Software Architecture of an Insurance System based on Services

André Filipe Alves Viegas

Thesis to obtain the Master of Science Degree in Information Systems and Computer Engineering

Supervisors: Prof. António Manuel Ferreira Rito da Silva
Eng. Pedro Magalhães Lemos

Examination Committee

Chairperson: Prof. Miguel Nuno Dias Alves Pupo Correia
Supervisor: Prof. António Manuel Ferreira Rito da Silva
Member of the Committee: Prof. José Manuel Nunes Salvador Tribolet

November 2015
Aknowledgments

I would like to thank my parents and grandparents for never allowing me to let my academic education be left unfinished. I also would like to thank Pedro Lemos, and Deloitte, for supporting me in this work. And finally, I thank my girlfriend Carolina Costa for all the patience and support.

Lisboa, November 20, 2015
André Viegas
Resumo

Este documento apresenta um Caso de Estudo relacionado com a transformação da arquitetura de uma companhia de Seguros, em especial com a arquitetura de alguns dos seus sistemas. Este documento foca-se na apresentação dos desafios apresentados aquando do desenho das novas arquiteturas, na apresentação dos atributos de qualidade tanto de negócio como de sistema, e das suas instancições em cenários concretos. São também apresentadas táticas para alcançar os cenários previamente definidos. Por fim, é feita uma análise do estado atual da companhia de Seguros, dos ganhos obtidos, e uma demonstração de desafios que acabaram por aparecer devido às decisões tomadas no decorrer da transformação da seguradora, pois o desenho de arquiteturas de software é um processo cíclico e contínuo, que resulta num caminho que se faz caminhando.
Abstract

This document presents a Case Study regarding a Transformational programme implemented by an Insurance company, with special focus on its architecture, and the architecture of some of its landscape’s systems. This document focus on the challenges presented when the new architectures were designed, in its business and system quality attributes, and how they were instantiated in concrete scenarios. It is also presented the tactics used to achieve the intended quality attributes of the previously defined scenarios. At last, an analysis of the current state of the Insurance company is done, presenting benefits from this Transformational programme, and also new challenges that were raised due to the decisions taken and reported in this document, because in the end, designing software architectures is a continuous, cyclic process, which results in a path that is made by walking.
Palavras Chave

Arquitectura de Software
Arquitectura Orientada a Serviços - SOA
Atributos de Qualidade Arquitecturais
Enterprise Service Bus - ESB
Keywords

Software Architecture
Service Oriented Architecture - SOA
Architectural Quality Attributes
Enterprise Service Base - ESB
Index

1 Introduction 1
  1.1 Transformational Programme 1
  1.2 Document's purpose 2
  1.3 Document's structure 2
  1.4 Methodology 3
  1.5 Assumptions and Restrictions 4

2 Challenges 5
  2.1 Falling Behind 5
  2.2 Starting with the wrong foot 6
  2.3 Finding the right path 7
  2.4 Business Qualities - what was intended 7

3 Designing InsuranceCorp's Architecture 8
  3.1 Easiness to add/modify an Insurance product 8
    3.1.1 How it relates with other qualities 8
    3.1.2 Scenarios 9
    3.1.3 How it was achieved 10
  3.2 New and old systems must be integrated 12
    3.2.1 How it relates with other qualities 12
    3.2.2 Scenario 1 12
    3.2.3 Tactic applied 12
      3.2.3.1 Service-Oriented Architecture 13
      3.2.3.2 Enterprise Service Bus 14
    3.2.4 Scenario 2 15
    3.2.5 Tactic applied 17
      3.2.5.1 IAA - The Canonical Insurance framework 17
  3.3 Modify/Extend the Service Catalogue 30
    3.3.1 How it relates with other qualities 30
    3.3.2 Scenarios 30
    3.3.3 Tactics applied 30
      3.3.3.1 Service Factory - Service implementation methodology 31
  3.4 Hold future load of requests 36
    3.4.1 How it relates with other qualities 36
    3.4.2 Scenario 36
    3.4.3 Tactic applied 37
## Current State

4.1 Service Catalogue ................................................................. 39  
4.2 Main Improvements Achieved ................................................. 39  
4.3 Post-Mortem Analysis ........................................................... 39  
   4.3.1 IAA’s understandability ................................................. 41  
   4.3.2 ESB services’ performance ............................................. 42  

## Conclusions

5.1 Lessons Learned and Personal Considerations .......................... 45  

Appendix A ............................................................................. 48
List of Tables

3.1 Main Criteria and Indicators for IAA's application driver choice .................................................. 24
3.2 Sample Overview table of the IAA operations to be implemented ....................................................... 25
3.3 Transformational Rules’ goals and examples ......................................................................................... 29

4.1 InsuranceCorp’s Service Catalogue ........................................................................................................ 40
4.2 Case 1 - Execution timings while retrieving an empty portfolio ............................................................. 43
4.3 Case 2 - Execution timings while retrieving the portfolio of a non-existent customer ............................... 44
Chapter 1

Introduction

The Insurance business, at first glance, may be seen as a simple one: a client pays a fee to an Insurance company in order to receive a compensation when an undesirable event occurs. As an example: when a incident occurs which produces a loss to a client business, he claims back from the company some support, usually money, to mitigate that loss.

Although some of these contracts are mandatory by law, such as Auto insurance for every vehicle on the road, the majority of the products dealt by Insurance companies are not. Those products exist because opportunities emerged from the market, in which clients needed to manage the risks related to their work and day-to-day life.

However, and since it is a profitable and proven business model, the market has a lot of competitors exploring those same opportunities. These pressures the Insurance companies to optimize their operations in order to be able to respond quicker to market opportunities and provide the best products to their clients, and do all that in a cost-efficient way.

It was this line of thought that led the Insurance company featured in this Case Study, from now on referred to as InsuranceCorp, into a big journey with the goal of optimizing its operation by doing more, cheaper and faster. To do so, the company started a Transformational programme that would led to multiple transformations in its organization.

1.1 Transformational Programme

A Transformational programme is a huge endeavour which typically impacts a multitude of levels in an organization. For InsuranceCorp, it impacted its organizational structure, its business processes, the insurance products it chose to sell, and specially the way it viewed its IT.

The whole Transformational programme was split into eight sequential releases, with a span of 12 to 18 months per release, where an incremental implementation of all the business of InsuranceCorp could be delivered and migrated to. This would require that the new systems would coexist with the old ones until a full decommission could be done. The timeline of the programme can be seen on figure 1.1.

The main strategy behind the split of the many releases was to focus first on the Commercial Lines’ products, which had a smaller and more manageable portfolio. Then, it was to increase the amount of products per release until all products (in their newer versions) were implemented in the new systems.
1.2 Document’s purpose

The main purpose of this document is to bring light to the complex process of transforming an architecture in a company with a relevant size, providing a real insight of the challenges and decisions that were taken during the first releases of the Transformational programme, focusing in the Integration work done to achieve those goals. Reveal the requirements that were needed at first, and the ones no one thought were needed upfront.

Therefore, the outcome of this Case Study is to share knowledge regarding this type of programmes, in order to understand some of the do’s and don’t’s, and best practices that can be reapplied should the same kind of challenges be presented in the future in a similar context.

1.3 Document’s structure

This Case Study is divided in 3 phases:

- **Context** - The first phase of this document provides an overview of the Insurance world, as it presents the circumstantial topics, and also the ones specific to the Insurance company that stem this Case Study. It also provides an understanding of this document. The chapters that comprise this phase are:
  - Introduction
  - Challenges

- **Report** - The second phase will provide a view of the work done in this Transformational programme regarding the architecture of InsuranceCorp;

- **Results** - The third and last phase wraps-up this Case Study, providing information regarding the current situation of the Transformational programme and InsuranceCorp, the achieved benefits, the lessons learned and some personal considerations. The chapters that make up this phase is:
  - Current State
  - Conclusions

Figure 1.1: Transformational Programme’s chronology and the focus of this Case Study
1.4 Methodology

The methodology used on this Case study follows the approach suggested by the referenced bibliography, such as the book by [Bass et al., 2003], where the architecture cycle is composed by the definition of the Business quality attributes, which will in time originate the System quality attributes intended for the architecture. Those are the drivers to design the architecture, but since those attributes may fall on generic/abstract high-level statements, they are instantiated in scenarios, as the one depicted in Figure 1.3 in order to define in a concrete manner the intended qualities of the architecture. In order to address those scenarios, the correct tactics are then chosen and implemented, so that the intended architecture can then be achieved. This described flow is described in the Figure 1.2. The architectural views presented throughout this document base themselves in the content of the book by [Garlan et al., 2010].

![Figure 1.2: Methodology used to report the architecture cycle in this Case study](image1)

![Figure 1.3: Elements of a Scenario as described in the book Bass et al., 2003](image2)
1.5 Assumptions and Restrictions

Two conflicting interests collide in this assessment: the commercial and privacy interest of the company behind the case study presented, and the academic interest of publicizing it. Taken that into consideration, some details have been omitted without prejudice to the overall case exposition. These are the assumptions, and also restrictions due to confidentiality agreements, that needed to be considered in this case study:

- The real name of the company referred in this Case Study, as the market in which it operates, will not be identified. Information that can directly, or indirectly, identify the company will be omitted;
- The real names of the applications/entities that made the company’s previous architecture will not be identified;
- Not all the technologies that comprise the new architecture will be identified;
- Details that may compromise the current architecture of the company will not be mentioned;
- No human resources from the parties involved in this Transformational programme will be identified;
- The Integration work will be the only one with a higher level of detail in terms of specification of the architecture, challenges, solutions and lessons learned;
- The architectural representations in this document will not be totally exhaustive in terms of number of systems in InsuranceCorp;
Chapter 2

Challenges

This section presents the state of InsuranceCorp, and the main issues it faced, before the start of its Transformational programme. It will also present a set of Business Qualities envisioned for its IT Architecture that would drive the decisions done throughout the programme.

Another topic addressed in this chapter is the fact that the first Release of the aforementioned programme was assigned to another Consulting company, which ended up in an unfruitful relationship that led to a partnership with Deloitte in the second release and beyond, much of it because of its different approach in terms of delivery.

2.1 Falling Behind

In 2010, the InsuranceCorp found itself in a somewhat precarious situation: its systems and processes, although working, were old and very inefficient. Not only that, but the way it worked, created a lot of entropy and obstacles to all the changes and improvements that the business required. Systems that were developed decades ago, with a multitude of “band-aids” in them through the years, made it very expensive and time-consuming to make the smallest change on the production environment.

The knowledge regarding the systems was decreasing year after year, due to retiring resources, and the documentation regarding the systems was lacking, which led to a lack of in-house knowledge in terms of development and maintenance of those systems. There was also the fact that the technology on which it was developed, COBOL on Mainframe, is a somehow scarcer skill nowadays, therefore expensive to hire. These characteristics highly compromised the buildability of the architecture. That increased the time to market and the cost of all maintenance projects, either corrective or evolutionary.

The architecture in place in InsuranceCorp, depicted in figure 2.1, was mostly sustained in Mainframe technology. There were 3 main systems, all bespoke solutions in Mainframe:

**Policy System** This system is the core system of InsuranceCorp’s business. Its responsibilities are: definition of insurance products, management of quotes and policies, and claims’ handling;

**Party System** This system is the master of Party information. Its responsibility is to maintain information of all Persons and Companies - customers, leads and interested parties;

**Account System** This system manages the accounting information generated by InsuranceCorp’s business, e.g. handling of payments, scheduling of remainders and issuing of invoices;

The architecture in place in InsuranceCorp was mostly based in Point-to-Point integration, via message queues and through batch. Each information domain resided in a separate technical component,
and in some cases, with information of the same domain being split in more than one component. Data integrity mechanisms running off-line at the end of the day, created gaps in terms of real-time access to data produced during a business day. A component-and-connector view of InsuranceCorp’s architecture before the Transformational programme can be seen in figure 2.1.

Business processes were also very coupled to the technologies supporting them: the mainframe’s UIs were used for many years, and processes, whether they were online or batch, existed depending on the way they were once implemented. Therefore, day-to-day work was very restrictive in terms of what and how things could be done, relegating the customer experience to a secondary role.

All these handicaps led to an under-performance of the company in terms of what was desirable by its Board.

2.2 Starting with the wrong foot

The first release of the programme was adjudicated to a Consulting company, from here on referenced as Company A. This ended up as a misstep by InsuranceCorp, since the lack of knowledge of the to-be implemented system, and the off-shore delivery model put in place by Company A created a lot of challenges in the delivery of what InsuranceCorp expected from the first release of its programme.

Regarding the delivery model, Company A was based off-shore, as well as their main workforce. By moving only a dozen of consultants to be close to the InsuranceCorp’s business people, they became a huge bottleneck for information on both ways, since they had to work as a proxy for every work stream of the project and all teams located off-shore.

The issues of the delivery model used, added to the lack of knowledge of the implementation of the technologies intended to be used in the programme, led to the delivery of the first release of the
programme that was not aligned with the InsuranceCorp’s business requirements, and also a technically solution that was not robust at the Integration and Core system levels - mainly due to bad coding quality.

2.3 Finding the right path

For the adjudication of the second release, InsuranceCorp wanted to go in a different direction in terms of its partner for their internal systems’ developer. By presenting a better vision of what should be the architecture for InsuranceCorp, based on prior experiences in similar clients, and also with a complete different approach in terms of the delivery model, Deloitte was hired to follow a more ambitious plan than what was initially expected: to implement the Release 2 and solve pending issues of Release 1.

The delivery model applied by Deloitte consisted in a more close relationship with the personnel of InsuranceCorp, done so by moving more than 70 consultants to the offices of InsuranceCorp. This was fundamental to gain the trust from everyone, and also to improve the communication between everyone on board of the programme.

The following chapters will present the approach and work performed by Deloitte in order to achieve the proposed objectives, focusing the work delivered at an Integration level.

2.4 Business Qualities - what was intended

In order to harness the opportunity created by the Transformational programme, InsuranceCorp decided to go through the complete architectural cycle and raised a set of non-software quality attributes that would influence all the other attributes gathered to ensure a quality final architecture. This process began with the identification of the 3 Business Qualities which provided the biggest drivers for the Transformational programme, listed in the following subsections.

Time to Market:
The easier it is to design and implement new products, the quicker is the time-to-market of it. That is specially important since it transmits to the market the image of a company that is always on top of the newest trends. There is also the need for the business to be able to easily adapt and tweak its products to incorporate new features, such as new coverages or legal compliances.

Cost and Benefit:
The margins and prices practise nowadays were already low, and with other Insurers on the market that are able to provide similar products and prices, the only way to increase profits is to assure that the business runs in the most efficient way. Therefore, InsuranceCorp were longing for an architecture that would provide the best benefits regarding its costs of implementing and maintaining it. That is something that couldn’t be achieved before the Transformational programme since the old architecture that was in place was costly to maintain, entrenched with technology which resources were hard, and expensive, to hire.

Roll-out schedule:
Systems in InsuranceCorp were outdated and in their way of being replaced, however its sunset would have to be gradual. The immense portfolio, and the fact that all their processes and business were supported on those systems, led to the need to build a new architecture that would allow the current systems to co-exist with the new ones, as those new systems would add up more functionalities, and the portfolio is migrated, product by product. The final goal is to have all those mainframe systems completely sunset and replaced with the newer systems.
Chapter 3

Designing InsuranceCorp’s Architecture

Once the Business quality attributes were set, the foundations of the architecture were in place. However, those qualities, although essential, are still in such an high-level that can’t be easily translated into design elements of an architecture. Such elements are only achieved when Business attributes are decomposed in System quality attributes, which will determine the expected behaviour and expectations from the systems at a more operational level.

This chapter will present the system quality attributes defined and taken into consideration in InsuranceCorp’s new architecture, how do those quality attributes relate to the previously identified ones and what was the approach taken for each one. A summary of this information is represented on Figure 3.1.

3.1 Easiness to add/modify an Insurance product

Modifiability System quality

As already stated before, InsuranceCorp wanted a system that would allow it to add a new product, or change an existent one, in a quicker and easier way compared with the situation it had at the moment. Before the Transformational programme, InsuranceCorp managed all their products, and their policies’ life cycle in the Policy System, (see figure 2.1), a bespoke solution with almost 30 years. The fact that almost no one in-house knew the technology (COBOL), nor was the system flexible enough to cope changes in an easy way, led to costly and long projects every time Business requested a change.

Therefore, InsuranceCorp looked for an alternative system, in order to replace their old one, that had to comply with a major quality attribute: to be easily modifiable.

3.1.1 How it relates with other qualities

This quality attribute that was desired to the System is a logical derivation from the Cost and Benefit and Time to Market Business Qualities. The desire to have a system that would allow a better and quicker response to Business requests, that would allow to perform those changes in a quick and cheaper manner, was one of the mainstays of this Transformational programme.
3.1.2 Scenarios

In order to construct a more concrete specification of this System quality attribute, the following scenarios were defined:

**Scenario 1 - Insurance Products**
One concrete scenario was created regarding the changes on the Insurance products that InsuranceCorp sold:

**Source** InsuranceCorp’s Business department;

**Stimulus** Request the addition/change of an insurance product;

**Environment** Design time;

**Artifact** Insurance Core System;

**Response** The roll-out of a new product, or a change, must be possible in a quick manner and with a short budget;

**Response Measure** The effort it takes in man/hours must be much faster than with the past architecture; The cost it takes to roll-out must be cheaper than with the past architecture;

\[^1\]Unfortunately there is no information regarding concrete metrics to measure the response of these scenarios.
Scenario 2 - Pricing changes
Another scenario was created regarding the changes in the Pricing algorithm used to calculate the premiums of the policies:

**Source** InsuranceCorp’s Business department;

**Stimulus** Reconfiguration of Pricing settings;

**Environment** Design time;

**Artifact** Insurance Core System;

**Response** The roll-out of the new Pricing algorithm configuration must be possible in a quick manner and with a short budget;

**Response Measure** The effort it takes in man/hours must be much faster than with the past architecture; The cost it takes to roll-out must be cheaper than with the past architecture;

3.1.3 How it was achieved

In order to address the previously presented scenarios, InsuranceCorp decided to break with the line of thought used when implementing their previous Core system, which was to implement a bespoke solution, tailor made for them. For this new era, they’ve decided with a new approach: use a market’s key Core system off-the-shelf (from now on referred to as TNCS - The New Core System), implemented in cheaper and contemporary technology. By using an off-the-shelf system, with its modules, processes and capabilities already pre-defined, InsuranceCorp would greatly increase its Time-to-Market quality attribute that so long desired. And since the technology used by TNCS - Java, Oracle Forms and PL/SQL - was much more spread and common nowadays, not only would the buildability of the architecture increase, the effort/costs involving every addition/change/configuration on the system would decrease, but also the infrastructural costs to run their new core system would decrease as it is much cheaper to run this new technology than to run Mainframe. There was also another feature that made TNCS the best choice to achieve those scenarios: its modularization allows for the isolation of changes when defining/changing insurance products and/or updating the pricing algorithm, reducing the ripple effect. Each one of the TNCS’ modules, which are presented in figure 3.2, have specific responsibilities, namely:

**Product Definition:** This module holds the code used to define all the insurance products: their structure, their coverages, their pricing structure and algorithm, the renewal rules, and the insured objects’ information. These sub-modules are identified in figure 3.3.

**Claims Management:** The creation and handling of all claims is done in this module. It also handles the claim’s structure for each insurance product;

**Policy Management:** This module has all the logic to handle the quotes and policies life cycles: creating new ones, cancellations and middle-term adjustments;

**Party Management:** Responsibilities of this module cover the creation and management of persons and companies, which can be customers, leads or interested parties (e.g. for claims handling);

**Account Management:** This module has the responsibility of managing the accounts and payments related to the policies and parties inside the system;
Authorization & Referrals: A module that encapsulates all the underwriting functionalities needed in the insurance business, such as the ability to define constraints and rules for policy creation and modification, generic or specific for each insurance product, and verify and enforce those constraints;

SOA Layer: This transversal module provides a set of services\(^2\) that allow the interaction with TNCS in multiple domains, e.g. creating quotes, retrieving party information;

The usage view in figure 3.2 shows the dependencies between modules, which allows a better understanding of the effects of adding, or changing, an insurance product. When adding/changing an insurance product, the majority of the effort will be spent in the Product Definition module - where the product will be fully defined, but leveraged in the already existent framework of TNCS, instead of defining it from scratch. As a product is created/modified, it will then require modifications on the Claims Management module, in order to define the claim's structure for the product, and it will require changes on the Authorization & Referrals and Account Manager modules in order to define specific configurations for that product.

Another factor that contributed to the adoption of TNCS as the solution to achieve these scenarios, was that the functionalities provided by the Account Management module provided an alternative to the existent Account System. By using TNCS’ Account Management module to decommission the Account

\(^2\)The characteristics of a SOA, and the definition of a service, are discussed in detail in section 3.2.3.1.
3.2 New and old systems must be integrated

Interoperability System quality

Taking into consideration the roll-out strategy set for the Transformational programme, new systems
would be introduced to the landscape of InsuranceCorp, while the old systems would still be active and
used on the day-to-day business operation. Because of that, the architecture had to be designed in a
way that would allow all the systems to integrate seamlessly in order to support the business operations,
independently of which system is being used - old or new.

3.2.1 How it relates with other qualities

This System quality attribute derives from the Roll-out Schedule Business Quality, as it is a direct
consequence of the necessary coexistence of all systems. It also relates to Cost and Benefit as the
cost of management of the interfaces between new and old systems in InsuranceCorp’s architecture
would desirably be kept to a minimum.

3.2.2 Scenario 1

In order to construct a more concrete specification of this System quality attribute, the following
scenario was defined:

Scenario 1 - Guarantee systems’ mutual communication

Source InsuranceCorp’s IT;
Stimulus Addition/removal of a system to/from the architecture;
Environment Design time;
Artifact InsuranceCorp’s Architecture;
Response All systems continue to work integrated;
Response Measure No system shall lose functionalities due to the removal of another system from the
landscape; Cost of adding/removing a system should be minimal;

3.2.3 Tactic applied

In order to achieve the intended interoperability between all the existent systems, and also the new
ones to be introduced in the landscape, and upon research and consulting with architectural experts,
InsuranceCorp opted to put in place a Service-Oriented Architecture, with the implementation of an En-
terprise Service Bus in order to centralize the communication between components, and orchestrate the
interfaces of all systems. The following subsections present this type of architecture and its characteris-
tics.
3.2.3.1 Service-Oriented Architecture

The Service Oriented Architecture (SOA) is an architectural model that stands in the paradigm of Service Orientation. This paradigm consists on the creation of self-contained logical units, known as services, which can be used repeatedly, individually or orchestrated together with other logical units, in order to achieve a certain goal, in this case, a business functionality. Each Service is designed with the following principles taken into consideration:

- **Standard Service Contract**: The contract of a service must be defined through the same principles and rules as the other services existent in the same catalogue;
- **Loose Coupling**: A service should define its contract independently of its consumers, logic or implementation;
- **Abstraction**: The contract of a service should not expose the logic behind it;
- **Reusability**: A service should be designed in a way that it could be reused through more than one orchestration, and by more than one consumer;
- **Autonomy**: A service should have full control of the logic regarding its operating range;
- **Composability**: Each service should be designed in order to solve a very specific small problem. By doing so, it allows the composition/orchestration of many services in order to solve bigger problems;

These business functionalities are shaped and aligned with a strategic view of the organization’s business and processes, therefore providing an architecture much more based on how the organization is intended to work and what requirements need to be met business-wise, instead of what systems do and how they do it.

The Service Orientation paradigm can be seen at different levels. For instance, a system can provide a layer of services, related to its information domain and capabilities, to be accessed directly by other systems, therefore creating a well defined interface for interactions coming from other systems, instead of opening up its data sources. This paradigm requires a much more planned and thought approach upfront when compared to others, since it needs to define the set of operations that will allow other systems to consume. However, the existence of a single system with these characteristics does not make the architecture where it belongs a SOA. For it to be considered a SOA, all online communications performed between its systems must be done through service consumption. And that can be achieved in 2 ways:

- **Direct Systems Communication**: It requires all systems to expose a service layer, and requires all systems to consume directly the services of the systems that provide the information needed;
- **Indirect Systems Communication**: This scenario requires the creation of a centralized intermediary, which will become the single point of contact for all inter-system communication, abstracting each system from the existence of the other systems;

As demonstrated by figure 3.4, the 2 scenarios create a very different architecture. If we assume, as an example, those 2 scenarios, with 8 systems, in which every single one of it needs to interact with the remaining 7 systems, the Indirect Messaging scenario demands 16 interfaces, one in each direction, while the Direct Messaging scenario would require 56 interfaces. Having 40 interfaces less, means less time required to develop them, to maintain them, which leads to a more cost-efficient approach.
That was the approach that was intended for the InsuranceCorp, since it had so many systems in its architecture that required to talk to each other: to create a Service-Oriented Architecture with a central intermediary, that would allow a more cost-efficient implementation and an easier and cheaper maintenance. That led to the implementation of an Enterprise Service Bus (ESB), explained in section 3.2.3.2.

3.2.3.2 Enterprise Service Bus

The Enterprise Service Bus (ESB) is a middleware component, used in Service-Oriented Architectures, that implements an intermediary element between mutually-interacting systems, which expose a set of loosely-coupled elements (services) called by the systems in need of an action/data. This allows the creation of a single point of contact for all systems, which defines a specific data model for its interactions, with a single technology. Therefore, systems only require to interact with the ESB, ignoring completely the transformations and calls the ESB has to perform behind the scenes, putting into ESB the responsibility of orchestrate the calls between all interfaces. ESB’s general capabilities/responsibilities can be summarized as:

- Control versioning of services;
- Monitor and control the flow of messages that are exchanged between all systems;
- Translate, enrich and route messages to the correct systems;
- Hide differences - technology, protocols and behaviour - between the calling and the responding systems;

This type of abstraction provided by this component is of paramount importance when applied in complex applications’ landscapes, such as the one in InsuranceCorp, with a multitude of technologies, data models and communication methods. It can than be stated that the ESB provides the unique way of interaction between systems.

As the center of all communication in the enterprise, the usage of an ESB presents a set of benefits, namely:
• **Scalability:** The ESB is completely stateless, which allows it to easily grow infrastructural-wise, helping the scalability of the architecture - This quality attribute is explored in more detail in section 3.4.

• **Modifiability:** Allows the introduction of new services and new systems without impact to the existent ones - This quality attribute is explored in more detail in section 3.3.

• **Monitoring:** It provides a great monitoring component since all messages exchanged between systems go right through the ESB;

The benefits of this communication type can only be truly harnessed if the existing systems in the architecture require several integrations with each other, which is the case in InsuranceCorps as evidenced in figure 2.1. The introduction of the ESB created one of the biggest changes in InsuranceCorp’s architecture since it relieved the existing systems from the responsibility of modelling their messages for each system it intended to communicate with. The introduction of this component aligned the InsuranceCorp’s IT with the drivers of the Transformational programme: it added flexibility, in order to adapt in a quick manner to changes, and it also reduced costs, since it is quicker, and therefore cheaper, to extend a system caring with only one target, provided by the ESB. With that addition, the architecture was now shaped as depicted in figure 3.5.

Although some non-service integrations, such as file and queue integrations, were left in the architecture, especially when connecting the minor systems, the main systems in the architecture saw their major load move to a service-oriented integration. That was achieved by adding a component in the Mainframe’s Policy System and Party System, that would expose a set of services, to be consumed by ESB, serving as a proxy to their queues. With ESB as a core part of InsuranceCorp’s architecture, it was necessary to define how ESB’s services would be implemented. For each service, 2 main modules were defined, as represented in figure 3.6 due to its separate responsibilities.

The Data Model module implements the model that the ESB provides, and due to its importance, a specific scenario was created for it, presented in section 3.2.4. The Logic module would address all the processing required to transform, enrich and direct requests and responses from consumers to providers. This module is address in more detail in section 3.3.

With a vision of how the new architecture in InsuranceCorp looks like with the introduction of the ESB, it is important to understand how the architecture was envisioned to expand with the migration process going on, and the extensions needed as InsuranceCorp’s business progresses. An insurance product is the changes’ driver on the architecture, either by designing and implementing a new one in TNCS from scratch or based on one, or more, insurance products in the Mainframe’s Policy System, and everything starts exactly there: TNCS. The design and implementation of one insurance product results in the modification of TNCS’ modules, especially the Product Definition one as depicted in figure 3.2. Those changes modify the information exposed by TNCS’ SOA Layer module, and multiple consumers of that information, e.g. the Agent Sales’ UI system, need to get access to that new information. So what is now done, due to the new architecture, is that the ESB is extended, either by adding a new service, because a new business operation is needed, or by extending the ESB’s Data Model because new business concepts were introduced. So once the changes in TNCS are propagated to the ESB, its up to the consumers to update themselves, if needed, to adapt to the ESB’s changes, since those are the only ones they have to consider.

### 3.2.4 Scenario 2

InsuranceCorp needed to find a Canonical model to be used in its ESB. One of the situations that would test the Canonical model extensibility, and how well it would cope with InsuranceCorp’s business,
Figure 3.5: Component-and-Connector view of InsuranceCorp’s architecture with the ESB

Figure 3.6: Usage view of modules running in ESB and its services
was the creation/modification of a product in TNCS, as its SOA layer would now provide some more semantic concepts. The understanding of that new semantic, and the ability to incorporate them, would determine the quality of the Canonical data model. That requirement is instantiated in the following scenario:

**Scenario 2 - Extend the ESB data model for an Insurance product**

One concrete scenario was created regarding the extensions in the ESB’s data model that would originate in the creation of new Insurance products, or by the gradual migration of the existent portfolio:

**Source** InsuranceCorp’s Business Department;

**Stimulus** Creation/modification of an insurance product in TNCS - causing new concepts to be exposed in its SOA layer;

**Environment** Design time;

**Artifact** ESB;

**Response** ESB’s data model is extended conserving its identity; Effort in extending the data model is limited;

**Response Measure** Man/hour’s effort in the extension work; Number of inconsistencies in the data model;

### 3.2.5 Tactic applied

The usage of a standard doesn’t guarantee interoperability when exposing a single system to an uncontrolled environment of consumers. However, in InsuranceCorp, the landscape of applications was known. So in order to find the best standard, InsuranceCorp’s IT architects researched the market for the industry best practices, and reached the following conclusion: for the Canonical Data Model it would be used an insurance meta-model created by IBM named Insurance Application Architecture (IAA).

#### 3.2.5.1 IAA - The Canonical Insurance framework

The Insurance Application Architecture (IAA) is a complete framework for the design, implementation and operation of an Insurance company. Providing a set of inter-related business models required to support the Insurance business ([IBM, 2008](#)), it offers a blueprint for application and data warehouse development, providing business and technical models, based on the set of insurance industry’s standards at multiple levels:

- **Common processes:** how business should be conducted;
- **Common business terms:** what should the communication be like;
- **Common application components:** the applications, and its components, required to provide support to business;
- **Common data and data warehouse models:** how business information should be stored;

In short, what this framework allows, is the creation of the operation of an insurance company from scratch, by abstracting from the core and support systems, and fully aligned with the industry standards and best practices, while reducing risk and saving costs usually spent in IT analysis and design. For
companies already running that desire to completely refactor the way they work, IAA offers a full Target Operating Model, which, depending on how far from IAA they are, can become costly to achieve in the beginning, due to the numerous transformations that it may require. However, once reached that state, a company will operate following a standard of the industry.

As described in figure 3.7, the architecture advocated by this framework is sustained in 4 groups of inter-related models:

- **Foundation Models**: The models that compose this group are intended for a standardization regarding the business terms and definitions used in the communication throughout the whole Enterprise:
  - **Business Terms**: Catalogue of Insurance Business terms used for defining a unique language through the organization, which eases the process of requirement gathering;
  - **Business Model**: A dictionary of business concepts, and the relationship between them, in order to provide a bridge between the Business and IT of an organization;
  - **Business Activity Model**: Provides a list of over 700 business activities for the organization to operationalize;

- **Data Models**: Models that provide an interpretation ground of the data spread throughout the Enterprise, focused in projects of Data Warehousing and Business Intelligence:
  - **Business Solution Templates**: Reporting structures and pre-defined Key Performance Indicators (KPIs);
  - **Enterprise Model**: Pre-defined structure of a central warehouse database;
**Data Marts:** How the data marts that compose the organization’s data warehouse should be designed;

- **Process Models:** Models that sustain the Business processes’ modelling and execution:
  - **Analysis Process Model:** Enterprise-wide Insurance business processes’ flow description;
  - **Service Models:** The models existent in this group define the components that need to be developed in a Service-Oriented Architecture that supports the Insurance company operation, and are the mainstays for the work in InsuranceCorp. It also provides models for accelerated insurance product’s design:
    - **Enterprise Component Blueprint:** Provides a description of each component, and its domain/role in the organization.
    - **Business Object Model (BOM):** Formalization of the Business Model concepts in service candidates, with their input and output messages;
    - **Interface Design Model (IDM):** Structures BOM’s analysis into detailed service definitions and collaborations between services to meet business needs. Its an extensible model in order to incorporate concepts that may not exist, e.g. if a company needs to expose a property in the object Vehicle to store the number of airbags, if it is not available in the model, it can be added, as long as it makes sense semantically;
    - **Product Model:** Guidelines, structures, rules and life cycles for Insurance Products’ design;
    - **Specification Framework:** Design tools for product engines, and their managing applications;

Each group of models cover a sector of an insurance organization, and the modularization of this framework allows for its implementation in multiple scenarios and realities. Just because it provides a complete roadmap for a new insurance company, that does not limit its usefulness to just that case. Existing companies, with several decades of operation, can cherry pick some parts of the framework, and implement them. And that was what InsuranceCorp decided to do: focus only in the Service Models in order to implement the Canonical Data Model for its ESB. In order to do so, and following IBM indications, as presented in [IBM, 2008], the main IAA models to be considered in a SOA implementation were the Business Object Model and the Interface Design Model.

**Interface Design Model**

The Interface Design Model (IDM) is the key data model for this Transformational programme. It can be viewed as the Service Catalogue of IAA, therefore it was the one used to define the service operations that were implemented in InsuranceCorp.

The IDM is structured in 3 different modules:

- **Transactional module:** The transactional module contains the business’ services offered to execute a business process. This means that the services presented in the transactional module are oriented to an operation done by a business application;

- **Structural module:** The structural module contains the services that support the maintenance operations required for each business process. This means that the services presented in the structural components are more oriented to a system functionality;

- **Utility module:** The utility module contains services that aid the execution of a business process. This means that an isolated call to one of its operations do not create business value, however its execution may be paramount to the execution of a business operation;
Each module is divided into sub-modules, domains of action, as presented by the figure 3.8. Each of these sub-modules has multiple interfaces, each one of these interfaces has multiple operations, represented in figure 3.10. This decomposition follows the structure depicted in figure 3.9 and can be instantiated in the following example:

```
Structural module
  ↓
Financial Services Agreement
  ↓
Insurance Policy Manager
  ↓
Establish Quote
```

This example is in fact one of the services that were implemented in InsuranceCorp as part of its Service Catalogue, which can be seen in detail in section 4.1.

![Figure 3.8: Non-exhaustive list of domains in Structural module of Interface Design Model](image)

![Figure 3.9: Structural decomposition of a service's operation in IDM](image)
Applying IAA, a product at a time

Purpose of IAA in InsuranceCorp was to design and implement services to support the operation, however the implementation of IAA is non-trivial, so it is necessary to guarantee a set of guidelines, strategies, and processes in order to develop a framework that allows repeatability and method in the implementation of IAA in InsuranceCorp.

IAA Principles and Guidelines in InsuranceCorp

As underlying principles, IAA stands for the usage of standard insurance concepts, and use them uniformly throughout all levels of IAA implementation. It also stands for reusability of its components, but always with the option to be extended as needed. This aligned well with InsuranceCorp’s intentions of implementing a Canonical Model that would provide them a solid base to work with, but with margin for their own extensions.

Strategy for the Transactional Services

The transactional services are envisioned to be mapped to the functional needs of services’ consumers. In this sense, they should be the first type of services to be identified. On the other hand an effort should be made to maintain them mainly with the original IAA vanilla features – this way, their original functionality and exposed interface are maintained.

Strategy for the Structural Services

In InsuranceCorp it was adopted a structural layer abstracting the transactional services from the back-end, as defined in IAA, but in some cases it was necessary to create, or extend, some IAA structural services. This was done because of the granularity of one back-end system interface and because of the need to merge subsets of information for which there wasn’t an IAA structural service defined. In the occurrence of such a scenario, the IAA concepts/types were still used, allowing the new/altered service to communicate with the rest of the framework.
**Strategy for the Utility Services**

Utility services are envisioned to be implemented at the same level of Structural Services, using the same strategy in terms of extension and use of the IAA concepts, still these services could only be called in the context of another service orchestration and never as a stand-alone execution, since its output had no isolated business value.

**Team breakdown to handle IAA**

The handle of the IAA required a team composed with members with business and technical knowledge, in order to completely manage the IAA framework to its full extent. With that in mind, a team with elements of 2 roles, with different backgrounds, was created to handle it, especially in the Design phase. Therefore, Business Analysts and Technical Analysts worked together to design the IAA model that would be used in InsuranceCorp’s ESB. Because the activities of the Business and Technical analysts were quite interdependent, an effort was made to guarantee that both were able to progressively perform each others tasks.

The main activities of each role are presented below.

**Business Analyst**

- Compilation of application requirements;
- Clarification of business concepts;
- Perform IAA functional mappings;

**Technical Analyst**

- Identification of operations to be implemented;
- Perform IAA technical mappings;
- Guarantee articulation with the implementation team;

A third role was present on this team, and it belonged to InsuranceCorp, with the regular presence of a resource which performed the following activities:

- Validation of the IAA mappings suggested by the Functional Analysts;
- Validation of changes in the IAA model;
- Generation of the Data Contract (WSDL file) containing all the objects, properties and relationships in IAA, including its extensions. This generated data contract, a DLL file, is the basis of the technical mappings work;

**Building through IAA: Step-by-Step decomposition**

The process of building the ESB based on the IAA can be depicted by the flow diagram in figure 3.11 which presents the breakdown of steps involved in what can be called as a framework in order to approach an IAA project. This breakdown also presents the roles separation between the Business Analyst and the Technical Analyst.

Next is described in detail each of the 12 steps, presented in figure 3.11 the InsuranceCorp’s IAA approach. This process is what defined the data model of the ESB, and the service catalogue that it would expose.
Figure 3.11: Step by step breakdown of building an IAA project
Step 1 - Define application that will drive the mapping process

By requiring a canonical model to be used in the architecture’s middleware layer, IAA had to be modelled by selecting one driver: the Front-end applications, or the Back-end applications. The decision would determine if the final data model would be aligned with the back-ends, namely the TNCS since it would be the main one, or with the Front-end applications’ needs. In order to aid that decision, a set of criteria and indicators were defined, as presented in table 3.1.

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Indicators</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Stability</td>
<td>number of fields that are mapped to IAA but that will not be used by the consumers</td>
</tr>
<tr>
<td>Mapping Effort / Scope Accuracy</td>
<td>number of fields that would be required to map but not reusable by other consumers</td>
</tr>
<tr>
<td>Knowledge of the system</td>
<td>number of people knowledgeable in the system’s functionalities and that are available to provide clarifications to the team</td>
</tr>
<tr>
<td>New Systems in the future</td>
<td>probability of a new system appear in the short / medium term with different field requirements</td>
</tr>
</tbody>
</table>

Table 3.1: Main Criteria and Indicators for IAA’s application driver choice

For InsuranceCorp, the decision was to model the IAA aligned with the Back-end systems - namely TNCS - due to its stability, and the low number of Back-end systems, which would add up to 2 if Party System were to be included, when compared with the higher number of Front-end applications - existent and foreseen. This means that the model would be structured in a similar way to TNCS, mimicking its behaviour and capabilities. This would not determine the objects, and attributes, to be mapped, and the reason is that TNCS exposes a considerable amount of fields, almost 500 - with the majority of them being dynamic fields which would have different functional meanings based on the insurance product it is being used/configured. For that reason, the driver for the IAA fields to be used was the consuming applications, which leads to the next step of the framework.

Step 2 - Gather application’s requirements

Once it is decided which system will the IAA be oriented to, it is necessary to understand the requirements that the consumer applications have, in terms of fields that compose their interactions and the way their processes are designed in order to obtain a top level view of the fields that have to be modelled to IAA.

Since there can be multiple applications involved, it was considered a good practice to individually gather the requirements of each application, and in the end compile them together in one single location, in order to address the redundancies or conflicts that may exist between the applications.

In InsuranceCorp, there were multiple consumers to the information in back-ends, like the Agents’ Sales Tool and the Printing Solution.

Step 3 - Analyse application’s requirements

Once a top level view of the fields involved in each application is gathered, its time to drill down the existent information in order to actually understand the business processes, and business value, involved in all the selected fields.

Through the study of the Functional and Technical designs of each application, and also through meetings with the application owners in InsuranceCorp, the IAA team got clarifications that would lead to 3 major results:

- Thorough understanding of the business landscape;
Deep knowledge of the application requirements and functionalities;

Clear notion of the type of services that would be required to support the business;

Step 4 - Identify operations to implement
Having understood the application functionalities, the IAA team studied and identified a series of operations, and documented the findings in order to obtain its validation from the application owners. For InsuranceCorp, in the early stage of the Transformational programme, the main operations identified were related to an Insurance Policy: create a quote, activate a quote into a policy, modify a quote, retrieve a quote’s information - all operations needed for the Agents’ Sales Tool.

The identification of operations to be implemented is a major milestone in the project and thus should be validated by both the application owners and be signed off by InsuranceCorp.

ESB services must check that the signature (input/output parameters) is adequate to satisfy the data requirements of the business activity that they are serving. If not, either the service chosen is incorrect, or it needs to be customized (customization of service signature is strictly governed by InsuranceCorp).

In order to support the validation process, a document should be produced with the following information:

- Type of service’s action required;
- Objects to be returned/used by that operation;
- IAA Operation to be implemented, taking as a baseline the operations defined in the IDM Model in IAA;
- Identification of the usage of the operation per application existent in the application landscape;

A sample of this artefact is presented in the table below:

<table>
<thead>
<tr>
<th>Action</th>
<th>Objects</th>
<th>IAA Operation</th>
<th>For 1?</th>
<th>For n?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retrieve</td>
<td>Policy/Quote</td>
<td>Retrieve Insurance Policy Details</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Create</td>
<td>Quote</td>
<td>Record Quote</td>
<td>N</td>
<td>Y</td>
</tr>
</tbody>
</table>

Table 3.2: Sample Overview table of the IAA operations to be implemented

Step 5 - Define Service architecture
The identification of the operations to be implemented allows the technical team to design the required extensions of the ESB Architecture. The way operations will work, with or without other existing operations, and what components will compose each operation, are then represented in two different artefacts: a Service’s Functional Description and a Service’s Component Diagram. The Service Functional Description, depicted by a real example in figure 3.12, is a sequence diagram designed to functionally identify the involved applications, which are usually the ESB and a consumer application, and their components and how they relate in order to provide the functionality of the Service operation. The Service Component Diagram, as depicted by figure 3.13, is a technical diagram identifying the technical components that make part of a service, also identifying the SOAP Actions involved, and what are the systems/services hit in order for the service to operate.
Figure 3.12: Example of a Service’s Functional Description

Figure 3.13: Example of a Service Component Diagram
Step 6 - Analyse field’s business concepts

The main goals of analyzing the business concept of each field are the following: identification of the scope of fields that require an IAA mapping and documentation of the detailed business concept that will support the IAA mappings.

In this step the team uses the application requirements information as a basis to have a meeting with the application owners and clearly define which application’s artefacts are mappable. For instance, types such as buttons, table headers or labels should not be mapped. However, these fields may provide clues to the services that should be implemented, for instance a search button may indicate a retrieve functionality requirement.

After having defined the scope for the IAA mappings, the business concept must be defined with the help of business experts. The detail of the business concept of the fields may vary according to its complexity / ambiguity.

Step 7 - Define the Product structure

In parallel to the operations identification, the insurance products structure must be defined in order to guarantee a proper scope when performing the IAA mappings. An insurance product structure is also used as a basis for the definition of common technical paths in step 8.

The classes represented in the product structure are IAA objects, and thus should respect the canonical cardinality and relationship defined in IAA – though they may be changed to meet requirements if necessary. The underlying reason for this step to occur is to facilitate the mapping and validation processes.

In case the system has a standard data structure (e.g. Policy, Policy Line, Object, Covers), which is the case in InsuranceCorp with TNCS, the product structure should attempt to mimic the existing data model structure, as referenced in figure 3.14, in order to guarantee consistency and speed up its understanding by the system owners.

Step 8 - Perform functional mappings

Having identified the business concepts and knowing the product structure, the Business Analyst must navigate through the IAA Models and identify the IAA object and property that match each of the business concepts and document it. This task required a considerable effort since each field previously gathered had to be matched single-handly to the IAA model, but in a way that the final set of fields made sense as a whole.

If the IAA model does not contain a specific business concept, it is necessary to present that gap to InsuranceCorp, so that its resources will manually extend the IAA model in order to accommodate that missing concept.

Step 9 - Document technical path in C#

The technical mappings strategy used was to start by documenting the most commonly used paths (from now on referred to as Common Paths), which are used to map paths and associations between the most commonly used classes in IAA, in such a way that they can be re-used in each field mapping; then map the Specific Paths, which are a result of the analysis of the functional mapping in a field by field basis, starting from the Common Paths. The path is obtained by navigating on the data contract, a .dll file generated by the InsuranceCorp, which can be added to any C# project (more information regarding C# can be consulted in [Microsoft, 2015a]), that includes all the IAA scoped fields and associations.

As an example, a common path can be how you reach a location in a Property insurance product’s policy which is:
Figure 3.14: Similarities between TNCS’ insurance product structure and InsuranceCorp’s IAA one
As for specific paths, it can be how several fields of location is reached, which builds in the last example as:

(\textit{insurancePolicy.ProvidedContactPoints[i] as PostalAddress).Street} \\
\text{or} \\
(\textit{insurancePolicy.ProvidedContactPoints[i] as PostalAddress}.City)

\textbf{Step 10 - Document transformational rules}

The transformation rule consists in complementary information to each mapping, and can be decisive in order to guarantee full understanding of the mapping by the implementation team. The Technical Analyst has a particular important role in this task, since he is acquainted with the implementation needs and thus is able to infer common implementation details.

The table 3.3 presents a set of examples of transformational rules.

\begin{tabular}{|l|l|}
\hline
Goals & Example \\
\hline
Help instantiate the product structure & \begin{itemize}
  \item Party = "Company";
  \item General coverage component = "Liability Common";
\end{itemize}
\hline
Identification of type names & Type of limit is defined in the \textit{MoneyProvisionRole} object’s attribute "TypeName" and should be named as the Core field, \textit{TypeName} = "Sum - maximum per year";
\hline
Instantiation of segments and categories & Field \textit{AgeSegment} should be mapped as \begin{itemize}
  \item if group of workers are younger than 25 years old, field should be mapped to \textit{AgeSegment} with value "0 - 24 years";
  \item (...)
\end{itemize}
\hline
\end{tabular}

\noindent \textbf{Table 3.3: Transformational Rules’ goals and examples}

\textbf{Step 11 - Implement components and mappings}

A service’s implementation is based on a template provided by InsuranceCorp’s IT department which assures consistency across all services to be implemented. This segmentation allows modularity in the developed software that enables faster implementation of each service. The work done in the previous steps allow for the production of the Mapping components of each service, and also the definition of what applications, and how, will the service interact with. A detailed view of a service implementation can be found in section 3.3.3.1.

\textbf{Step 12 - Manage changes to mappings}

As a result of actual implementation of the field mappings, it is expected that a set of changes to the initial mappings occurs. On the other hand, the mappings are consumed by different application implementers. This means each mapping modification must be made public to all consumers.

Considering the multi-thread characteristics of the InsuranceCorp project, namely the existence of parallel developments, the synchronization of all teams is a crucial task.
In order to guarantee full alignment between InsuranceCorp and other consumers of IAA framework/mappings, a set of procedures were implemented that contributed to the following goals:

- Guarantee a single repository for changes in IAA mappings;
- Provide visibility of implementation status of each change per consumer (of the implemented service);

3.3 Modify/Extend the Service Catalogue

Modifiability System quality

InsuranceCorp was very focused on developing things faster and cheaper, and with an ESB in place, the modifications in its service catalogue had to respect those same directives. That is why ESB required another system quality attribute: to be modifiable.

3.3.1 How it relates with other qualities

This quality attribute that was desired to the System relates to the Cost and Benefit and Time to Market Business Qualities. Not only that, but is directly linked to the Interoperability attribute quality, which originated a SOA with an ESB, as changes made in this component are crucial for the correct operation of InsuranceCorp, and also one of the hot points for costly maintenance operations - one thing that InsuranceCorp wanted to decrease with the Transformational programme.

3.3.2 Scenarios

One concrete scenario was created regarding the effort around the creation, or extension, of a service:

**Scenario - Create/Extend a service**

**Source** InsuranceCorp’s IT department;

**Stimulus** New service/extension of an existent service;

**Environment** Design time;

**Artifact** ESB;

**Response** ESB’s modification is done with small cost and effort;

**Response Measure** Man/hour’s effort of the work;

3.3.3 Tactics applied

InsuranceCorp’s ESB was going to be a critical point of changes as the Transformational programme moved on, specially for 2 reasons: as the migration would progress, more insurance products would be exposed in TCNS, which would also make other entities communicate with InsuranceCorp (e.g. the National Vehicle Registration entity when the Motor product would migrate to TNCS), and ESB had
to be able to expand its data model to cope with those extensions, and also guarantee the mappings between its internal data model and the services’ providers data models; another reason is the evolution of the ESB in terms of number of services as more business functionalities are required. Based on that, three different tactics were implemented in order to guarantee a system that would be aligned with InsuranceCorp’s expectations, namely:

**Anticipate Expected Changes:** Services to be added to ESB would eventually suffer changes during its life cycle. Anticipate those changes was paramount in order to decompose a service in a set of modules that would guarantee that a change would limit the number of modules to be impacted;

**Hide Information:** The decomposition of modules, based on the previous tactic, had also to make sense in terms of splitting responsibilities, and also in terms of what information was exposed, or not, in each of those modules;

**Component Replacement:** By allowing multiple parts of a system to be replaceable in load time, it adds flexibility to the system, facilitating the process of deployment;

The implementation of those tactics led to the creation of a service’s template, with a number of modules, with its responsibilities well defined. That template was named Service Factory and is detailed in the following section.

### 3.3.3.1 Service Factory - Service implementation methodology

In InsuranceCorp, it was defined a template for all services implementations. This template, named as Service Factory, was developed in-house by InsuranceCorp’s IT department, together with its Code and SOA CoEs. The existence of this template provides a great benefit to the organization, since it ensures the normalization of the implementations done in-house and subcontracted to multiple contractors.

#### Technology involved

InsuranceCorp had its IT equipments and technologies bound to Microsoft licensing, so it made sense, from a contractual and technology familiarity point of view, to continue with implementing solutions based on Microsoft. For that reason, and taking into consideration the role of the ESB, it was defined that it should be implemented in BizTalk ([Microsoft, 2015b]), an application integration server of Microsoft, a middle-ware technology intended for brokering messages and integrate multiple systems and technologies. Independent from the input to all BizTalk communications, it internally transforms its messages to XML format. This aligned well with InsuranceCorp’s intention of implementing services based on SOAP standards, making service requests based on XML messages.

The Service Factory based its implementations in a generic BizTalk orchestration, named Transformer, which would load custom modules that are developed in C#. Each module is developed as a library project, with its build output, a .dll (Dynamic-Link Library) file being loaded in runtime by the Transformer in a dynamic way, depending on the service that it was requested to instantiate.

#### Service Factory’s template

The architecture designed for the implementation of a service, regardless of its complexity, is based on the execution of a single generic orchestration, that would work as the entry point of the requests that originate a service call, while pre-designated modules are loaded dynamically to address the specific
Figure 3.15: Usage view of Service Factory’s modules by the Transformer

behaviour that is expected from each service. The design of Service Factory’s template defined 5 different and inter-independent modules, presented in figure 3.15, that would have specific responsibilities, and used on different stages of a service execution. Their description can be stated as:

- **Resolver**: A Resolver component, developed in C#, allows components to dynamically use other components based on current invocation attributes. The resolver allows, for a specific SOAPAction and message type, the load of the modules (Sequencer, Message Splitter, Message Aggregator and Mappers), in its .dll files, best suited to handle that request. The result of resolving a SOAP Action is an element which uniquely identifies the configuration relevant for the SOAPAction received. This enables behaviours such as content-based or header-based routing configurable at the database level without any changes to the deployed services. This is depicted in the view represented in figure 3.16 and explained in more detail below;

- **Sequencer**: The Message Sequencing component is intended to allow other components to be called in sequence enabling control flow of the execution. It is a state machine that reacts to the last response allowing decisional logic to be performed using C# components;

- **Splitter**: Receives a XML request message from the Transformer as input and produces one or multiple XML request messages according to a specified criterion;

- **Aggregator**: Aggregates multiple XML response messages into a single XML message;

- **Mapper**: Used to map/translate a message from a data model to another. Converts a specific C# type into another in a structured way, segregating the mapping of each subtype to a specific mapper class. This means that a mapper only knows how to map the primitive types of an object, relegating to other Mappers the responsibility of mapping its complex properties;

These modules are interfaces for specialized implementations, each developed for each service. This way, Transformer just needs to understand the type of module it has to load, disregarding of its specialized implementation, adding to the intended modifiability of ESB. An example of this specialization is presented in figure 3.17.

A default specialization was implemented for each of these modules, which consisted in pass-through modules, as in the output of each module would be equal to its input. This would be useful since not all services have equal complexity in terms of business logic. Still, for each service, and based on its functionality, complexity, and target data provided, multiple modules had to be specifically implemented. As an example, a real InsuranceCorp’s service named CreateQuote, which functionality is to create a quote
Figure 3.16: Example of Transformer’s use of Service Factory’s module specializations

Figure 3.17: Example of Transformer use of Service Factory’s modules independent of its specialization
in TNCS based on the information provided by the service consumer, required the implementation of just
two Mappers: one to map information from IAA, the ESB's canonical model, to the TNCS's data model;
and another one to map TNCS's response from its data model to ESB's IAA. The remaining modules of
this service are the default ones. As another example, the structural service RetrieveInsurancePolicies,
which had a very complex business logic, e.g. based on the mask of the policy number it would have to
either retrieve a policy from TNCS or the older Policy system, required the implementation of 1 Resolver,
1 Sequencer, 2 Aggregators, and 8 Mappers. A diagram presenting the modules and the structure of
this service can be found in appendix A.

As stated previously, the dynamic loading of these modules depended on the execution of a service.
Transformer's execution is divided in 3 stages, regarding the processing of the request message, before
it is sent to the specific target contact point from where the information is going to be retrieved (this
contact point can be another ESB service or a back-end system), as illustrated in figure 3.18. In each
of the 3 stages, the modules are dynamically loaded in order to address the type of request that was
received. The 3 stages are:

- **Sequencing:** This stage is responsible for loading the Sequencing module, which has all the logic
  of the service in terms of sequential steps needed to be done. The simpler services will have a
  pass-through step, while the more complex services will have multiple steps in order to orchestrate
  all the necessary calls and transformations in order to produce the desired response;

- **Debatching:** A single service request may produce multiple calls\(^3\) in parallel, to some other
  service. That is what occurs in this stage, where Aggregator and Splitter modules are loaded in
  order to transform the original message, and then aggregate/enrich the results from the calls done
  afterwards;

- **Mapping:** This stage's role is to load the Mapper modules responsible of mapping/translating the
  requests from IAA to the target's data model, and the responses from that target's data model to
  IAA;

The sequence presented in figure 3.18 shows how the different modules interact inside the connector
between ESB and a target provider, in order to support the intended qualities of each module, namely:

**Resolver:** Loading of the correct modules for a specific execution is split in 3 different stages which
allow more isolation in terms of implementation;

**Sequencer:** A sequencer can sequence an unlimited number of steps inside a service;

**Mapper:** Mapping is isolated from other service's logic, leaving it with the sole responsibility of mapping
from, and to, whatever data model;

**Aggregator:** Enriching messages is always performed in the same data model: the one internal to ESB;

**Splitter:** Splitting of messages is done right before the message is mapped to the target provider;

The flow in figure 3.18 is static, meaning that all services respect that same execution flow, guaran-
teeing InsuranceCorp's that all its services work in a standard way, improving also how easier it is to
monitor their execution and tracking messages knowing exactly in which stage of a service they are.

\(^3\)For example, that can occur when one identifier, such as a License Plate, must be used to retrieve information
from two different systems. In that case, the original request is "split" in order to produce two different messages to
be sent to each of the back-end systems.
Figure 3.18: Sequence of Transformer's flow of execution and Service Factory modules' load

Dynamic loading of modules

As stated previously, Transformer loads dynamically the Service Factory’s modules based on the type of SOAPAction of the request message sent to the service (figure 3.16). This is performed by loading BizTalk’s internal Single Sign-On (SSO) database with configurations regarding all services, based on key-value pairs, resulting on the concatenation of a service name and an element to be loaded. These services’ configurations are aggregated by interface. Figure 3.19 provides an example of one interface’s content and for one specific service.

Figure 3.19: Example of content of BizTalk’s SSO database for Transformer’s execution

The process of loading the intended modules for a specific SOAPAction has the following steps:

1. The Resolver module concatenates the SOAPAction with the string ”|Element” in order to retrieve
the Key name for that service. In this case the retrieved Element is RetrieveInsurancePolicies;

2. The result of concatenating the Element retrieved, with the module intended, is then used to load 2 values:

   - by concatenating it with "\Assembly", the name and version of the assembly to be loaded is retrieved. This assembly will match with a .dll file installed in the Global Assembly Cache (GAC);
   - by concatenating it with "\Type", it retrieves the class implementing the intended methods to be executed in the retrieved assembly;

3. Redo the previous steps for all configurations intended to be retrieved;

All the service’s configurations are stored in this SSO database, including the endpoints of the target providers for each service. This allows to easily reconfigure, in load time, the back-end system/service that provides information - either because the provider of the information is now another system, or is now deployed in another environment/location.

### 3.4 Hold future load of requests

**Scalability System quality**

As the migration process of all the portfolio and processes continues, it is expected an increase of requests to be handled by the new pieces of the architecture, specially the ESB. With that in mind, the ESB would need to be built in a way that would be easily scalable as the number of requests that it would deal with increases - either by the increase number of InsuranceCorp’s business people using the new systems, due to the increased number of insurance products, and/or due to the increased number of services that it would be provided.

#### 3.4.1 How it relates with other qualities

This System quality attribute derives from the **Cost and Benefit** and **Roll-out Schedule** Business Qualities, and also with the Interoperability system quality that led to the SOA implementation.

#### 3.4.2 Scenario

In order to construct a more concrete specification of this System quality attribute, the following scenario was defined:

**Scenario**

- **Source** Migration process;
- **Stimulus** Increase in the amount of requests to be handled;
- **Environment** Runtime;
- **Artifact** ESB;
- **Response** The ESB handles the new load seamlessly;
Response Measure  The operational times of the systems do not float regarding the increase of requests they are asked to deal with;

3.4.3 Tactic applied

The previous scenario was accomplished with the help of the capabilities that the chosen technology provides. Microsoft’s BizTalk already has a set of capacities that, together with infrastructural configurations, would allow the architecture to withstand the amount of load that it is being asked to handle. Some of the decisions made are:

- Create 2 clusters dedicated to BizTalk (depicted in figure 3.20);

- Deploy applications in each cluster taken into consideration the duration of the transactions it would handle (low latency operations vs. long-running transactions) - the artifact deployed in the low latency server has its install view depicted in figure 3.21;

- Increase the infrastructural capacity to each cluster to handle the amount of load in runtime;

![Diagram](image)

Figure 3.20: Component and connector’s deployment view of InsuranceCorp BizTalk’s cluster
Figure 3.21: Component and connector's install view of figure 3.20's artifact
Chapter 4

Current State

As the Transformational programme advances, InsuranceCorp continues its evolution in order to modernize itself and to find some solid technology grounds to leverage its business. InsuranceCorp already is different from when the programme started, but still not yet as it envisioned itself. Some of the modifications made achieved the intended benefits, but there were some that didn’t quite hit the target as it was initially thought, and others that raised issues that were not a priority before. This chapter describes goals achieved by InsuranceCorp, issues that the implementations done through the Transformational programme raised, and also a look of the next steps being taken into consideration to handle those same issues.

4.1 Service Catalogue

The end result of a SOA implementation, and perhaps the most important artefact to be produced, is the Service Catalogue, which consists in an inventory of the available IT services offered by an organization. Table 4.1 presents the services that were developed in the Transformational programme, applying the decisions presented in the previous chapter.

4.2 Main Improvements Achieved

InsuranceCorp already harvests benefits from its Transformational programme. Half of its Commercial portfolio is already migrated and managed exclusively in TNCS. This alone reduced the processing in the Mainframe, and since its costs are associated to the MIPS (Millions of Instructions Per Second) it uses, the operational costs are already lower. Even with some drawbacks, explained in more detail in the next section, the maintenance costs and the time-to-market resultant of the ongoing programme is perceived by InsuranceCorp as very positive, which can be interpreted by maintaining the assignment of successive releases of its programme to Deloitte as its implementing partner.

4.3 Post-Mortem Analysis

With a considerable amount of developments done throughout the Transformational programme already in production, and although the final state of InsuranceCorp’s evolution is not yet attained, it is

---

1ITIL v3 defines the Service Catalogue as a “[…] structured document with information about all live services, including those available for deployment. The Service Catalogue is the only part of the Service Portfolio published to Customers, and is used to support the sale and delivery of IT Services”
<table>
<thead>
<tr>
<th>Service Type</th>
<th>Interface Name</th>
<th>Service Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transactional</td>
<td>Policy Inquiry</td>
<td>Retrieve Insurance Policy Details</td>
</tr>
<tr>
<td>Transactional</td>
<td>Policy Inquiry</td>
<td>Retrieve Places For Agreement</td>
</tr>
<tr>
<td>Transactional</td>
<td>Policy Inquiry</td>
<td>Retrieve Insurance Policies for Role Player</td>
</tr>
<tr>
<td>Transactional</td>
<td>Policy Inquiry</td>
<td>Retrieve Insurance Policies for Insured Object</td>
</tr>
<tr>
<td>Transactional</td>
<td>Prospect Quote</td>
<td>Record Quote</td>
</tr>
<tr>
<td>Transactional</td>
<td>Policy Status Change</td>
<td>Activate Insurance Policy</td>
</tr>
<tr>
<td>Transactional</td>
<td>Policy Status Change</td>
<td>Cancel Insurance Policy</td>
</tr>
<tr>
<td>Transactional</td>
<td>Customer Information</td>
<td>Retrieve Contact Points</td>
</tr>
<tr>
<td>Transactional</td>
<td>Access Control</td>
<td>Retrieve System User Details</td>
</tr>
<tr>
<td>Transactional</td>
<td>Claim Manager</td>
<td>Retrieve Claim Details</td>
</tr>
<tr>
<td>Transactional</td>
<td>Party Information Manager</td>
<td>Search Party Details</td>
</tr>
<tr>
<td>Transactional</td>
<td>External Vehicle Manager</td>
<td>Retrieve Vehicle Information</td>
</tr>
<tr>
<td>Transactional</td>
<td>External Vehicle Manager</td>
<td>Retrieve Vehicle from Source</td>
</tr>
<tr>
<td>Transactional</td>
<td>General Communication Manager</td>
<td>Create Notes</td>
</tr>
<tr>
<td>Structural</td>
<td>Insurance Policy Manager</td>
<td>Retrieve Insurance Policies</td>
</tr>
<tr>
<td>Structural</td>
<td>Insurance Policy Manager</td>
<td>Modify Insurance Policy</td>
</tr>
<tr>
<td>Structural</td>
<td>Insurance Policy Manager</td>
<td>Establish Quote</td>
</tr>
<tr>
<td>Structural</td>
<td>Insurance Policy Manager</td>
<td>Modify Quote</td>
</tr>
<tr>
<td>Structural</td>
<td>Insurance Policy Manager</td>
<td>Get Insurance Policy for Party</td>
</tr>
<tr>
<td>Structural</td>
<td>Party Manager</td>
<td>Establish Contact Point</td>
</tr>
<tr>
<td>Structural</td>
<td>Party Manager</td>
<td>Modify Contact Point</td>
</tr>
<tr>
<td>Structural</td>
<td>Party Manager</td>
<td>Retrieve Contact Points</td>
</tr>
<tr>
<td>Structural</td>
<td>Party Manager</td>
<td>Retrieve System Users</td>
</tr>
<tr>
<td>Structural</td>
<td>Party Manager</td>
<td>Retrieve Party Details</td>
</tr>
<tr>
<td>Structural</td>
<td>Communication Manager</td>
<td>Establish Notes</td>
</tr>
<tr>
<td>Structural</td>
<td>Physical Object Manager</td>
<td>Retrieve Pledge Status</td>
</tr>
<tr>
<td>Utility</td>
<td>Data Utility Manager</td>
<td>Retrieve Alternative Identifier</td>
</tr>
</tbody>
</table>

Table 4.1: InsuranceCorp's Service Catalogue
currently possible to assess what came up short to its goals, and also what issues were raised due to
the decisions and implementations performed in InsuranceCorp and reported in this document. Two
cases that fall under that situation are the IAA data model, used by the ESB, and also the performance
registered by some of its services. Each of these cases are address in the following sections.

4.3.1 IAA’s understandability

Achieving modifiability in the data model of ESB was one quality seen as paramount, and to which
InsuranceCorp dedicated much thought, that’s why IBM’s IAA data model was chosen, and why the
process of implementing it to InsuranceCorp’s reality was so meticulous, and it required the use of
existing generic concepts in order to keep the model as open as possible in case some other change
was needed. However, this line of thought ended up creating a very complex, deep, nested-relations
filled data model, that had portions so unnecessarily generic and abstract, that eventually grew into a
model very difficult to understand. This raised two main issues, namely:

- It created a steep learning curve regarding the understandability of the data model. This is a
  sensible topic because it ended up affecting the exact same system quality attribute it had initially
  proposed to solve, since the modifiability of the ESB was impacted;

- The “user-unfriendliness” of the ESB’s data model made it not considerable as a candidate to
  services exposed to the outside world to external consumers;

The first issue is being addressed in a process-wise manner, with implementations and mappings
being currently made with a lesser fixation in doing things generically, in a manner that would easily cope
with the wildest possible change in the future. This, however, is something that is noticeably in the newer
extensions to the ESB’s data model, but the first developments made in it still reflect that older line of
thought. The second issue was something that did not was not a major concern in InsuranceCorp’s ar-
chitects minds in the beginning of the Transformational programme, but there was the need to eventually
expose some of the ESB’s services in the Internet, in order to allow external partners to communicate
with InsuranceCorp. The already mentioned complexity of its data model, that is manageable for internal
developments, raises some difficulties when it is time to instruct external partners on how to consume
the services if the ESB’s data model was to be used. This puts InsuranceCorp in a position where
it needs to find a way to provide an easy data model to its external partners, and this need can be
instantiated in the following scenario:

**Scenario of Interoperability**

**Source** External entity;

**Stimulus** External entities consume InsuranceCorp’s services;

**Environment** Design time;

**Artifact** External services’ data model;

**Response** The services’ data model is intuitively understandable;

**Response Measure** Understandability of the model; Time spent documenting/explaining service con-
sumption to external Partners;
This is currently an open issue in InsuranceCorp, which is still being addressed. The more backed-up solution is to create a lighter variation of the ESB’s data model, that would share the same naming conventions, and be semantically close, but with less nested-relations, and shallower. This “Light IAA” version would be exposed externally, than a set of modules would be implemented in order to map it to the normal IAA data model. Although being the more probable solution to go forward, some risks were already raised:

- Performance would be impacted due to the overhead created with an extra phase of mappings (Light IAA ↔ IAA);
- The existence of another data model to be governed inside InsuranceCorp would imply extra effort/costs;
- The new “Light IAA” would be tightly-coupled to IAA, so it may cause some difficulties to extend one, or both, in the future;

### 4.3.2 ESB services’ performance

With a huge focus on the modifiability of the services implemented in InsuranceCorp, supported on a mainstay of this Transformational programme which was to reduce maintenance costs, one crucial aspect was overlooked: the performance. It was assumed that the performance registered by all services would all be good, and equal throughout the whole service catalogue. However, that was not true, and it caused some issues in some business processes that were based on the execution of some services that could not perform to the expectations.

That situation was caused by two identified issues:

- The fact that it was not envisioned beforehand that the payload asked to be dealt by some services would be of a great size, in some cases, not even candidates to be dealt in an online fashion;
- The modifiability, and the dynamical loading mechanism, designed for the implementation of InsuranceCorp ESB’s services did brought benefits in terms of easiness, however it also created a significant overhead in the execution of each service. And if in more simple services, the overhead impact is marginal, in more complex services that require orchestration with multiple back-end systems, transporting big payloads, that overhead becomes significant;

As a concrete example, the performance of the service Retrieve Insurance Policies for Role Player, present in table 4.1 was one of those that impacted the business process that was actually the driver for its existence. The business requirement for it was that an agent, using the Agent Sales Tool UI, would like to consult the whole portfolio of a single customer. That would be achieved by consuming an ESB service that would retrieve the information from the back-end systems. The premise was simple, so the expectation from business was that it would be a simple (and fast) service, and IT’s expectation was that it would be a perfect candidate for their new SOA. However, it was forgotten that commercial customers often have portfolios with multiple policies that cover a big multitude of insured objects. In some executions of this service, it took so much time for the back-end systems to produce a response, that the service timed-out in the ESB (timeout was initially configured to 5 minutes), so no response was sent back to the service consumer. This situation raised a new quality attribute for the ESB, and its services, that was not considered previously: its performance. This new intention is better instantiated in the following scenario:
Scenario - Performance in heavy payloads’ case

Source  InsuranceCorp’s IT;
Stimulus  A service consumer makes a request that might originate an heavy payload;
Environment  Design time;
Artifact  ESB service;
Response  The ESB service responds the data expected by the consumer within an acceptable time frame;
Response Measure  Round Trip Time of the service execution; Absence of timeouts to the consumer;

This scenario is still currently being addressed in InsuranceCorp, however there are already some ideas being crafted in order to address this situation, namely:

- Categorize the service catalogue, splitting the existing services in separate categories based on its complexity, and average response time and size of payload. Then deploy each one of those groups in separate BizTalk clusters, with different timeouts’ configurations in them. This would avoid trying to force a one-size-fits-all type of configuration since not all services are equal;

- Identify the services in the service catalogue that handle, in an average way, the biggest payloads, and change its implementation from synchronous services to asynchronous services;

- Identify business processes, based on services’ consumption, that would be better addressed by a batch solution rather than by an online solution;

It is important to notice that these presented ideas are not exclusively dedicated to improve the current design/implementation, as the last idea exemplifies. It is not an act of failure to admit that a solution does not fit the day-to-day business requirements, as it was initially designed to. The humbleness of taking a step back, evaluate the big picture, and try to search for answers at multiple levels, is a quality demonstrated by an organization that really wants to get better, and perform better.

Another insight that was obtained by measuring this same service, was how much did the overhead impacted the response of a service. Table 4.2 presents the execution timings, while retrieving an empty portfolio, of Retrieve Insurance Policies for Role Player service, and an older version of that same service, in a more simple implementation, not respecting the framework described in this document. Table 4.3 presents the execution timing, while retrieving a portfolio for a non existent customer, of the same 2 services as previously.

<table>
<thead>
<tr>
<th>Table 4.2: Case 1 - Execution timings while retrieving an empty portfolio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service Version</td>
</tr>
<tr>
<td>Simple Version</td>
</tr>
<tr>
<td>With Service Factory</td>
</tr>
</tbody>
</table>

For both service implementations, the exact same back-end systems were targeted. Focusing on the average times on each of the cases, it is noticeable how the more simple version of the service reduces its execution time in 25% (from 1902,33 milliseconds to 1430,54 milliseconds). Being measured in similar a situation, it was expected the same type of drop in the service’s version with the Service
Table 4.3: Case 2 - Execution timings while retrieving the portfolio of a non-existent customer

<table>
<thead>
<tr>
<th>Service Version</th>
<th>Min (ms)</th>
<th>Max (ms)</th>
<th>Average (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple Version</td>
<td>991</td>
<td>2097</td>
<td>1430.54</td>
</tr>
<tr>
<td>With Service Factory</td>
<td>2131</td>
<td>3299</td>
<td>2526.26</td>
</tr>
</tbody>
</table>

Factory modules, however the drop was marginal, around 2.5%. The conclusion in InsuranceCorp was that the service was already in its lower limit in terms of performance, since its implementation did not allow it to be quicker.

This issue was more relevant in cases where the quickness of a service was paramount, and so, that conclusion led to a new scenario:

**Scenario - Performance of fast services**

**Source** InsuranceCorp’s Consumer system;

**Stimulus** A service consumer makes a request with high responsiveness requirement;

**Environment** Design time;

**Artifact** ESB;

**Response** Consumer receives its response in a very fast manner;

**Response Measure** Round trip time of the service;

This situation has not been addressed yet, and there is no consensus on the way forward, but the two more solid possibilities are:

- Do not change anything, and try to improve services’ quickness by improving the infrastructure;

- Identify these special cases, and provide them with an "architectural dispensation", allowing the consumer system to go directly to the back-end systems’ services in order to avoid the overhead caused by the ESB;
Chapter 5

Conclusions

The Transformational programme that InsuranceCorp is still going through provides a very good image of what are the vicissitudes of designing software, and what is the processes, and issues, that an architect faces when it needs to design such a transversal and heterogeneous system. The main lessons learned throughout this case study are presented in the section below.

5.1 Lessons Learned and Personal Considerations

Based on the experience during this Transformational programme, the following are the major lessons learned, and some personal considerations, with the whole process:

- Using insurance products as a main driver for planning a Transformational programme in an Insurance company is a good idea, which allows a gradual growth of the architecture, and a good isolation of old and new systems’ responsibilities;

- The ESB's data model, with the purpose of being used as a canonical model, must take into consideration its understandability, by minding the amount of nested relations, and how deep it is. The lack of consideration for these issues may result in a data model that produces bloated messages that it will impact performance, due to the size of payload, and it will increase maintenance costs, due to the fact that it will have a steep learning curve for someone in charge of maintaining it;

- The ESB's data model has to be generic in order to support future extensions in an easier way. However, abstracting so much the model, taking it far way from InsuranceCorp's reality, severely impacted the quality of the data model. It is not an easy job, nor there is a magical formula to it, but it is very important to balance how generic a data model is;

- The Service Factory is a good framework for creating services in a similar context. Provided that the overhead it creates in faster services is acceptable, the benefits of having a modularized solution with dynamic loading in runtime makes it a very good candidate for future projects with similar conditions;

- Whenever a software has to be designed, the requirements and the quality attributes gathered upfront, are rarely the only ones expected the system is in production. This comes because the high priority system qualities are the ones pushed upfront, which is expected, however the architect must mind that the process of software designing is cyclic, and that once a system quality is addressed, other system qualities may, and most likely will, be requested;
There are some system quality attributes that may impact other system quality attributes, e.g. modifiability of a system vs. its performance, therefore an architect must mind the trade-off existent in designing a software, and more often then not, it has to arbitrate the prioritization of a system's quality attributes. This requires negotiation with the system's owner in order to reach a compromise;

Past experiences and knowledge of best practices are paramount to a good software design, because not only it will help addressing the expected system's quality attributes, but it will also help predict future requirements;

Designing a software's architecture is a continuous, cyclic and ever evolving process. Due to the fast evolution of technology, and the businesses it supports, a software's architecture is never completely finished, as it will need to evolve frequently to meet with its new qualities/requirements. That is why designing software architecture is a path only made by walking.


Figure A.1: Modules of Retrieve Insurance Policies’ service - image rotated 90° anti-clockwise