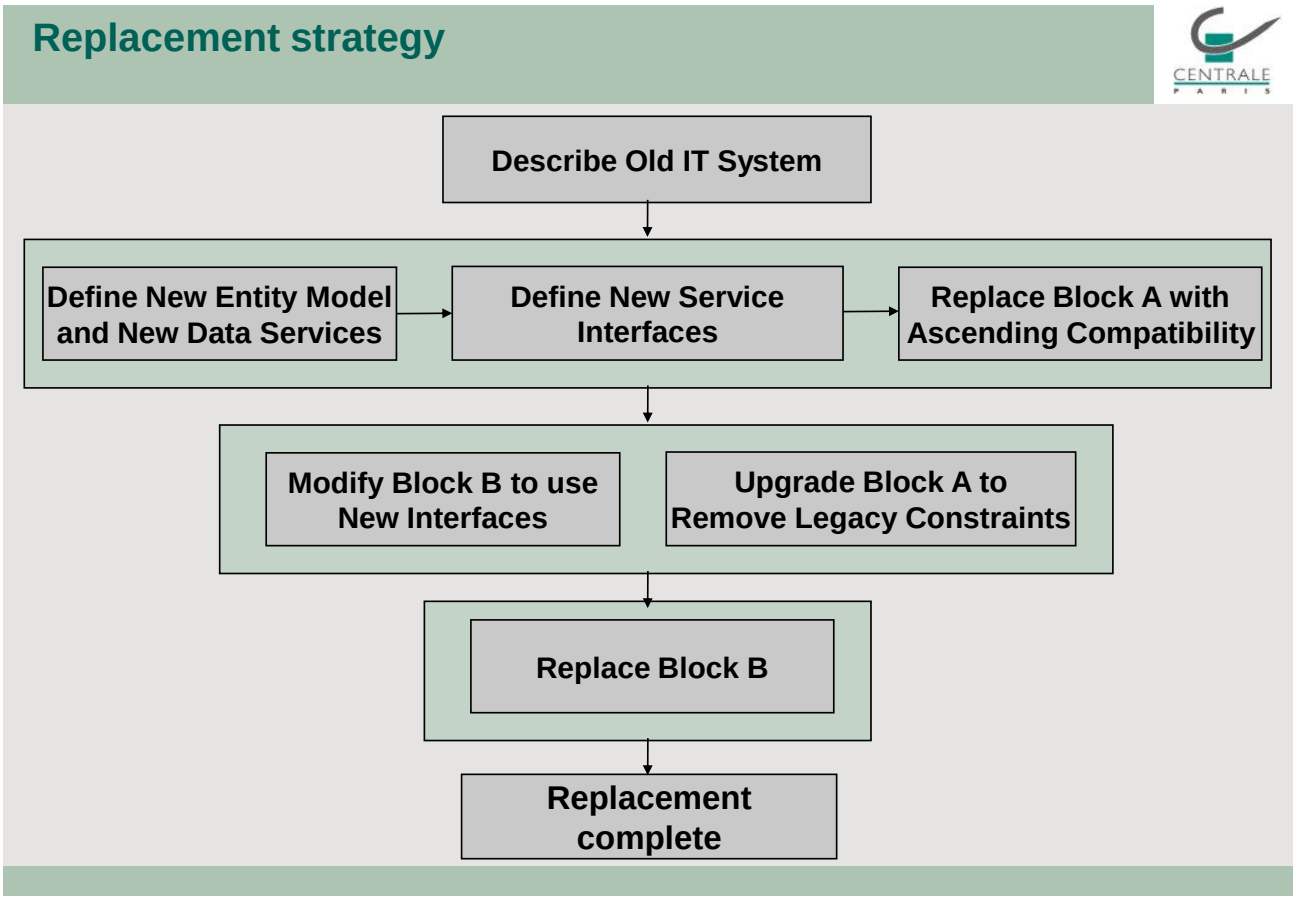


Figure 22: Replacement strategy

We start with the assumption that **preparation work** (See chapter 3.1) **led to two Blocks that are cleanly interfaced**: A and B use each other only through input/output or Software or Data Services (Figure 23).

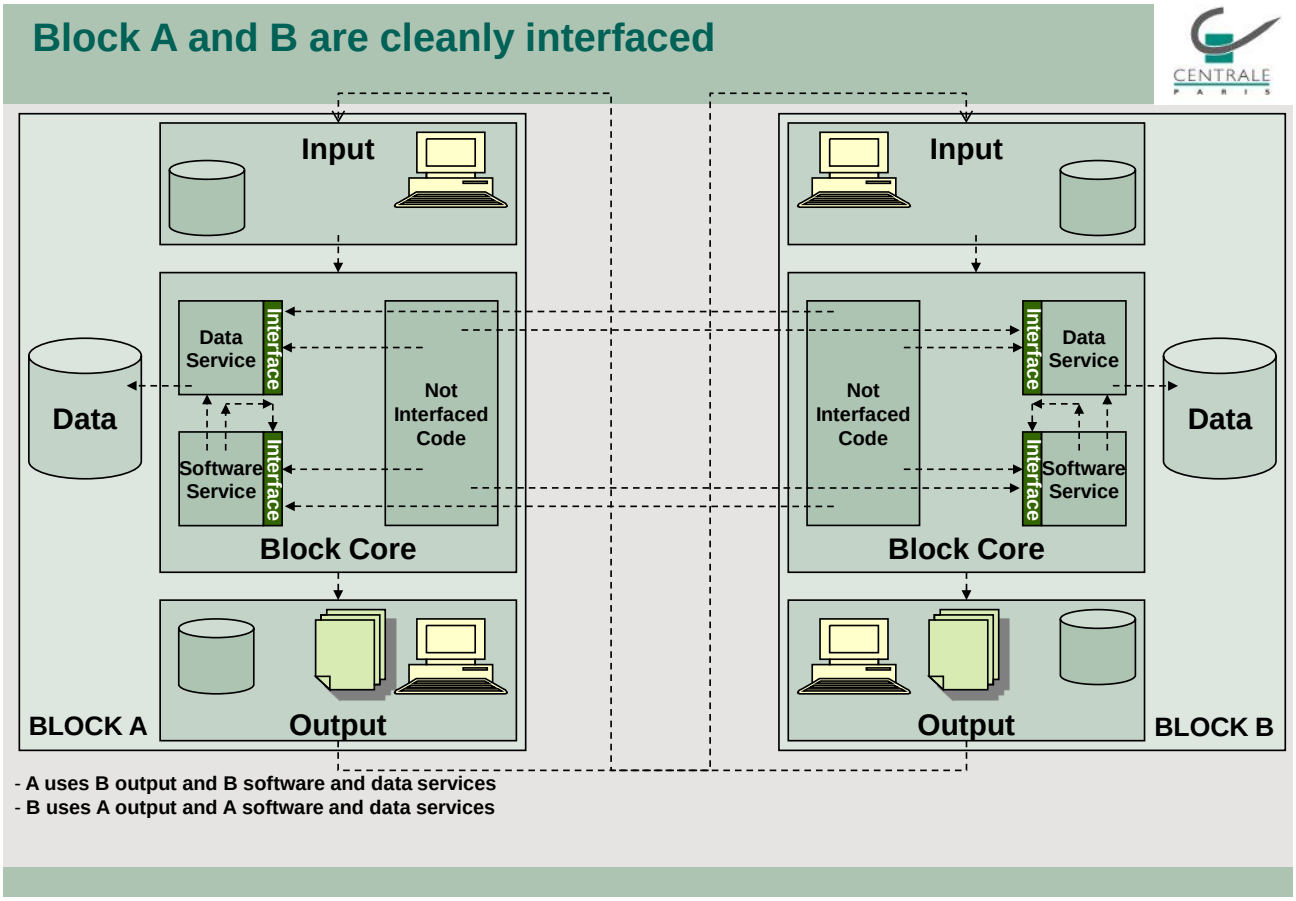


Figure 23: Block A and B are cleanly interfaced

The proposed **approach is made of three progressive steps**, each of them possibly final.

- Step 1 allows to replace Block A with no impact on Block B. It can be the final step when Block B is outside of the scope and shouldn't be modified.
- Step 2 allows to replace Block A with limited impact on Block B. It can be the final step when Block B is outside of the scope but can handle some modifications. These modifications allow to reduce final complexity compared to step 2, at the expense of additional work and some Block B transformation.
- Step 3 is the final step when Block A and Block B are within the scope.

Table 1 summarizes which steps should be followed depending on B's situation.

Table 1 : Replacement Process

B is out of scope		B is in scope	
B cannot handle any change	B can handle some changes	B is replaced	B is transformed
Step 1	Step1 Step 2	Step 1 (Step 2) Step 3	Step 1 Step 2 Step 3

Unless otherwise noted, **CEISAR advises to follow steps 1, 2, 3 progressively** as they add security. They limit the change of each step to a single entity (Block or interface change), making it easier to spot out potential problems.

Variants are proposed depending on the context of the simplification strategy.

Step 1 consists of **replacing Block A but maintaining its legacy interfaces** so that Block B can be left unchanged (Figure 24).

In this situation, Block A accepts new inputs as well as old inputs and provides old interfaces as well as new interfaces: it offers **ascending compatibility**.

Block B can be left totally unchanged, which limits scope propagation.

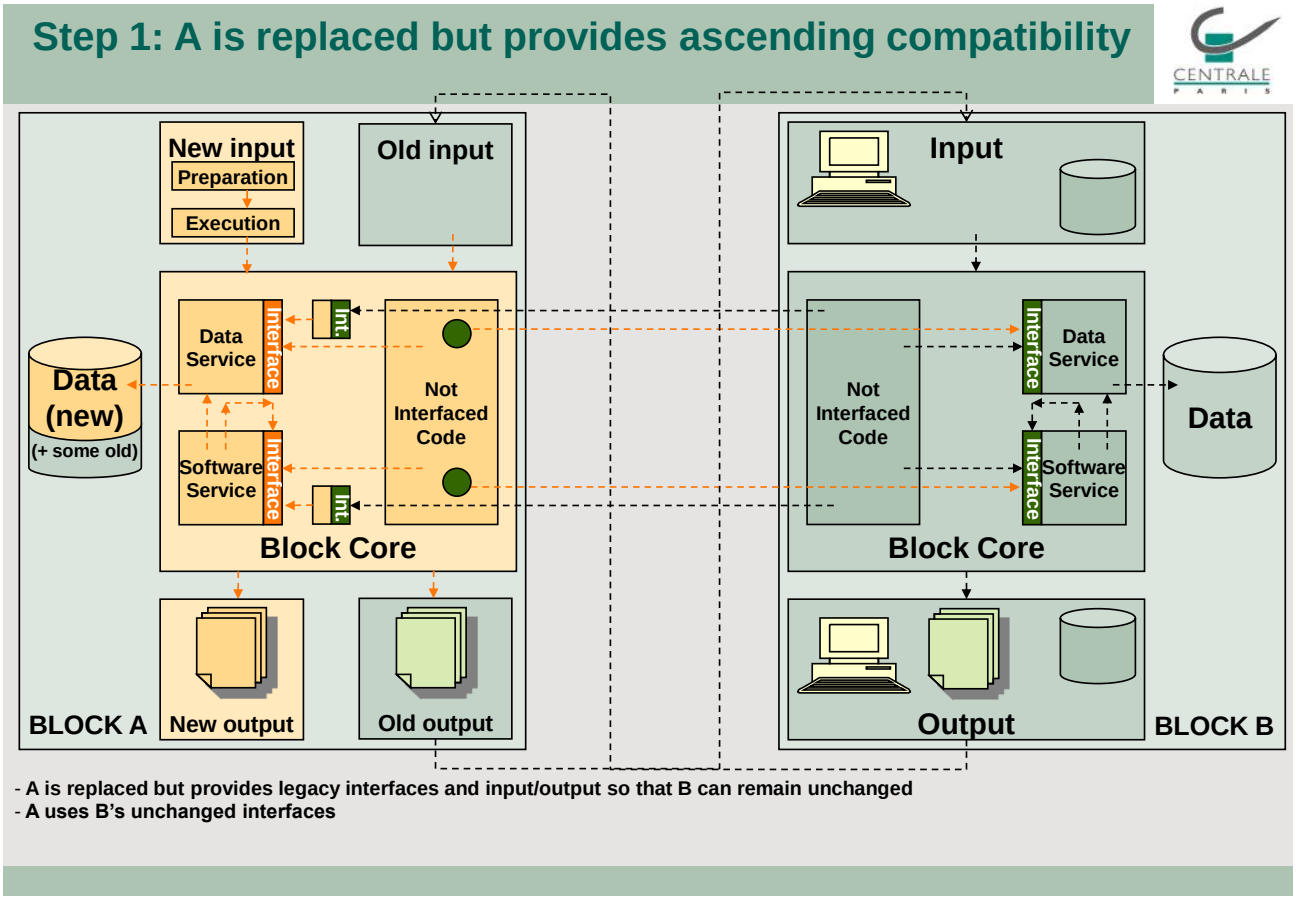


Figure 24: A is replaced but provides ascending compatibility

The **legacy interfaces of block A should be considered as deprecated**: new applications should not use them and old applications are encouraged to move to the new interfaces, but this can be done at their own rhythm, on their own budget and technical agenda.

Providing legacy interfaces puts constraints on A: for example, A might need to maintain some legacy data to convert new data in a legacy format (Mapping a new universal ID into an old organization-dependent ID for example), or it might need to be able to do automated transformations when constraints have been released on new data (like Windows transforms long file names into DOS compatible 8.3 file names).

Since these limitations are expected to disappear, **the code that maps old interfaces to new interfaces or old data to new data should be cleanly isolated** so that it can be removed when the old interfaces are not used anymore.

Sometimes, it is even possible to isolate the mapping code within the middleware: connectors or transformation engines within the EAI or the ESB handle the transformation job between the new data or service contract offered by A and the old data or service contract expected by B (See Chapter 4.8).

Step 1 is the **final step when Block B is outside of the scope and should not handle changes**.

> *A **variant of Step 1** can be used when B is in the scope. It **consists in adding B's new interfaces** (with light implementation) **to Block B before replacing Block A**. This variant can be used when B is transformed rather than replaced. It has the advantage of delivering directly a version of Block A which will not need to be adapted to B's future interfaces.*

Even when B is eventually replaced, it can make sense to go through this transformation if new interfaces are much more convenient to call from A.

*CEISAR advises to **use this variant when A and B are simplified in the same phase and one has clear ideas about B's future interfaces**.*

In some cases (especially when a software package is used for replacement), it is more complicated to provide legacy interfaces that to adapt peripheral blocks to the new interfaces. In this case, step 1 is ignored and one goes directly to step 2. However, doing that is more risky as the scope of changes is wider (all peripheral systems have to be adapted beforehand) and defects are harder to isolate. As a consequence, skipping step 1 should be considered very cautiously.

Step 2 consists in **modifying Block B so that it uses Block A's new interfaces** rather than the deprecated interfaces (Figure 25).

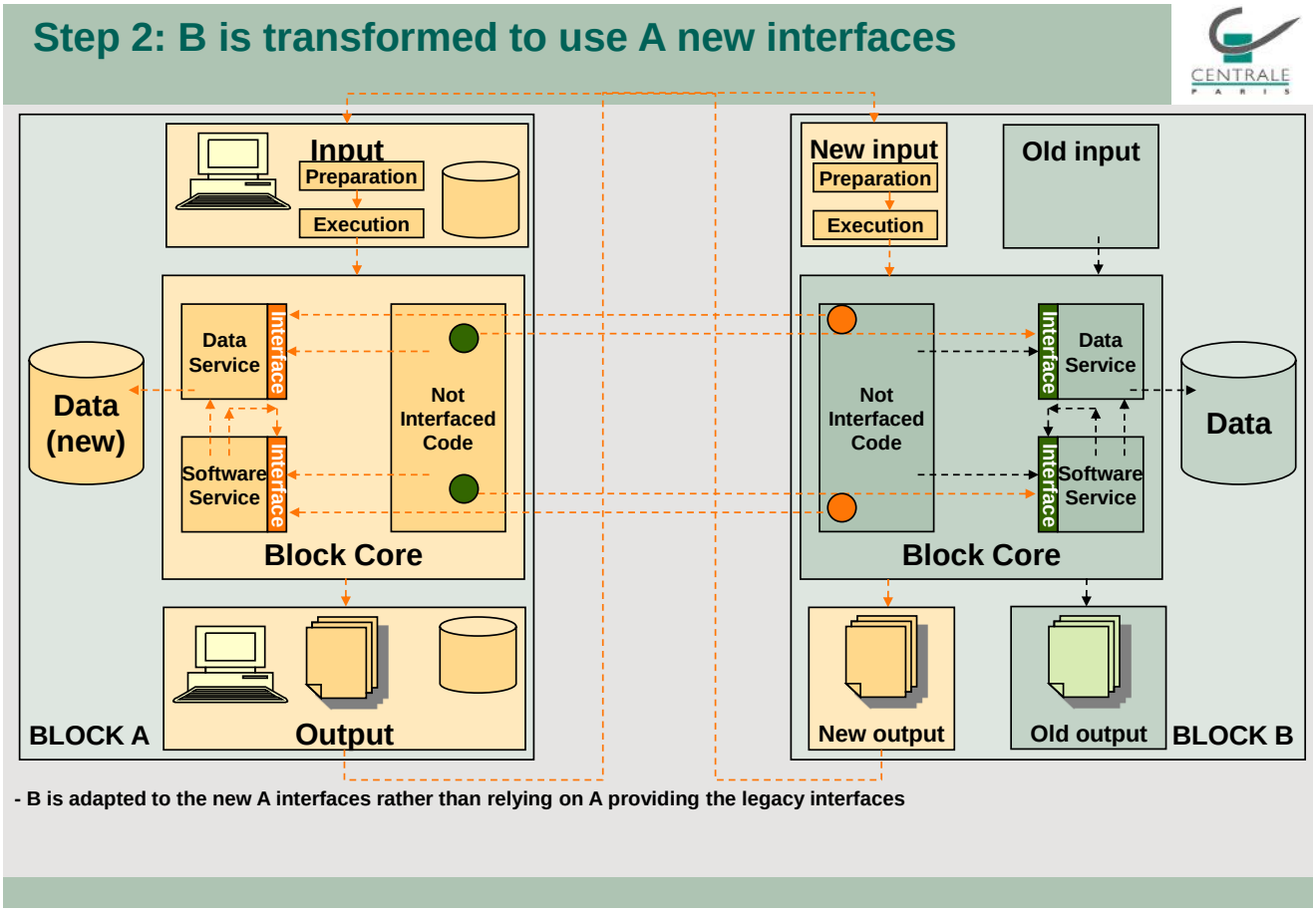


Figure 25: Block B is transformed to use A new interfaces

Step 2 is a better final point than Step 1 as the situation is simplified: Block A is fully renewed with no trace of legacy constraints or legacy code. Mapping code between old and new interfaces, old and new input/output and old and new data can be removed. **Block A is slimmed down and more coherent**.

The disadvantage of Step 2 compared to Step 1 is that **it requires some transformation of Block B**, which is not always possible.

Block B transformation should remain limited if B is outside of the scope or to be replaced. If this is not the case, then CEISAR advises to stop at Step 1 if Block B is outside of the scope, or to go directly to Step 3 if Block B is to be replaced.

Step 3 consists in **replacing or transforming Block B** (Figure 26). Block B now presents the new interfaces and **Block A is modified to use these new interfaces**, if not done already (Step 1 variant).

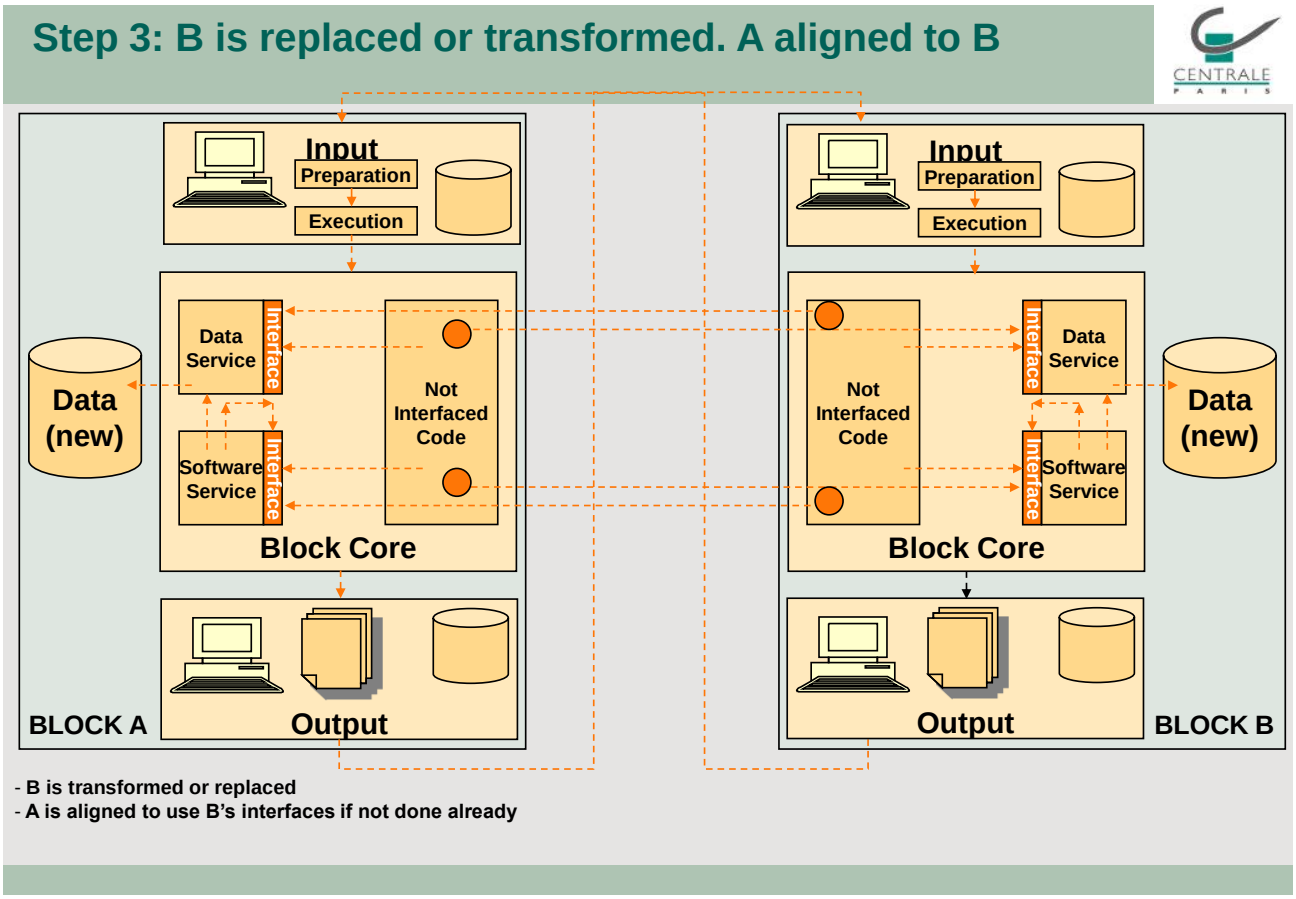


Figure 26: Block A and B have been renewed

3.3.2 Big Bang Approach

Now that we have seen the Replacement process, let's examine execution strategies.

In the Big Bang approach, a Block is replaced totally and in a single shot: on a given date, users switch to the new System and the old one is shut-down simultaneously (or at least closed: it might still be running without connections and treatments for users to consult dead data).

Although most companies say that do not want to go Big-Bang, most end-up doing it because it is the simplest replacement strategy and because it is often the only solution when a block is replaced by a monolithic package.

Functional and Architecture Constraints

As the replacement of the old block is total and one-shot, the Big-Bang strategy **does not impose any functional constraints** on the new System, except sometimes when legacy interfaces with peripheral blocks have to be maintained.

The Big-Bang strategy **does not impose any architectural constraints** on the new block either (that is why it is often the preferred strategy, especially with packages). Again, limited architecture constraints can be found on interfaces if legacy interfaces must be maintained by the new Block.

Old System Prerequisite

A key advantage of this solution is that it **does not require any transformation of the old System that will be replaced** (except to avoid scope propagation; see Chapter 3.1), and little knowledge of its technical issues. For this reason, it is an interesting solution when technical knowledge has been lost or when any modification would be dangerous.

On the other hand, a **very good and detailed understanding of the functions of the old System and of its interfaces is required** as the new system will replace the old one in one shot and with **no way to go back**.

Data and Interface Migration

With this solution, **data migration needs only to be one way** (from the old System to the new) and can be a single one-shot operation. As a consequence, the new data does not need to maintain legacy properties (a legacy ID for example in addition to the new ID) or respect legacy constraints (e.g. a limitation on a field size or malformed integrity constraint). This is true however only if the new System does not need to provide legacy interfaces.

Being one-shot also gives a lot of flexibility in the execution of the data migration. **It does not need to be perfectly automated**: the 5% of cases that are difficult or impossible to handle automatically can be just logged and resolved manually during the data cleansing of the migration phase.

> ***In some cases*** *(packages in particular) the* ***replacement solution cannot implement legacy interfaces****. In this case, all peripheral systems must be modified to adapt to the new System and changes must go live at the same time the new System goes into production. This* ***increases the risk****,* ***adds up to the upfront cost*** *of this strategy and* ***mitigates its speed advantage****. Consequently, CEISAR does not advise to use this strategy when the renewed Block is used by many peripheral systems while legacy interfaces cannot be supported.*

Execution and Deployment

The Big Bang approach can be **fast** (little engineering work) but **requires a lot of upfront investment** before the first delivery (detailed functional analysis, interfaces migration) and, as it name implies, **does not leave room for much phasing**.

Consequently, because of its lack of progressiveness, it requires **great care in the specification and testing phases** (in particular in pre-production environment).

Often **a pilot phase can be organized around a subset of the Enterprise data or organization**: for example, operations of organization X will switch to the new System while operations of the rest of the organization remain in the older System for a while. One must of course find data or organization subsets that are mostly independent (as consistency for shared information will have to be maintained by hand in the two Systems).

For critical Systems, some companies will start with a dual execution during the pilot phase: users will enter dual inputs in the old and new System (running independently) for a while until they are sure that the new System is ok. As it is very labour intensive, **this practice must be very limited in time** however.

3.3.3 Parallel Execution

In the parallel execution approach, there is a transition phase where two instances of the same Block (the old one and the new one) run in parallel an in sync. This implies a more complex migration path with transition code to synchronize both Systems, but it reduces the risk and allows more progressiveness in the phasing and change management.

Functional and Architecture Constraints

In the Big Bang approach, **functional constraints only arise from possible support for legacy interfaces**. Parallel Execution works exactly the same way except that it introduces **one additional interface with the old System being replaced** during the transition phase.

The **Parallel Execution** of the new and old System **imposes some architecture constraints** to the new System. In particular, it is important that the new System uses internally a data & service access layer which allows to update the old System as well as the new System during the transition phase.

Figure 27 shows the principle architecture of the Parallel Execution. The **interface between the old and new System** is critical because it **must insure that both Systems remain** in sync without any

discrepancies. **It is complex** because it does two-way transformation: from the old System to the new one and vice versa and implies data replication (See Chapter 4.6).

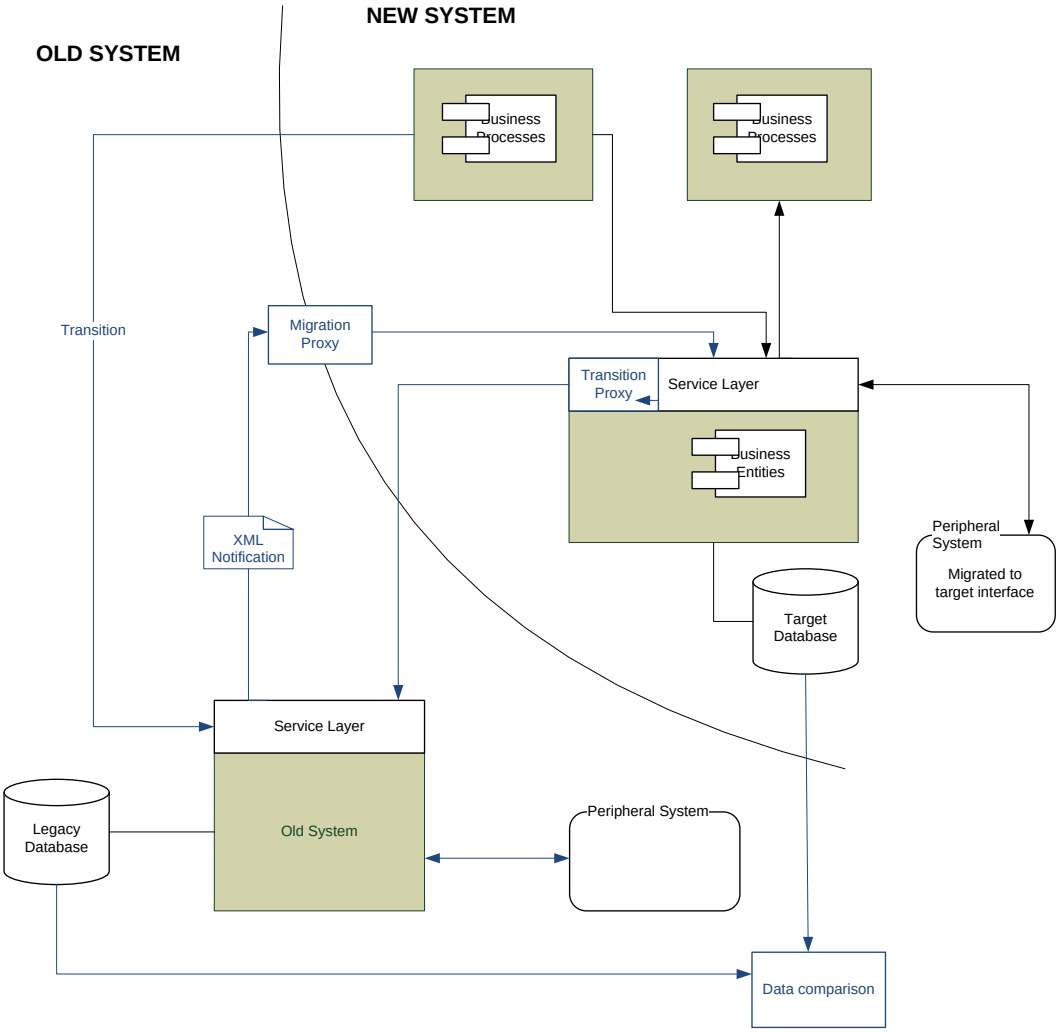


Figure 27: Architecture of the parallel execution

Since parallel execution is just a transition phase, it is important to isolate all legacy constraints and legacy migration code, preferably outside of the new system and in the middleware (migration proxies, ESB, etc.) sitting between the old and the new System.

> *Business rules will inevitably be duplicated since the new System's implementation must replace the old one. However, Parallel Execution can help testing the new rules: before a new rule is activated and goes live in the new System, it can be silently executed in the new System with live production data but without updating the master data (doing local updates in the new System for testing purposes). Automated daily batch can compare the new local results with legacy data and log-out all differences for debugging analysis. Once the new rules are validated, they are activated: they update the master System and the new data repository can be used by peripheral Systems.*

Old System Prerequisite

The prerequisites on the old System are that it provides services (see Chapter 4.1), including updates and notifications (if the old System is real-time). Otherwise, **little or no modification is necessary**.

Compared to the Big-Bang approach, Parallel Execution allows more **phasing** and **makes it easier to progressively (re-)understand the old System**. Also, it allows to **mix and reuse existing functions** during the transition phase (not unlike the Transformation strategy) through direct user access, webization, service calls, etc.

Data and Interface Migration

In the Parallel Execution scheme, **data migration is** a **complex** issue. It needs to be **done in real-time** (usually) and, during the transition phase, the **transformed data must offer a compatibility path to the old System**, as it will be used to update it from the new System.

Since the old System is still running for a while during the transition phase, one **does not have to tackle the** issue of the **interfaces with the peripheral** systems **upfront**. This is convenient when the replaced block is interfaced to many peripheral Systems: it is easier to handle the single interface with the System being replaced than to provide legacy interfaces for all the peripheral Systems or ask them to adapt.

> *Of course, one will eventually have to address these interface migrations but the parallel execution phase will give time for that and testing will be easier because it can be done against a real running System. Also, as time goes, it is likely that some peripheral Systems be replaced by added Functions of the new System (such as a state-of-the-art reporting function replacing predefined peripheral batch reports), making migration unnecessary in the end.*
>
> *And more importantly, **postponing this work can have a very positive impact on the ROI**.*

Execution and Deployment

The main advantage of the Parallel Execution strategy is to **reduce the operational risk**: since the old System is mostly unchanged, and since the new and old Systems run in parallel and in sync, the **old System can be trusted and used as a fallback any time** during the transition phase, without any data or work loss. This is a **key argument for critical systems handling very operational activities** (such as the systems that handle day-of-ops aircraft and crew control of a major airline).

The other advantage is in the execution. Since the new System comes in addition to the old one during the transition phase, the new System does not need to be complete when it goes live: **it can be phased and delivered progressively, yielding business benefits earlier** and bringing good visibility to the project.

To summarize, Figure 28 compares the Big-Bang approach with the Parallel Execution approach.

Big-Bang vs Parallel Execution

	Big Bang **One-shot replacement of old System**	Parallel Execution **New and old Systems run in sync during transition phase**
Advised When	New System is a package	System handles critical operation
	System doesn't have too many interfaces Peripherals can be converted	System is interfaced with many peripherals that are slow to convert
	Functional understanding of old System is good	Functional understanding of old System is poor
Properties	**Simple** (but cutover must be handled with care)	**Complex** (two-way data conversion between old and new System)
	Fast (but lot of upfront investment)	**Slow** Progressive with early business benefits
	Risky: no fallback	**Safe**: old System can be used as a fallback anytime without loss of data
	Change management is **brutal**	Change management is **progressive**
		Old System must have services including event notifications (or data replication)

Figure 28: Big-bang vs Parallel Execution

4 Questions

4.1 Extension of an Existing System?

The easiest way to offer new Services like web applications is to build extensions connected to the old System (Figure 29).

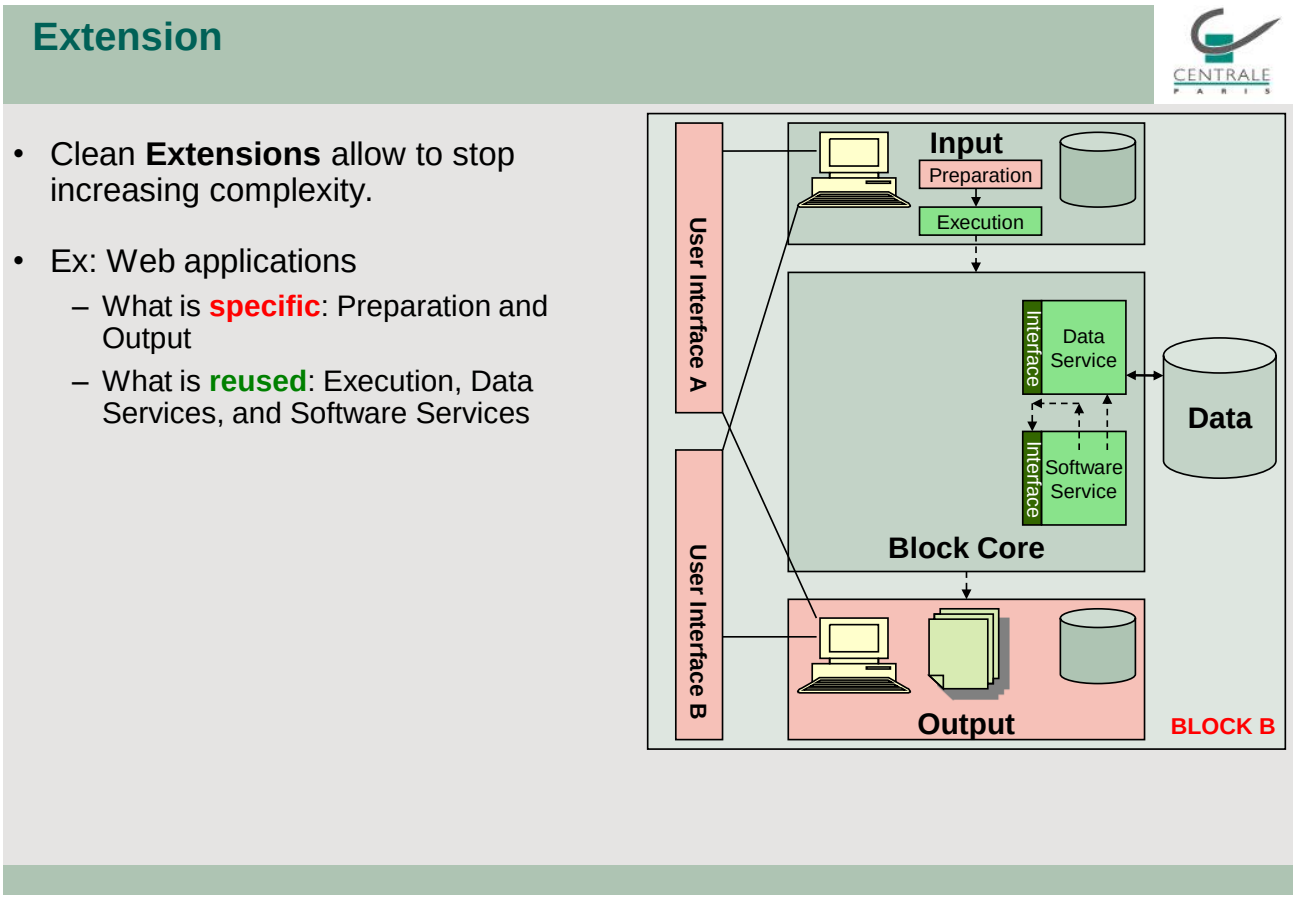


Figure 29: Extending an old system

Allowing extensions to the old System does not simplify it per se, but it allows to stop investing in it with its possibly outdated technology/competency, thus reducing its proportional share in the information system. As such, it is already a first step towards simplification.

Also, applying the extension strategy and adding Services to the old System is **often a prerequisite and the first steps of the Replacement and Transformation strategies**.

If the old System cannot support Services (ex: old packages), you must replicate its data (See chapter 4.6) and build Services on top of the replica or switch to a Replacement Strategy.

A **common mistake is to define Services dedicated to the "extension system"** that need them, without further thought on the definition of the Business Entities of the domain, on future reuse, or on the presentation of the services. Although it can sometimes be an efficient strategy (limited tactical extension, blitz migration), in most cases it will bring **more complexity** to the system. Indeed, it introduces additional interfaces that must be maintained and additional cross dependencies between the different pieces of the information system that will make it harder to simplify or migrate it later.

The ideal situation is to define Services on top of the old System with the same model, design and standards that one would have used for Services offered by the new architecture (See chapter 2.4). By doing this, one introduces a **steady Service layer** which reduces the need for custom one-to-one

interfaces between systems and **which can later act as a switchboard to move away from the old System**.

The main asset of the old System is usually the data. Consequently, data access services (CRUD) are a good first step to extend the old System. They are usually the easiest to define both functionally and technically. Yet, some care must be taken in their definition, to move away as much as possible from raw data to a proper Business Entity model.

Ideally, the standards of the new IT architecture should be respected (Web Services, etc.). They should preferably be built on industry standard (SOAP, etc.). Often, they will be built on technical layers (XML, HTTP) which are ubiquitous and can easily be implemented on most platforms with most programming languages. In any case, the **key effort on the presentation of the services should be on the payload** which contains all the business and functional value of the message. The payload should **preferably** be **defined with XML**. XML is ubiquitous and provides good interoperability (since it is text base). And above all, it is extensible and **allows content evolution without disturbing existing systems** (if properly managed), which is a good architecture property to limit systems coupling. In addition, XML is the supported standard of many tools that help introducing Services in an existing system (See Chapter 4.8), and can be understood by many off-the-shelf packages that you might want to hook-up to these Services.

The way to develop the Services will depend heavily on the platform (tools and middleware offering available), on internal skills available, and on the risks associated with modifying the old System (Table 2).

One should also distinguish between three main types of Services: read-only queries initiated from the extensions, updates initiated from the extensions and asynchronous event notifications coming from the old System to the extensions.

Table 2: Adding services to a legacy

	Legacy Modification Possible	Legacy Modification Impossible
Queries	Custom development	Screen scrapping, EAI/SOA/ESB with ad-hoc (screens, transactions, DB) connectors, Replication + query on replica
Updates	Custom development	Screen scrapping, EAI/SOA/ESB with ad-hoc (screens, transactions, DB) connectors
Notifications	Custom development, DB triggers	Websphere Information Integration, Data mirror, Teal-time ETLs, DB triggers

Read-only queries are the easiest to implement because data can be exposed from various sources: from a transaction, from a program, or from custom access to the database. Many tools can help build such queries (See Chapter 4.8) and custom development of read-only queries is usually not too difficult and very unlikely to destabilize the System.

Update queries are trickier to implement because they have to go through a control point (Input Block preferably) which insures the integrity of the update and triggers all consequent modifications (consequent computations, updates and notifications). This single point of update usually exists in the old System but not always.

Unlike read-only services, respecting this single point of update excludes almost all solutions based on data replication.

Yet, the most complicated Services to implement are the asynchronous notifications (Outputs Blocks) because they usually have to trap and handle events in multiple parts of the old System. Documentation and knowledge is often lacking, making it difficult to trap all the events in the legacy code. Also, adding notifications can destabilize the old System (impact on performance, reliability, etc.).

For these reasons, if notifications do not pre-exist, a non intrusive approach supported by specialized tools, such as database triggers, database log monitoring, etc. is advised (See Chapters 4.6 and 4.8).

4.2 What if Different Entity Model?

What if Entity Model is different ?

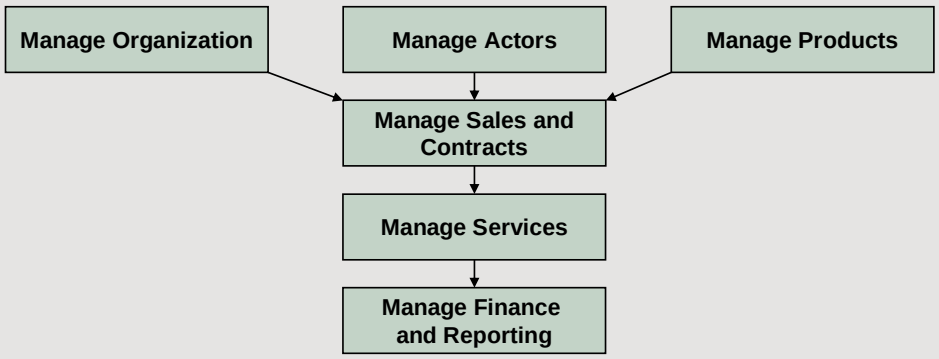
- If some **types** have changed: Conversion Services needed
- If some **attributes** have been added to new Entity: depending on the transformation strategy, may require to add a new Data Base with new attributes only
- If a new Entity groups **several** old entities: easy problem
  - Example: old « Region » and « Branch » replaced by « Organization Unit »
- If an old Entity is divided into **several** new Entities: a cache can be useful to optimize performance
  - Example: old « Person-Address » replaced by « Person » and « Address »
  - Example: old “Contract” replaced by “Subscriber” and “Contract”
- If the Entities are not the same, it becomes more difficult: to be solved case by case
  - Example: “branch-client” replaced by “global-client”
    - Which id ?
    - Requires software adaptations

4.3 What if Output Blocks First?

What if Output Blocks first ?

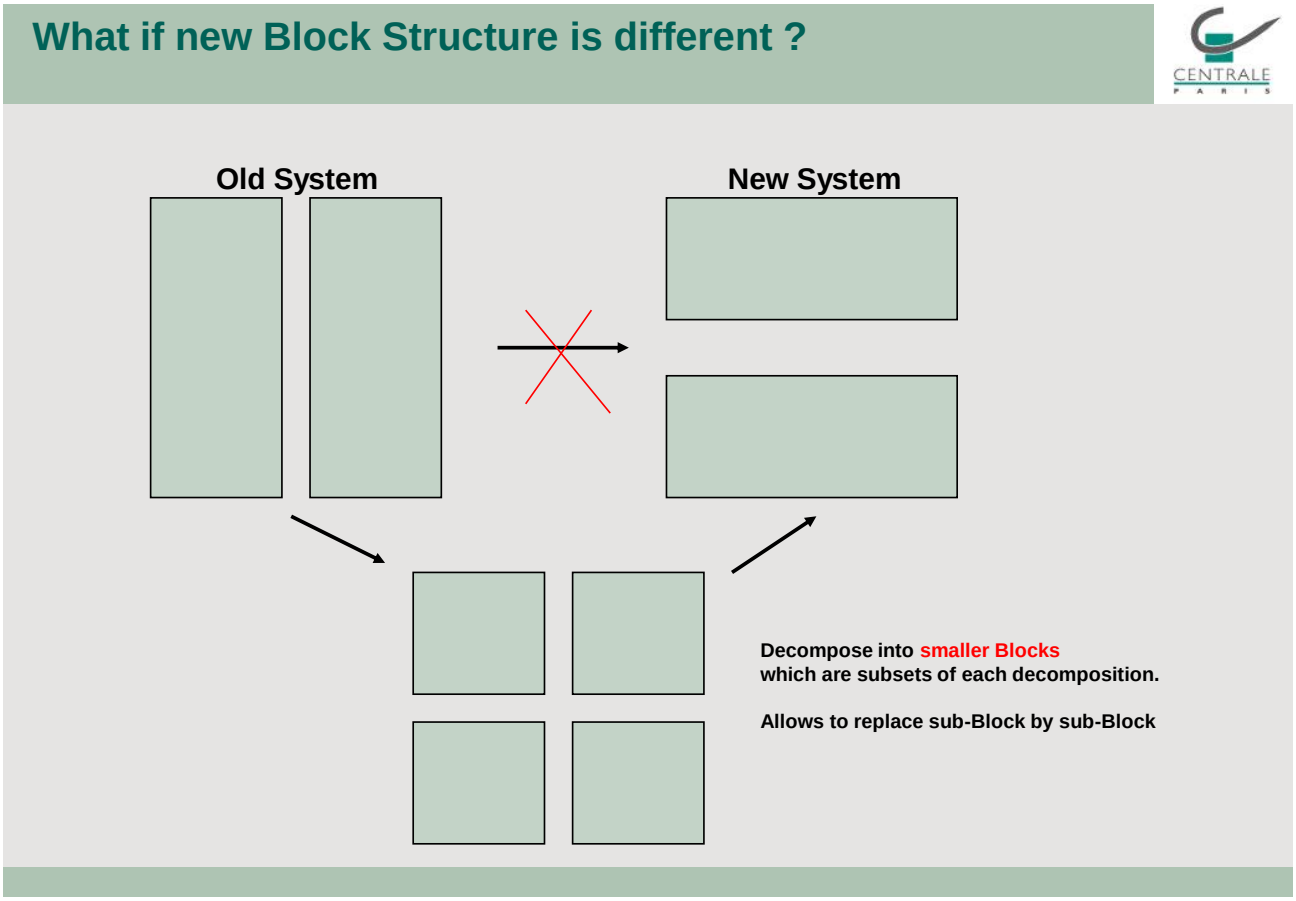
- If you are free to deliver Blocks in the order you want, **start with Input Blocks** which will deliver more complete data, and go on in the data flow order.

Example:



- But if a **business constraint** imposes to deliver Output Blocks first, then :
  - New data will not be available in these new Blocks: they will require to be adapted after upstream Blocks will deliver the new data.
  - New Blocks must be more permissive if data is lacking

4.4 What if Block Structure is different?



4.5 What About Conversion?

Conversion consists in translating a system from one technology (e.g. COBOL/CICS) to another (e.g. Java/J2EE).

Conversion is a Replacement Strategy that can be used if a Block operates under an obsolete technology and is well structured (otherwise conversion will fail).

Conversion is not used very often because it can hardly be automated (converting a COBOL program to Java will lead to a Java program that looks like COBOL and which will be hard to maintain), and its benefits are limited (new System will have the same limitations of the old one, except for the programming language).

Some software houses offer conversion services where Systems are entirely re-written. This is a labor intensive process but the converted Systems respect the best practices of the target environment (i.e. object oriented concepts for a Java target) and are easier to maintain.

4.6 What About Data Replication?

Although every architecture book discourages the use of Data Replication, it is a technique which must be known as it is often used in simplification projects. In particular, it is used in any Transformation or Replacement project which requires (at least temporarily) parallel execution of the old and new System with access on the same data.

Simple Data Replication without model translations is well supported by databases and tools. It does not present any challenge, but is used (in our case) only for re-hosting.

When we look at legacy simplification, we most likely want to **do model transformation when we replicate** data from the old System to the new System. There are several ways to do that:

- If the old System always writes its data through a data access layer, then it is possible to **modify the data access layer** to write in the old database as well as in the new database. But that implies that modifying the old System is ok (stability and performance are at risk) and that it has enough information to write data in the new form.
- A popular alternative, especially if the old System does not always use a data-access layer, is to use database triggers. **Database triggers are less intrusive** because they do not imply code modification, **but they can affect performances**. For this reason, and for improved flexibility, it is advised that the triggers only track database modifications and send out messages. These messages are then handled by the replicated system which does model translation (possibly using additional data only available on the new System) and writes down the replicated data in the new System.
- A variant of database triggers is the use of **specialized data-integration tools** (See Chapter 4.8) that produce event messages (usually XML-based and sent on a message queue) based on database changes. These tools are non intrusive and do not affect performances because they run outside of the database (usually using log scrapping).
- If the data do not need to be replicated in real-time, then scripts or preferably ETL tools offer all the flexibility to do data replication with model transformation properly. **Real-time ETL tools are arriving on the marketplace** and can also prove to be a good solution, even for real-time updates.

In any case, access to replicated data should only be read only. Synchronizing two data repository that can be updated separately proves to be very complicated (even though techniques such as tokens exist). It is consequently advised to **respect a master-slave mechanism** where the data-access layer of the new system forwards updates to the old System and awaits acknowledgement before committing them on the replica. It is only when updates from the old System are phased-out that the new data can become the master.

Note that with off-the-shelf packages, this rule can sometimes be hard to obey (the package will likely commit changes in its own database before receiving an acknowledgement from the external System). One must then implement data reconciliation techniques that are more complicated and not as safe.

4.7 Parameters and Rule Engines?

One strategy may consist in focusing first on what changes often: the idea is not to progressively replace the old System; it is just to extract the part which requires reactivity. Let's call it "**flexible part**". This flexible part can include data and/or rules.

To give examples:

- to increase "time to market", the company must be able to quickly modify existing products or create new products
- to modify commission system, the company must be able to quickly modify commission parameters or rules

Modifying existing products means modifying the flexible part composed of **parameters** or **rules** which define Pricing or Eligibility rules.

When the old Systems are complex, it is very difficult to directly update them because parameters and rules are spread out among many different tables and modules.

Creating new products follows the same principles: the flexible part must allow assembling new products by reusing product parts. Then new parameters and rules are embedded in product definition.

When the Flexible Part is **data** driven, one efficient way is the following **non intrusive** approach:

- Define the Business model and the class model
- Import data from the old System
- Offer visibility on products
- Offer a tool to modify or create products

- Export data to old Systems

If the part is **rule** driven, it is more difficult to do because representation of rules depends on the architectures of the old System and the new System. If architecture are different, it is more efficient to replace flexible rules by a rule engine (see **White Paper on Rule Engines**).

4.8 Which Middleware and Tools?

Complexity Analysis

Many tools can provide **complexity metrics** (cyclomatic complexity, modules coupling, etc.) through static code analysis: some are software packages (Cast, Relativity, McCabe…), others are consultants' tools (Qualixo, Isoscope…), others are open-source tools (Eclipse Metrics, CheckStyle…).

They differ by the number of languages they support (Cobol, C, C++, Java, ABAP, etc.) and how they support system-wide technologies (giving a global view of a HTML-JSP/J2EE/CICS/COBOL system for example) or architecture metrics (tier access violations for example).

Most of these tools can provide **system cartography** with links and coupling between modules, and can trace data flow (from database columns to programs' usage). They usually generate a HTML map of the System which allows simple navigation. This gives very interesting insight into the System when one wants to find-out dependencies, analyse impact or wants to regain functional knowledge.

Some of these tools (Relativity, Asetechs…) offer **remediation workbenches** which can do refactoring (remove dead code, reorganize flow controls, normalize code…), slice code (to isolate data access in separate programs for example), and provide base documentation (UML artefacts) which can then be enriched by an expert.

Data Replication / Data Integration

Many **ETL** vendors (Informatica, Cognos, Business Objects/Acta…) provide very robust tools which can extract data from different referentials (files, hierarchical or relational databases, SAP, etc.), transform them (merges, cleansing, transformation) and load the results into the target replicated database. These tools handle large volumes of data (they have been typically used for datawarehouses) and usually run as batches. They can be used for data replication but only when real-time updates are not required.

Some companies have recently moved towards **real-time ETL** (DataMirror, IBM Information Server…). Their products are non intrusive: they can monitor database changes (using various techniques such as log scrapping), extract the changed data, transform them and update a target database. Alternatively, they can generate XML messages, transform them and send them on a queue based on configurable rules.

These tools can be used for **data replication as well as data integration**. They come close to the EAI in terms of message generation, transformation and routing, but they do not invoke services or transactions (except for database updates).

Business Integration

Webization tools permit to expose legacy screens (usually mainframe) in a HTML format. This allows UI-level integration within modern applications (portal or other Intranet tools).

EAI (Enterprise Application Integration) tools offer connectors to expose legacy data (databases, mainframe transactions, SAP, etc.) and events, and send them to an engine which transforms and routes them to other applications, thus providing process and business integration.

SOA differs from the EAI concepts by focusing on Services rather than data, and by leveraging open standards (WS-* web services standards, SOAP, etc.). Generic Business Process orchestration engine allow to implement the process orchestration, i.e. invocation of web services following process logic, with an open high-level (possibly graphical) language (**BPEL**).

ESB are a mix of open-standard based EAI and SOA, combining web services, messaging middleware, intelligent routing and transformation. ESB provide WS-* compliant connectors to call and expose web

services and offer asynchronous messaging facilities to transport data (usually under an XML format) with configurable routing and transformation engine (we talk of EDA: Event Driver Architecture). Java-based ESB usually comply with a standard (called JBI) which allows connectors and transformation engines to be independent of the ESB vendor.

Major SOA/EAI/ESB vendors are IBM, Microsoft, SeeBeyond (Sun), Tibco and WebMethods (Software AG).

5 Case Studies Abstracts

5.1 Axa

AXA has defined a **strong governance model** for legacy simplification with the objective to incorporate complexity decrease in all IT projects and activities.

Every IT unit must include convergence & simplification in their annual IT strategic plan. This initiative is supported by template business case and a standard approach for application selection (aligned to business priorities).

Simplification is mostly justified by the reduction of soft costs, i.e. impact of complexity on business efficiency, which are not easy to justify. Consequently, **business must be strongly involved**.

Execution is primarily opportunistic and done during functional upgrades of strategic business applications.

AXA has started experimenting with **legacy complexity analysis tools** and has found them valuable.

5.2 BNP Paribas

BNPP's main approach for simplification is **Legacy Extension** with an important effort to expose data access services to external applications.

Common Business Entities have been defined with attention paid to be organization-independent. Then, in 10 years, most applications have been modified to expose or use **~1000 basic Data Interfaces that have been defined with IDL**. Client proxies and server stubs are automatically generated in different languages/technologies (COBOL, Java, ActiveX) thus facilitating integration.

BNPP maintains a catalog of ~2000 reusable components: Entity access but also Business Functions (data checks, risk evaluation, etc.).

Each Business Domain (Credit, Insurance, Securities) owns its own Information System with **clear data ownership responsibilities** and subscription mechanism for external systems/application.

Replacement was chosen for Payroll (HR Access) and Human Resources (SAP).

Transformation is considered very difficult and used only for the most complex IT systems.

5.3 Michelin

Michelin is simplifying its legacy information system by standardizing on worldwide "**master applications**" for each Business Domain. This is **done in parallel to streamlining the processes** and making them common in the different countries.

For that matter, Michelin has set-up a strong governance model where each domain's IS is governed by a committee whose chairman comes from the business but is involved with legacy obsolescence issues.

Michelin is currently studying its simplification approach for their order-to-invoice system. After a very **thorough examination of the legacy** system (an in-house 4.5 million LOC COBOL system), including **block cartography and complexity analysis** (with metrics), Michelin is leaning towards a **Transformation Strategy** where **Data Access and reusable "Business Engines" will be isolated and exposed as Web Services**. Process logic will be rewritten with a **Business Process Execution Engine** which will offer the flexibility required by the business.

In some cases (end of vendor's support on a technology/product), Michelin subcontracts **application rewrite**. In this case, they are always done without any functional modification.

5.4 Total

SAP Simplification

After the merger of Total-Fina with Elf Aquitaine, the company standardized its ERP on SAP. Because of the number of affiliate companies and needs for IT environments (development, QA, pre-production…), 44 SAP instances were needed for small and medium companies.

In 2006, Total launched a project to **simplify its SAP infrastructure** and reduce its number of templates. The project reduced IT costs by reducing the number of maintenance and operations activities and reducing the number of servers (Servers and SAP transports have been reduced by 40%). ROI should be met in 1.5 years, proving **there can be a case for IT-based ROI** when the simplification scope is well defined.

Environments were **compared and merged two by two**, starting with the simplest and closest to best practices. Each step was very short to complete (10 weeks).

Identity Directory renewal

Total is **moving the identity and organization logic away from the applications** into a central Identity Directory, which will help streamline and automate processes (when someone arrives or leaves the company) and reduce maintenance costs. This central Identity Repository will expose its shared Business Functions through web services.

The new directory is going to replace the old one. It is **deployed progressively and runs in parallel** to the old one during a transition phase. At first, it handles only new functions and supports a limited subset of users who need these new functions. Then, **usage grows in terms of implemented functions and concerned users**, until the new System can replace the old one.

Applications are **migrated by clusters defined by a dependency analysis** on Block cartography. For each application, migration to the new System is big bang with extensive regression testing beforehand.