]Designing complex softwares.

PLAN

[Figure Table 2](#_Toc19656248)

[Abstract: 2](#_Toc19656249)

[Introduction: 2](#_Toc19656250)

[I. Research problem 3](#_Toc19656251)

[A) Problem description 3](#_Toc19656252)

[B) What is Design, develop, complex software? 4](#_Toc19656253)

[C) Fined grained view and sub categories of the problem 4](#_Toc19656254)

[II. State of the art 5](#_Toc19656255)

[A) Model Driven Architecture (MDA) 5](#_Toc19656256)

[i) History 5](#_Toc19656257)

[ii) What is MDA? 7](#_Toc19656258)

[iii) Drawbacks of MDA 9](#_Toc19656259)

[B) Behavior Driven Design (BDD) 10](#_Toc19656260)

[i) History 10](#_Toc19656261)

[ii) What is BDD? 10](#_Toc19656262)

[iii) Drawbacks of BDD 14](#_Toc19656263)

[C) Domain Driven Design (DDD) 15](#_Toc19656264)

[i) History 15](#_Toc19656265)

[ii) What is DDD? 15](#_Toc19656266)

[iii) Drawbacks of DDD 21](#_Toc19656267)

[D) Conclusion 23](#_Toc19656268)

[III. Research Methodology 23](#_Toc19656269)

[A) Qualitative analysis 23](#_Toc19656270)

[B) Quantitative analysis 23](#_Toc19656271)

[C) Results and comments 23](#_Toc19656272)

[IV. Assessment and validation of results: case of AXA Bank DSI 23](#_Toc19656273)

[V. Conclusion and further work 23](#_Toc19656274)

[Table of acronyms 23](#_Toc19656275)

[Bibliography 24](#_Toc19656276)

[Appendices 25](#_Toc19656277)

[A) Semi-structures Interview Guide 25](#_Toc19656278)

[B) Questionnaire for the quantitative analysis 25](#_Toc19656279)

# Figure Table

[Figure1: MDA Overview 8](#_Toc19649545)

[Figure 2: MDA Methodology 9](#_Toc19649546)

[Figure 3: How Cucumber Works 12](#_Toc19649547)

[Figure 4: A feature “Is it Friday yet” with steps defined in Gherkins 12](#_Toc19649548)

[Figure 5: “Is it Friday yet” corresponding steps definition in a java test file 13](#_Toc19649549)

[Figure 6: Two bounded Contexts independent from each other 16](#_Toc19649550)

[Figure 7 : One business bounded context, one micro-service, one domain model, one database 19](#_Toc19649551)

[Figure 8: Overview of Hexagonal architecture 20](#_Toc19649552)

[Figure 9: Extended Hexagonal Architecture: The Onion Architecture 21](#_Toc19649553)

[Figure 9: Micro-services searches trend since 2004 22](#_Toc19649554)

# Tables

[Table1: The research problem distilled 6](#_Toc19663836)

[Table2: Existing approaches with respect to the research problem 24](#_Toc19663837)

# Abstract:

# Introduction:

According to the IT context, a complex software is considered to be related to critical business areas (health, insurance, banking), where business knowledge is held by a small group of industry experts. In addition, the software is designed with a distributed architecture, allowing the deployment and execution of the latter on several platforms and on different environments. Such software are also able to handle traffic of millions of users and run 24/7.

We use complex software platforms on a daily basis (Airbnb, Leboncoin, platforms and mobile applications for banking services) and we cannot ignore how vital they are to us. Complex software are undeniably important, and so are the challenges to developing it. Addressing the issues implied by these challenges is therefore inevitable. Over the past 30 years, we have seen a series of project management methodologies and software design models emerge to solve these issues that are of several types: from project management to software design patterns and technical coding best practices. However, to get to know these practices, you have to read tons of documents, sometimes not clear enough and not very explicit, which can lead to misunderstandings, abandonment of the learning process, misuse and finally facing the same issues that the methodologies where supposed to solve.  
This paper did focus on software designs patterns and technical development best practices with the aim to suggest guidelines with a declarative methodology.  
Methodologies have been elicited and categorized by the kind of issues they are addressing:  
Design issues, Software implementation related issues, and Methodology concreteness related issues, then thanks to qualitative and quantitative research through semi-structured interviews and questionnaires, feedbacks from stakeholders [[1]](#footnote-1) of complex software projects have been extracted with the most important methodologies and practices in order to provide the guidelines, that have been assessed through a bank loan project at AXA Bank.

# Research problem

## Problem description

Companies seek building customer’s loyalty, supply ever better products and services on an ongoing basis. Hence they face the challenge of creating software for critical domains such as health, finance, e-commerce, loans and mobile banking.

Yet, they are facing poor collaboration between domain experts and development teams, since such collaborations happen only on early project steps. Moreover there is no specific software architecture; the *big ball of mud* pattern is then applied.

As i noticed at AXA bank Information System Department, where I’ve been participating to the designing of Java based applicative systems providing customer bank services, the lack of frequent synchronization between domain expert and technical teams leads to the following problems:

Sluggish pace of feature enhancement, buggy releases and frequent production incidents and Developers firefighting continuously.

As the software community tried to come across these problems, we witnessed the venue of a bunch of designing principles that must be applied to a proper context to prove their usefulness, unfortunately, this decision is not easy for developers, since deciding of a designing principle with respect to the context requires reading and mastering the concepts across years of experiences and tons of documentation.

As a result, developers find themselves applying design patterns either to an improper context:

* Applying advanced architecture pattern to simple problem domains e.g. presentational website, small size school management system, event registration system
* Applying advanced architecture patterns without a skilled, motivated and passionate team
* Trying to implement a domain based design without having access to domain experts
* Applying iterative driven design methodology without using an iterative development methodology

In a nutshell, how can teams Design and Develop complex Software, following concrete steps, in order to tackle the previous problems?

## What is Design, develop, complex software?

Designing is deciding what to build And How to build.

Developing is implementing what have been decided.

A complex Software is characterized with the following:   
- Related to critical business areas (health, insurance, banking),  
- Business knowledge held by a small group of domain experts.   
- Distributed architecture, allowing the deployment and execution of the software on several platforms and on different environments,  
-Handle traffic of millions of users and run 24/7.

According to these definitions, the problem can be reformulated as following:

How to decide what to build and how to build to provide critical services to thousands of users?

How to build and correct quickly what have been decided?

Following clearly defined and concrete steps?

Still, these statements represent the problem in a coarse grained view that we need to zoom in to get more insights about resolution axis.

## Fined grained view and sub categories of the problem

Getting to know a more fined grained view of the problems will definitely help us answering the good questions.

* Deciding what to build and how to do it relies heavily on the collaboration quality between domain experts and the development team in order to fill the gap between the both:

How to educate the teams and let them notice the importance and the priority of getting aligned with the business?

How to speak to the business people and stakeholders in order to captivate their interest and get the most correct domain knowledge as possible?   
As business people are not as available as the development team wants, she must master how to captivate their interest about the value of frequent exchanges with her.

* When it comes to implement a maintainable solution, according to what have been decided by the stakeholders, the software development patterns come into play:

Which development pattern helps the team to focus on domain issues as described by domain experts?

How to separate technical implementation concerns from domain logic issues?

How to apply an architectural style facilitating scalability and features enhancements?

* Defining concrete steps that are easy to apply doesn’t mean defining a silver bullet resolving all kinds of complex software related issue but is mainly about providing guidelines that are quite declarative rather than imperative.

With fine grained statements comes out that we are facing three kinds of issues:

* Designing issues
* Technical implementation and maintenance issues
* Approaches concreteness issues

Table1: The research problem distilled

|  |  |  |
| --- | --- | --- |
| **Research problem** | **Coarse Grained Issues** | **Fine Grained Issues** |
|
| **How to design and develop complex softwares ?** | **How to decide what to build and how to build to provide critical services to thousands of users?**  **Designing issue** | **How to educate the teams and let them notice the importance and the priority of getting aligned with the business?** |
| **How to speak to the business people and to captivate their interest?** |
| **How to get the most correct domain knowledge as possible?** |
| **How to build and correct quickly what have been decided?**  **Implementation and maintenance issue** | **Which development pattern helps the team to focus on domain issues as described by domain experts?** |
| **How to separate technical implementation concerns from domain logic issues?** |
| **How to apply an architectural style facilitating scalability and features enhancements?** |
| **Approach concreteness issue** | **Declarative guidelines rather than imperative?** |

Getting to suggest solutions to the aforementioned issues requires first of all analyzing what have been done so far in companies and projects.

# State of the art

## Model Driven Architecture (MDA)

### History

A model is a set of simplified representations of a reality that is too complex to understand as is. A model is supposed to be understood by everybody, including people not holding enough knowledge about the related domain.  
A Meta model defines the informations such as the language, needed to build, read and understand the model.  
Before MDA standards, models were used in a contemplative way, meaning models were designed, and address to the developers as an inspirational tool. In software engineering, the Unified Modeling language (UML) is a meta-model defining the language used for describing Object oriented software artifacts.

Ex: Designing class Diagrams, just showing the developers what classes are to develop.

Moreover as there were many Meta models languages being developed independently such as UML and the Object Constraint Language (OCL), the fear of inconsistency and compatibility between different Meta models was growing so the need of synchronization.

### What is MDA?

#### Definition

#### Model Driven architecture is an approach to using models in software development to produce applications independent of the infrastructure they use. (Bhatti & Malik, 2015)

#### That is the generation of code from models. MDA has some core standards/meta-models such as XMI (XML metadata interchange, Unified Modeling language (UML), Meta object facility (MOF), Common Warehouse meta-model (CWM) which help developments being more aligned on models.

#### Basic principles/goals of MDA:

#### Everything is a model,

#### A model can be transformed to another model based on Meta models,

#### Meta models help separation of concerns,

#### There must be a separation of concerns between business concepts and platform (technical) aspects each corresponding to a meta-model,

#### Separation of implementation details from business functions,

#### Make applications independent of the infrastructure they use,

#### Description of the developed system by specific domain/business language concepts,

#### Convert automatically business based models to platform/technical specific models, Ex: From UML class model to JAVA Code using JMI standard or TO XML using XMI.

#### Not focusing on programming languages only but modeling languages too.

#### Models are not perceived as simple documents anymore as they are reusable and portable.

#### Easily integrate new implementation infrastructures into existing designs,

#### Generate significant portions of application-specific code, configuration files,

#### Data integration bridges and other implementation infrastructure artifacts from models,

#### More easily synchronize the evolution of models and their implementations as the software evolves, and rigorously simulate and test models. (France & Rumpe, 2007)

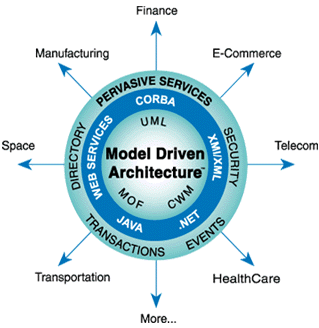


Figure1: MDA Overview[[2]](#footnote-2)

#### Methodology

The MDA methodology can be defined as follows (Bhatti & Malik, 2015):

* Specify a system independently of the platform that supports it:

This is about creating the models that highlights how does the system deliver value: the business functions to be implemented.

These models are named after Platform Independent Models (PIM) and they will survive to platforms related changes as they are entirely independent.

Ex: Defining UML models: Use case diagrams, class diagrams …

Deciding to build the system with JAVA, C++ or PHP programming language will not affect the UML Diagram.

* Specify platforms and choosing a particular platform for the system:

Transform the PIM to a platform specific model, which is linked to a specific programming language, operating system or database that computers can run.

This is done by using Query/View/Transformation (QVT).

The QVT is the standard language specification for model transformation that has been defined by the OMG. It is implemented by many software tools so that the transformation could be done easily.

Ex: Transforming UML class diagram to an Oracle relational Model or to a Plain Old Java Object Model.

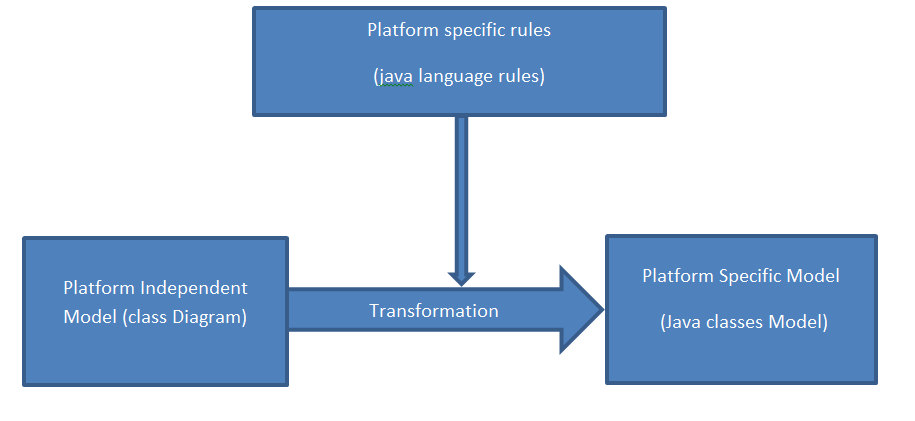


Figure 2: MDA Methodology

How does MDA bring answers to our fined grained problems?

MDA is Models driven: it uses models to emphasize understanding, implementation and maintenance hence bridging the gap between domain experts and artifacts experts. Moreover it helps focusing on the requirements for the system, the details of structure and processing being hidden.

### Drawbacks of MDA

Although MDA has been widely accepted and used in the software development world, it presents some pitfalls:

* Customizing the generated PSM lead to an inconsistency between the PIM and the PSM: The logic implemented in code is not always represented in models, therefore how to create/maintain the code base while keeping alignment with the models?
* MDA relies on a variety of technical standards nevertheless, some of which are incomplete or not yet specified or implemented (Thomas, 2003)
* MDA required specialized skillsets: Practitioners of MDA based software engineering are required to have a high level of expertise in their field, such profiles are scarce relative to the availability of traditional developers (Ambler)

## Behavior Driven Design (BDD)

### History

During the 1990s, the term **Specification by Example** started floating around.  
Specification by example is well emphasized by the following quote:   
**“We use realistic examples as a single source of truth for requirements and automated tests on software projects”** – Ward Cunningham, A Pattern Language of Competitive Development (1996)

Specification by Example was a game changer in the software development world as it allowed for a more precise method of describing and defining requirements. Testers and developers were able to extract information about requirements through concrete examples rather than abstract specifications.

Later, Test-Driven Development (TDD) coined by Kent Beck was introduced. Nevertheless, this approach was completely driven by technology and was more of a programmer’s discipline rather than that of a tester’s – however, it was also missing a high level and business readable aspect. In 2006, Dan North introduced Behavior-Driven Development (BDD) to fill this void by placing specification on a business level – not through code, but business behavior. (Wanivenhaus, 2017)

Nowadays: BDD is widely used as it allows teams to create living requirements documentation that is easy to maintain and can be consumed by all team members, including testers, developers, and product owners.

### What is BDD?

#### Definition

BDD is a specification technique that certifies that all functional requirements are implemented properly, through the connection of the textual description to automated tests. (Atem de Carvalho, Luiz de Carvalho e Silva, & Manhaes, 2010)

BDD relies on Test-Driven Development, which is a technique consisting of defining test cases for functionalities before they get implemented. This way, each implementation increment will get tested again the written cases and must meet the tests requirements to be validated.

#### Methodology

BDD relies on a set of pillars that must be used in order to follow successfully a BDD methodology:

* **Identifying the expected behaviors of the system that are more concrete and easy to identify, from the business outcomes that the system intends to produce.**
* Splitting behaviors into a set of features that indicate what should be done to achieve the business outcome: Discovering feature sets relies heavily on discussions between stakeholders and developers on business outcomes during all the product life cycle. A feature is defined by the mean of a user story and scenarios.
* **Describe Plain text description with User Story and Scenario Templates:**This is about representing the description of user stories and scenarios using a specific format: the Gherkins language

This is defined as follows:

For a user story:  
[Story Title] (One line describing the story)

As a [Role]

I want a [Feature]

So that I can get [Benefit]  
  
For a scenario:  
Scenario 1: [Scenario Title]

Given [Context]

And [Some more contexts]....

When [Event]

Then [Outcome]

And [Some more outcomes]....

Scenario2: [Scenario Title].... (North, 2006)

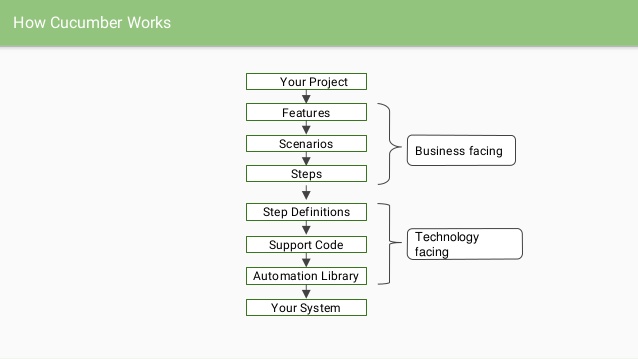
* **Automated acceptance testing with Mapping Rules:** This is where the mapping of scenarios into tests that are automatically executed happens.  
  A **toolkit** is then used to parse these textual in natural language descriptions and map them to a programming language description, which in turn is used to define automated tests that a program must meet to be validated.  
  Example: **Cucumber,** for JAVA automated acceptance tests   
  

Figure 3: How Cucumber Works **[[3]](#footnote-3)**

* **Use the Ubiquitous Language (UL) in order to define readable behavior oriented specification code**:  
  The UL is the language which structure comes from the domain model. It helps stakeholders to speak the same language than developers without ambiguity.

BDD suggests that code should be part of the specifications as well as specifications should be part of the code. That is, the names of methods and classes in the code should indicate what the method should do, should be written in sentences readable by any stakeholder. This is done by sharing the ubiquitous language between code and specifications.

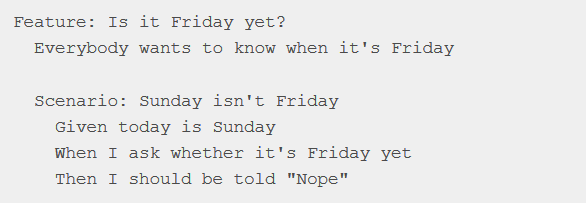
Example: Step specification file and step definition code sharing the UL  
  


Figure 4: A feature “Is it Friday yet” with steps defined in Gherkins

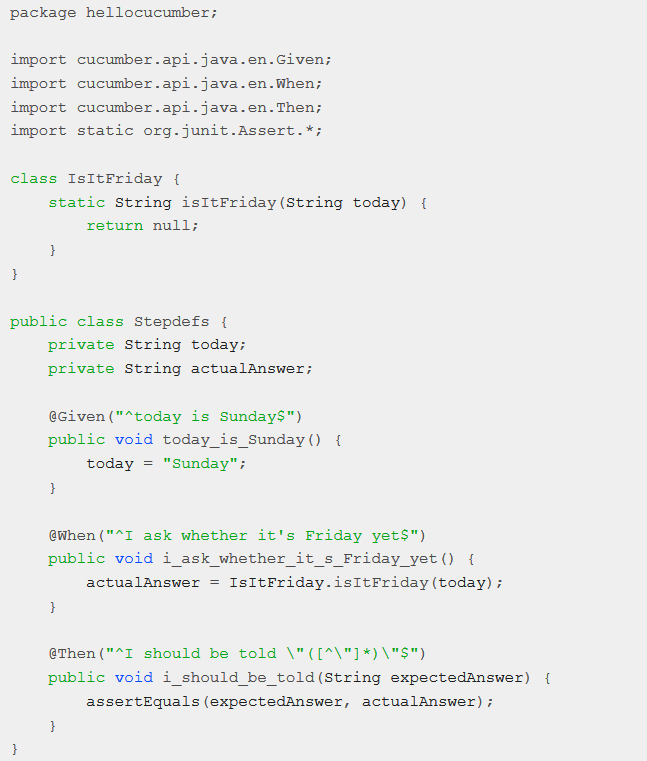
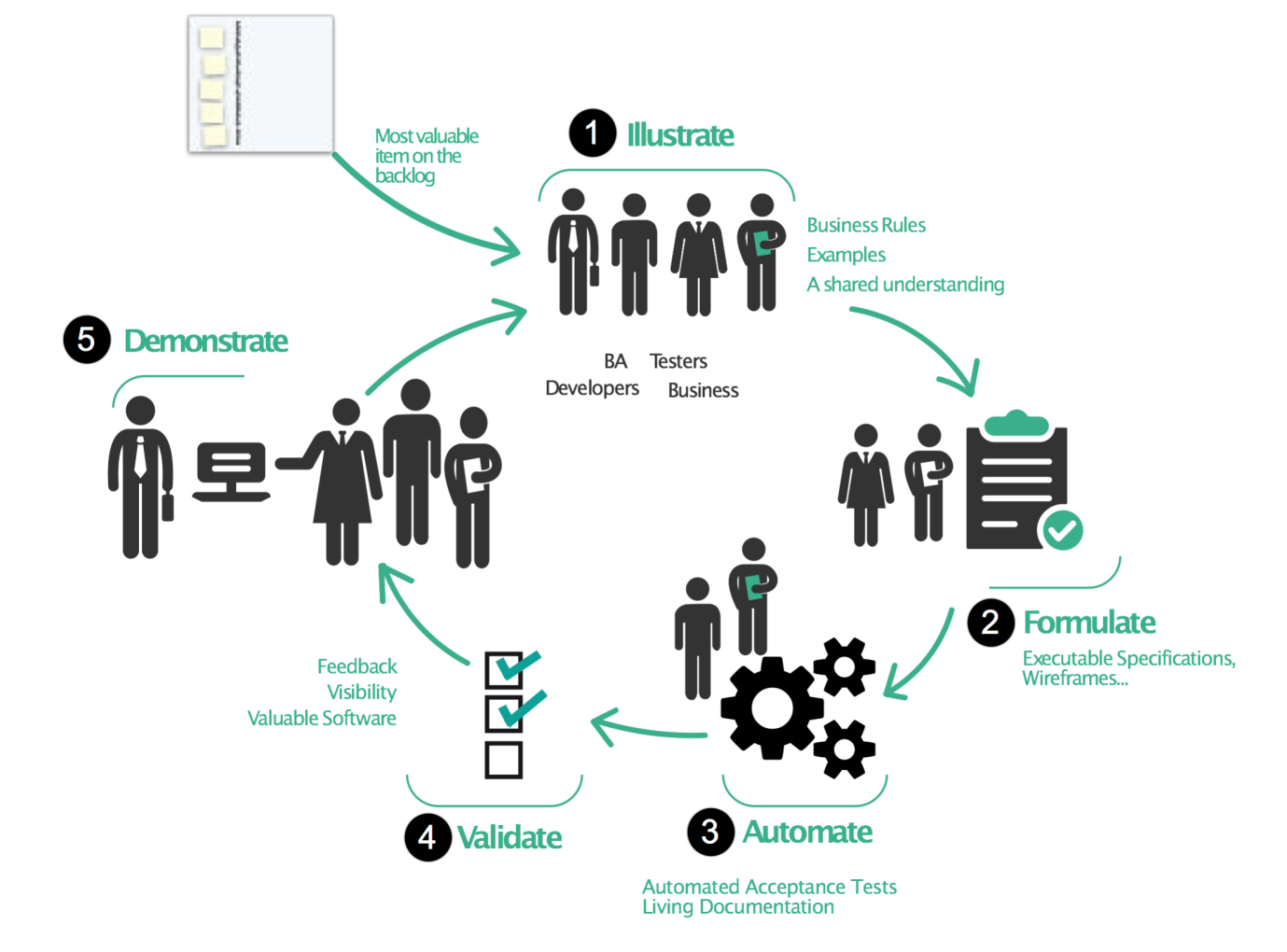


Figure 5: “Is it Friday yet” corresponding steps definition in a java test file

Behavior driven happens at different phases of software development.

* Initial phase: Behavior => Business outcomes
* Analysis phase: Business outcomes => set of features => behavior of the target system
* Implementation phase   
  => Automated acceptance testing => validation of behaviors,  
  => Class and methods names reflecting the behaviors.

BDD and TDD are being adopted widely in the software engineering world because they improve software quality and productivity. (Solis & Wang, 2011)

Figure 6: Behavior Driven Development methodology[[4]](#footnote-4)

How does BDD bring answers to our fined grained problems?

By supporting the UL, BDD enhances the collaboration between development teams and stakeholders. Moreover, BDD relies on the process of gathering requirements and checking if they are well implemented, which is done within an iterative process of automated tests checks. Such process has the effect to drastically ameliorate the quality of the produced software with respect to the cohesion between what is built and what was meant to be built.

As any other valuable notions, BDD come with his own costs.

### Drawbacks of BDD

BDD has some drawbacks that should not be neglected as they can impact negatively the software development lifecycle.

* Time consuming:   
  One of the most important steps of BDD methodology is the edition of the feature and scenario
* files with the Gherkins language. As it requires an overhead investment of time and effort, It may not be worth it for small projects, but for complex projects with lot of iterations, the return on investment of such efforts is quite sure.
* Intensive Communication:   
  There needs to be a good amount of communication between the person writing the feature files and the person developing the automation code. The coder needs to accurately interpret these files and the scenarios in order to implement them as automation steps. If there isn’t a mutual understanding about the structure and approach being used, problems will arise as the scenarios become increasingly difficult to turn into working automated tests.
* High abstraction level needed:   
  Although the concept of Gherkins is simple enough to understand, still the need of specific skills are required to structure scenarios in a way that facilitates the writing of automation code.

## Domain Driven Design (DDD)

### History

Many softwares are made following a data centric architecture. Characterized by a monolithic service layer handling all the business functions that have been implemented, followed by an Implementation layer which discuss with a Data base.

Such architectures, in complex projects, lead to:

* Fat components that are painful to maintain and scale up,
* Shared database which decreases the robustness

The Domain-Driven Design is a philosophy (introduced by Eric Evans in 2003 in his book Domain-Driven Design: Tackling Complexity in the heart of software) consisting of enabling software developers, to effectively manage the construction and maintenance of software for complex problem domains. (Millet & Tune, 2015)

### What is DDD?

#### Definition

DDD helps achieving not only the challenge of understanding a problem domain but also the creation of a maintainable solution that is useful to solve problems. The power of DDD is its strategic and tactical patterns that can be used to deal with the aforementioned challenges.

#### Methodology

The DDD approach is based on 3 main pillars:

-        Collaborative Modeling

-        Strategic Design

-        Tactical Design

The first two items are about helping the project team to discover understand a problem domain, we talk about the **problem space;** meanwhile the last is about giving the teams the guidelines in order to create a maintainable solution that is useful to solve problems: The **solution space**

**Collaborative modeling**

Domain Driven Design (DDD) is a business designing approach driven by the use cases in order to make the implicit explicit. The main goal of Collaborative modeling is to elicit use cases. This is done by exploring the existing, discussing with business people to understand them.

Techniques used are:

-        Event storming

-        Agile workshops

-        Impact mapping

-        User story telling

-        Etc.

Interaction between developers and business guys is highly needed here.

**The strategic Design**

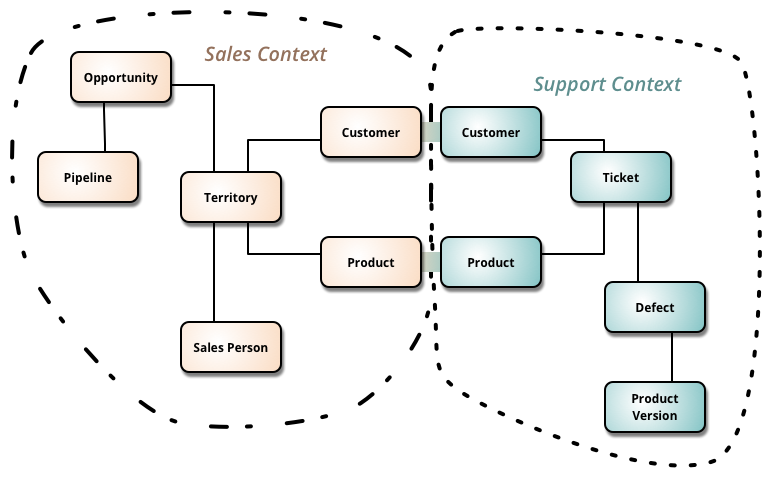
Once the use cases are extracted, they need to be separated within diverse Bounded Contexts (BC), each being independent from the others and characterized, discussed by the UL.  


Figure 6: Two bounded Contexts independent from each other[[5]](#footnote-5)

There must be no misunderstanding between stakeholders and development teams. The BC, is the communication area between the business and the development team. Similarly to the fact that a Product Owner belonging to the functional development team must be on the same wave length with the Product Manager who belongs to the business team.

As a result of applying strategic patterns we end up with:

* Core part of the system identified and isolated so that it can be invested in and evolve independently,
* Big problem domain is split up into subdomains for easier understanding: a model in a specific bounded context is created for each subdomain
* Each context of the entire system communicate through well-defined interfaces
* Each context has its own separated model

Going in depth with the strategic pattern, it consists of:

* Distil the problem domain into **many kinds of subdomains related to a bounded context that is simpler to handle**. This distillation reveals the **core-subdomain** (the reason the software is being written) **where to put the attention the most**.
* Create an **analysis model** (e.g. with UML) and a **code model** (e.g. with Java) for each subdomain, this where the **Model-Driven Development approach** patterns could be used. Models are not created equal as the subdomains complexity differs from one to another. The most appropriate MDA pattern should be found based on the complexity needs of each subdomain.
* Use a shared Language to enhance communication and collaboration.

Since models (analysis and code) are built through the collaboration of domain experts and the development team, communication is achieved using UL. That is the common base where both the analysis model and code model pick their terms and concepts.

The UL is constantly evolving fed by both business teams and development teams. The same term or concept should not be mentioned within the analysis model and in the code model under different names.  Concepts and terms that are discovered at a coding level are replicated to the UL**,** if the business reveals hidden concepts at the analysis model level; this insight is fed back into the UL too, therefore into the code model. This is the key for domain experts and development teams to evolve the model in collaboration**.**

At this stage we understand that DDD and MDA are complementary, DDD cannot be achieved without constructing models with business teams and developers teams.

* Isolate model from ambiguity and Corruption. Subdomains are related to a bounded context to form a protective envelope around models so that the entire software does not evolve in a big ball of Mud (BBOM), likewise, when a modification within a specific model does happen, it does not have an impact on other models.

Code models, often referred as domain models, are isolated from infrastructure code: where the complexity related to technical issues and external libraries are placed, in order to avoid merging technical issues with business concerns.

Tactical Design:

This is the implementation part where the bounded context (the domain model) needs to be implemented following specific software architecture.

DDD relies on a set of patterns that that help implementing models for complex bounded contexts.

They help managing the complexity in the solution by shaping the most appropriate architecture for the application. In a nutshell, they help implementing what has been done during the strategic design stage.

After have defined the bounded contexts, each corresponding to an analysis model that will be translated to a code domain model, we need shape that into a micro-service like Application Architecture.

DDD is a more 16 Years old approach but it is expanding only right now because of micro services.

That’s why they say “The DDD is the keystone of Micro-services”

But, what is a micro-service? Is first of all a context related to a business use case, which will be materialized by a domain model (DM), which it represents a boundary: exactly what have been done during the strategic pattern.

Then the Domain model is related to its specific data management system (database, flux …).

The resulting DDD application will be then a set of micro-services.

Micro services advantages are:

-        Flexibility

-        Maintainability

-        Scalability

-        Polyglot

Keynote: One business bounded context, one micro-service, one domain model, one database.

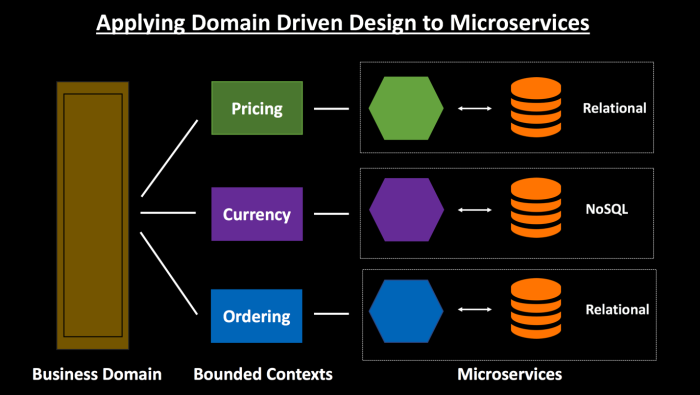


Figure 7 : One business bounded context, one micro-service, one domain model, one database [[6]](#footnote-6)

The DDD relies also on what are depicted as Clean coding patterns:

Some clean architectures:

-        “High cohesion, low coupling”

-        “SRP: “Single Responsibility principle”

-        “Hexagonal Architecture”

-        Inversion of Control, Dependency injection Principle

…

We will consider an implementation with a hexagonal architecture.

A hexagonal Architecture is defined:

* At its center by a hexagon, the biggest part, the most important one, representing the application core, agnostic to any dependency, everything that it needs is defined within this same layer where we put the best developers: This is the DOMAIN layer.
* The hexagon is then protected by a layer named after APPLICATION layer forbidding the domain layer to be touched or modified, on this protection, will be plugged subsequent areas such as the INFRASTRUCTURE LAYER , linking the app to databases or any data management systems (flux, RDMS, NoSQL DBMS …)

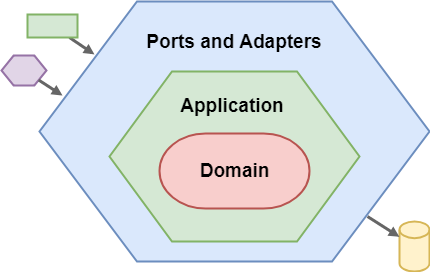


Figure 8: Overview of Hexagonal architecture[[7]](#footnote-7)

Some evangelist, designed hexagonal architecture extensions such as the Onion Architecture.

* Here we have the domain layer at the middle of the Onion where have been defined:
* Domain services: this is the core functions of the system that delivers value to the business
* Domain models: corresponding to the analysis model designed with stakeholders.  
   Domain models does not reflect how the data are managed by the Data management system, the Data model, included in the infrastructure layer is.
* Repositories: interfaces describing how data are retrieved from the data management system which can be a database, a JSON file…),
* Around the domain, we have the application layer, within which domain services are injected in order to be orchestrated and serve to upper layers.

Others layers are above the application layer, they are injected with items from both domain layer and application layer.

They are:

* The Infrastructure layer:

- implementing the repositories/interfaces described in the domain layer,

- defining all services interacting with items beyond the application scope (DB, rest request, external API calls …), every technical related logic resides here, away from the domain logic of the application. This way we avoid the business logic to be corrupted by technical concerns.

* The exposition layer:
* Exposing APIs to the outside of the system, to Clients such as web applications, mobile applications, third party applications.
* The exposition model reflects how data are exposed to API clients, i.e. how clients must consider handling the API calls returned data.

Despite the fact that the models defined at each layer are not intended to reflect each other, they may need to be mapped to each other; Therefore, Adapters need to be defined to do the mapping.

Mapping between Domain model and Exposition Model.

Mapping between Data model and Domain model.

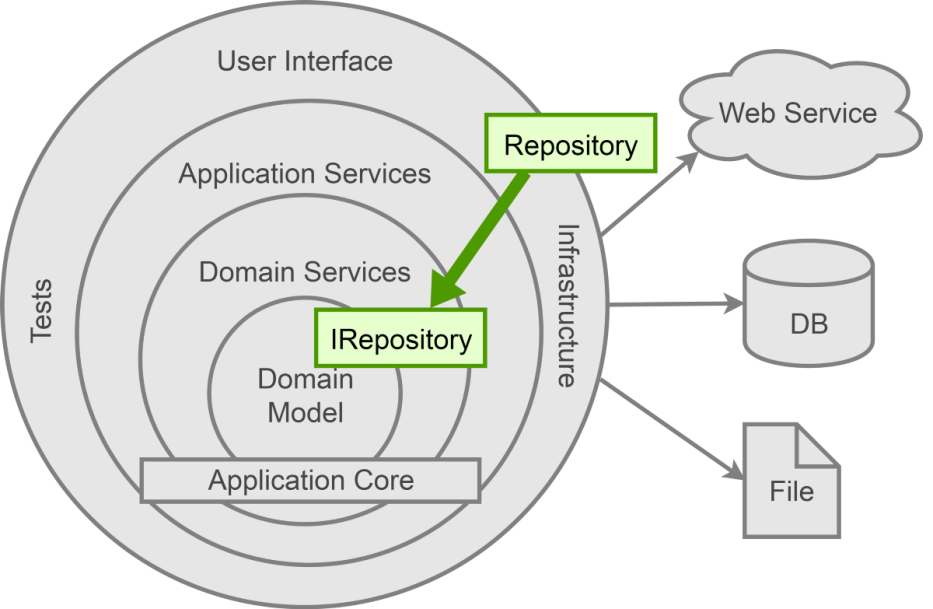


Figure 9: Extended Hexagonal Architecture: The Onion Architecture[[8]](#footnote-8)

How does the DDD bring answers to fined grained questions ?

Unfortunately people (developers ) focus on tactical patterns especially designing architectures, object oriented models, nevertheless, DDD is not a code-centric philosophy, nor a handful of implementation patterns, they forget about the work that have to be done using strategic patterns.

Strategic patterns shape the solution and tactical patterns implement a rich domain model that has been defined.

It doesn’t worth it jumping into tactical patterns without being through the step of strategic patterns.

The Domain-driven design doesn’t only focus on the knowledge of the subject (Collaborative Design), but give us useful ways (Strategic Design) of understanding the subject matter, as well as implementing maintainable and scalable software(tactical designs).

### Drawbacks of DDD

DDD has been first evoked by Eric Evans in 2004 but it’s just on 2013 that we have started noticing activities around that term, with the venue of micro-services.

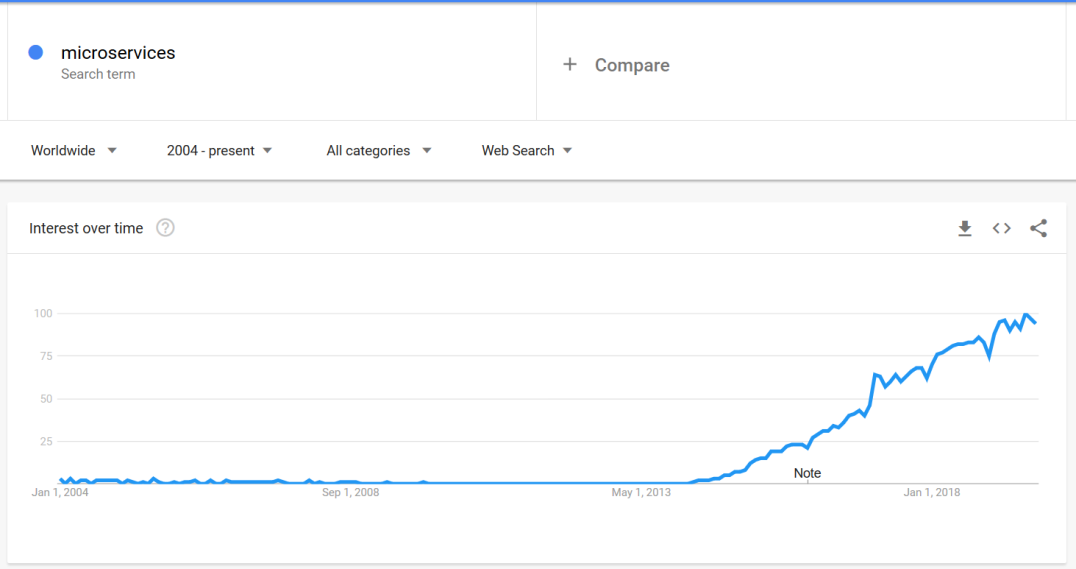


Figure 9: Micro-services searches trend since 2004[[9]](#footnote-9)

This is also due to the fact that it comes with some drawbacks, namely:

* High cost:

DDD Take a lot of effort to implement, moreover, domain expert expensive to hire.

* Need smart and humble developers to learn from the domain expert.
* Waste of time when applying it to simple applications: DDD should be applied only to complex domains not to simple cases applications.
* Easy to do it wrong:  
  Unfortunately, many teams try to implement the DDD in the wrong way by only focusing on tactical patterns; which leads to skipping the DDD step of “distilling the domain” (part of the strategic pattern). Therefore, Sub-domains are not well defined or are not defined at all; Domain Boundaries are then ignored, the core domain on which the best developers and resources should be involved is unknown, the ubiquitous language that is mandatory for the correlation between Analysis model and Code model is not defined.   
  Because Domain boundaries are not clearly defined, software code falls back in the anti-pattern which DDD is trying to fight against: the Big Ball of Mud (BBOM).   
  Because the UL is not defined, code model evolves during iteration without a feedback loop to the analysis model, and developers start diverging from what have been defined by the domain experts as needs at the beginning.  
  With the BBOM, the application becomes hard or almost impossible to maintain over iterations.  
  With Domain divergence, Software works but not for what it is intend to, leading to **Huge amount of money lost.**

## Conclusion

On our way to find means to tackle the problem of designing complex softwares, we have been studying three existing approaches in order to get some clues out of them and rely on it to suggest enhanced approaches. Each of the approaches addresses some specific subjects matter more than the others. However, the main outcomes, with respect to the coarse and fined grained questions are highlighted in the following table.

Table2: Existing approaches with respect to the research problem

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Research problem | Coarse Grained Issues | Fine Grained Issues | Existing solutions | | |
| MDA | BDD | DDD |
| **How to design and develop complex softwares ?** | **How to decide what to build and how to build to provide critical services to thousands of users?**  **Designing issue** | **How to educate the teams and let them notice the importance and the priority of getting aligned with the business?** |  |  |  |
| **How to speak to the business people and to captivate their interest?** |  |  | **Collaborative**  **Design:**  **Event Storming** |
| **How to get the most correct domain knowledge as possible?** | **Platform Independent Models** | **Ubiquitous Language** |
| **How to build and correct quickly what have been decided?**  **Implementation and maintenance issue** | **Which development pattern helps the team to focus on domain issues as described by domain experts?** |  | **Automated Tests** | **Strategic**  **Design:**  **Bounded contexts** |
| **How to separate technical implementation concerns from domain logic issues?** | **PIM to PSM** |  | **Tactical Design:**  **Hexagonal & Onion architecture** |
| **How to apply an architectural style facilitating scalability and features enhancements?** |  |  |
| **Approach concreteness issue** | **Declarative guidelines rather than imperative?** |  |  |  |

We notice that there are some blank holes in the Solution pane of our table, meaning that despite the usefulness of the existing solutions, there are elements of our research problem that still need to be addressed. In order to suggest solutions, we will proceed to a research methodology relying on surveys and feedbacks.

# Research Methodology

## Qualitative analysis

## Quantitative analysis

## Results and comments

# Assessment and validation of results: case of AXA Bank DSI

# Conclusion and further work

# Table of acronyms

BDD: Behavior Driven Design

DDD: Domain Driven Design

ISD: Information System Department

MDA: Model Driven Architecture

PIM: Platform Independent Model

PSM: Platform Specific Model

QVT: Query/View/Transformation

TDD: Test Driven Design

UL: Ubiquitous Language

UML: Unified Modeling Language

XMI: XML metadata interchange

# Bibliography

Ambler, S. W. (n.d.). *Are You Ready For the MDA?* Retrieved September 17, 2019, from agilemodeling.com: http://www.agilemodeling.com/essays/readyForMDA.htm

Atem de Carvalho, R., Luiz de Carvalho e Silva, F., & Manhaes, R. S. (2010, June). *Mapping Business Process Modeling constructs to Behavior Driven Development Ubiquitous Language.* Consulté le April 29, 2019, sur researchgate.net: https://www.researchgate.net/publication/45925747\_Mapping\_Business\_Process\_Modeling\_constructs\_to\_Behavior\_Driven\_Development\_Ubiquitous\_Language

Bhatti, N. S., & Malik, A. M. (2015, April 10). *Model driven architecture.* Retrieved April 29, 2019, from researchgate.net: https://www.researchgate.net/publication/274916541\_Model\_Driven\_Architecture

France, R., & Rumpe, B. (2007, June). *Model-driven Development of Complex Software.* Retrieved April 29, 2019, from researchgate.net: https://www.researchgate.net/publication/4250888\_Model-driven\_Development\_of\_Complex\_Software\_A\_Research\_Roadmap

Millet, S., & Tune, N. (2015). *Patterns, Principles, and Practices of Domain-Driven Design.* Indianpolis: Wrox.

North, D. (2006, March). *Introducing BDD.* Retrieved April 29, 2019, from dannorth.net: http://dannorth.net/introducing-bdd

Solis, C., & Wang, X. (2011, October). *A Study of the characteristics of Behaviour Driven Development.* Retrieved April 29, 2019, from researchgate.net: https://www.researchgate.net/publication/224265781\_A\_Study\_of\_the\_Characteristics\_of\_Behaviour\_Driven\_Development

Thomas, D. (2003, January). *UML - Unified or Universal Modeling Language?* Retrieved September 16, 2019, from www.jot.fm: http://www.jot.fm/issues/issue\_2003\_01/column1/

Wanivenhaus, J. (2017, October 05). *Behavior-Driven Development: The Origins.* Retrieved September 17, 2019, from DZone: https://dzone.com/articles/behavor-driven-development-the-origins

# Appendices

## Semi-structures Interview Guide

## Questionnaire for the quantitative analysis

1. **Stakeholders**, often referred as **business people** or **domain experts** in this paper, referred all to the ones defining how and what to build, they must be kept happy during the project lifecycle. [↑](#footnote-ref-1)
2. Source : https://www.omg.org/mda/ [↑](#footnote-ref-2)
3. Source : https://www.slideshare.net/kalhanrl/cucumber-for-automated-acceptance-testingpptx [↑](#footnote-ref-3)
4. Source : https://johnfergusonsmart.com/so-you-say-you-are-doing-bdd-the-story-of-the-whiteboard-and-the-nail-gun/ [↑](#footnote-ref-4)
5. Source: https://martinfowler.com/bliki/BoundedContext.html [↑](#footnote-ref-5)
6. Source : https://dzone.com/articles/breaking-the-monolithic-database-in-your-microserv [↑](#footnote-ref-6)
7. Source: https://github.com/cwoodruff/ChinookASPNETCoreAPIHexArch [↑](#footnote-ref-7)
8. Source: https://dzone.com/articles/stem-in-onion-architecture-or-fallacy-of-datanbspl [↑](#footnote-ref-8)
9. Source: https://trends.google.com/trends/explore?date=all&q=microservices [↑](#footnote-ref-9)