Tackling Business Complexity in Enterprise Application's Development

A survey about methodologies to construct maintainable systems for complex business scenarios

Capstone Project by

Renê Eduardo Pereira Cardozo

Supervisor

Daniel Macêdo Batista

In Partial Fulfillment of the Requirements for the
Bachelor's Degree of
Computer Science

UNIVERSITY OF SÃO PAULO

São Paulo

2023

ABSTRACT

This work aims to summarize approaches and techniques to model and implement a software application that addresses complex enterprise business necessities. It presents well-established processes for effectively extracting requirements and distilling the main problems from complex domains. It also justifies the necessity of delineating a domain model and advocates against using Transaction Scripts and a procedural approach in those scenarios. Lastly, a series of design patterns and abstractions are described and associated with the main goal of reducing complexity and isolating the core domain code from other concerns.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

Chapter 1

INTRODUCTION

The fast-paced business environment of enterprises results in their systems being faced with complex and rapidly changing requirements. This makes it challenging to create software that can keep pace with business needs while maintaining high reliability and scalability levels.

One of the main challenges in this context is how to avoid the software complexity rising exponentially as the system reflects complicated business rules, exceptions, and even conflicting requirements from different stakeholders. Also, as all big companies today have a digital infrastructure as their backbone, many changes in how their business is done are reflected as software changes. Integrating the internal system with many external services is also necessary.

Without proper care, the stacking effect of system changes can span a massive quantity of branching in the flow of information. It can build a gigantic net of possible paths that can make the system essentially unpredictable for any developer. It also can generate a coupled architecture where a change in one module of the system can require changes in many other modules.

The system can become unmaintainable, and a fear of changing the application could quickly spread over the development team. As this happens, the system's abstractions become incompatible with the actual business activities. The productivity of the technical staff may dump, bug quantity can start to rise, and their work may become inconsistent, promoting distrust between them and the rest of the company. A culture of distrust is appointed by the book Accelerate [1] as a significant indicator of low-performing technology organizations.

Different techniques can be used to avoid those problems. One is agile software development, which embodies change as a certainty rather than something sporadic. Others include using a high quantity of tests and pipelines of continuous integration and continuous development (CI/CD) to allow faster feedback cycles between developers and stakeholders.

This work is about another approach, focusing on the software architectural decisions that allow a system to evolve consistently and how to avoid coupling its different

modules. Its objective is to compile different techniques and common decisions in the architectural process of high-scale enterprise applications. Those decisions need to reflect the business rules, terminology, and expectation in the software system and define technical abstractions about organizing a complex application.

The main topics of this text are derived from the book Domain-Driven Design (DDD) [2] written by Eric Evans in 2003, as well as complementary ideas that have been born around its work. It also uses the main architectural concepts delineated by Martin Fowler in Patterns of Enterprise Application Architecture (PoEAA) [3] from 2002.

The code examples are presented in `Go`, due to the simplicity of the language, common use in systems programming, and the similarity of its syntax with C-like languages.

The rest of this document is organized as follows: Chapter 2 presents the definition of complexity in the business context considered in this work. Chapter 3 and Chapter 4 discuss an approach to designing an effective and specific solution for a complex business problem using knowledge crunching techniques and domain modeling. Chapter 5 gives examples of code and system abstractions that allow the implementation to be flexible and to cope with the complexity of the domain model and changing business requirements. Chapter 6 concludes the monography.

Chapter 2

MAKING SENSE OF COMPLEXITY

Foreseeing future events and outcomes in a rapidly changing environment becomes quite tricky. The systems that emerge from those environments aren't ordered and cannot easily predict their outcomes. A slight variation in initial conditions yields a drastically different output.

2.1 The Cynefin framework

The Cynefin Framework splits problems into four domains: simple, complicated, complex, and chaotic. The diagram in Figure 2.1 depicts the Framework.

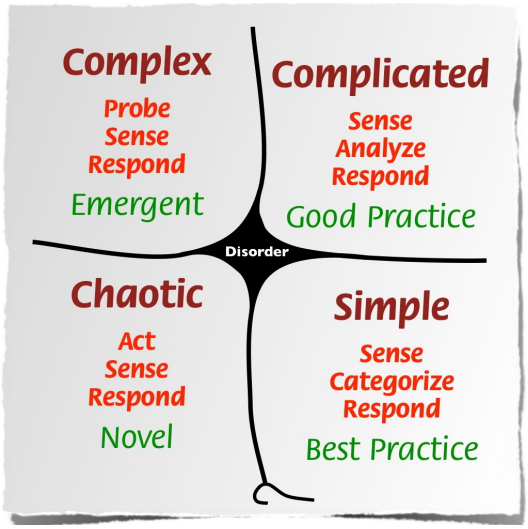

Figure 2.1: Cynefin diagram by Dave Snowden ©

A disorder represents the inability to classify a problem into the four aforementioned categories. A little fold at the bottom of the diagram indicates that simplicity can quickly become chaotic [4].

The right-hand side of the diagram represents the ordered domains. Simple (or clear) problems can have their cause and effect instantly associated. The problem is sensed, categorized in a strict category, and treated following a well-established solution. A typical forms-over-data application, where the user interface is a thin interface of forms to be persisted, is a simple problem. The CRUD solution is

considered the best practice for this situation; every operation needs to fall into the *create, read, update* and *delete* schemes.

The complicated domain is predictable, but the path between cause and effect is not as straightforward as for simple domains. It demands some analysis before responding to the problem. One example of a complicated system is a relational database management system (RDBMS). Building such an application requires much effort and analysis; some trade-offs must be assessed, and choices must be made. But there are well-known good practices such as B-trees, indexes, and whole decades-old theory that backs relational models. Therefore, the decisions are guided by principles [5] [6].

The complex domain is characterized to be highly composite, with a multitude of interactions between its subunits. A feedback property and an emergent behavior characterize those systems. Cause and effect can be perceived in retrospect but not in advance. To assess these problems, it is necessary to experiment and observe (probe) and see what kinds of patterns and behaviors they exhibit before coming up with a response.

The chaotic domain looks random from the outside because there is no relationship between cause and effect at a systems level. It's always out-of-equilibrium, so emergency action is needed to derive some coherent state to stabilize the situation in one of the other three domains.

2.2 The abstraction layer

Given the right granularity level and interpretation depth, the four domains from Fig. 2.1 could be used to classify many problems. The heating exchange between two metal plates can be simple from a real-life scale and ordinary temperatures, described by an accurate mathematical solution. It can be complicated in the context of thermodynamics, complex from a statistical mechanics perspective, and chaotic in quantum mechanics, where randomness is an inherent factor.

So the level of abstraction is essential to contextualize the mean of complexity used by this text. The concerns of this monography are in the level of software architecture and software design, in the layers between the global requirements, such as infrastructure, architectural patterns, and integration patterns; the local requirements that govern a solution, namely design patterns, OOP principles, and project folder structure.

More specifically, the problems treated here are those of enterprise software,

business-based systems built to fulfill an organization's necessities. The kinds of systems that belong to this classification are inventory management, payment, customer relationship management, enterprise resource planning, human resources management, networking, and a plethora of systems dedicated to managing business concerns.

Martin Fowler describes it as:*" Enterprise applications are about the display, manipulation, and storage of large amounts of often complex data and the support or automation of business processes with that data"*[3].

Enterprise businesses are inherently complex. They are built aiming to support a myriad of requirements that arise from business activity. The art of doing business requires constant negotiation and adjustments to cope with the different necessities of clients. This flexibility leads to rules that are not so clearly defined and could be full of exceptions.

A situation where we need to parse and identify fields of commercial contracts – including contractor names, due dates, locations, or signatures – needs to deal with a large variety of contract templates that could be used. Even though they follow a standard template to not only parse this information but interpret the logic associated with it, it is probable to become lost in the exceptional agreements and clauses made to achieve a deal.

The programming world is ordered, so to treat a complex problem, we need to be able to extract its relevant parts to the complicated or simple Cynefin's domains. To describe something complex programmatically is to describe the observed emergent patterns from it in an ordered way.

2.3 Achieving order

Humans typically use patterns to force a degree of ordering into complex scenarios. Patterns are actively extracted, not passively discovered. As the anthropologist Mary Douglas described:

"...whatever we perceive is organized into patterns for which we the perceivers are largely responsible. . . . As perceivers we select from all the stimuli falling on our senses only those which interest us, and our interests are governed by a pattern-making tendency, sometimes called a schema. In a chaos of shifting impressions, each of us constructs a stable world in which objects have recognizable shapes, are located in depth, and have permanence. . . . As time goes on and experience builds

up, we make greater investments in our systems of labels. So a conservative bias is built in. It gives us confidence" [7].

It is, therefore, crucial to "distillate" the primary concepts pertinent to the problem discussed in the same manner. Crude oil is heated to separate the products relevant to each necessity. Only the essential elements of the problem are extracted from the complex reality. This way, the assortment of relationships between the domain entities can be restricted, and we can adequately describe the domain logic algorithmically.

In the previous paragraph and hereafter, the meaning of domain will differ from Cynefin's attribution. It will be considered that the domain is the *"...sphere of knowledge, influence, or activity. The subject area to which the user applies a program is the domain of the software"* [8].

Chapter 3

KNOWLEDGE CRUNCHING

Effective application designers are essentially "knowledge crunchers" in the way that they can transform the torrent of information and associations into abstract concepts. This conversion, although, can never be obtained by oneself. Cooperation between business experts, developers, and architects is the key to properly describing the domain.

Domain experts are the ones that truly understand the domain. They are effectively the advanced users of the applications to be developed, and typically have years of experience in the area. However, the developers and architects have the technical knowledge to translate requirements into implementation and assess a solution's viability.

3.1 Binding business and implementation

Suppose an application is modeled without proper alignment with business experts. In that case, the developers will converge to the faster-to-built system that can solve the subject, trying to guess the real necessities of its users and achieving abstractions that do not necessarily correspond to real entities.

The issue with this approach is tied to the fact that business necessities will continually change due to innovation, competition, change of mentality, or anything related to the volatility of human behavior. To cope with this transient nature of enterprise business, the system must have its abstraction intimately related to the business rules that govern the field. To accomplish that, it's necessary to establish a model for the domain, that describes the business. This model will never be completed, though.

The nature of constant change is intimately related to the principles of Agile methodologies [9] [10] and Continuous Improvement in the way that the model will always be discussed and refined between developers and business specialists to capture alterations in business rules and entities. The team that defines the products and develops the solution will not be segregated, and a shift in mentality is required. The system must no longer be perceived as a project with an end in its development but as a product that is constantly evolving and adapting to the business necessities [11].

The team, from now on, will be defined as composed of technical and domain-specific staff. The whole group will constantly communicate and review the model, so the system can properly reflect the problem that is trying to be solved.

3.2 Ubiquitous language

A critical aspect of knowledge crunching is creating a common language, known as the "ubiquitous language," used throughout the development process to ensure that everyone speaks the same language and works towards the same goals.

The goal of the ubiquitous language is to create a shared understanding of the domain among the team members and to ensure that the software is being developed accurately. The system must reflect the common language in its software abstractions, so changes in business rules could be directly related to changes in the respective software correspondence.

It is important not to underestimate the importance of the ubiquitous language in software development and domain modeling. Having a shared understanding of the domain is critical to the success of a software development project. It helps to ensure that the model accurately reflects the underlying complexities of the domain and that it can be effectively used to inform the design of the software.

Without a common language to describe the domain, it can be harder to identify the root causes of problems and find solutions. The abstractions used in the technical field can differ from the concepts used by domain experts, and changes in the business domain are trickier to reflect on the system.

As the team gains a deeper understanding of the domain and as the model evolves, it is important to refine and update the ubiquitous language continuously. This will help to ensure that the language remains accurate and relevant as the project progresses. It can be helpful to make the ubiquitous language visible to the team by creating a glossary or reference document that defines the terms and concepts used within the domain. This can help ensure that everyone on the team uses the language consistently.

Establishing a ubiquitous language requires gathering and synthesizing information about the domain, and identifying key concepts and terminology. Those details are best obtained by mobilizing the team to aggregate domain knowledge as a model.

3.3 How to aggregate domain knowledge

The discussions to describe the domain in terms of a model can be formulated in various ways, typically involving interactive meetings or workshops. Two examples of those meetings are User Story Mapping and Event Storming:

User story mapping

User stories are an agile way of describing requirements that focus on a narrative way of reporting the user journey throughout the system to operate some functionality [12]. They are designed to shift the emphasis from writing about features to discussing them. Ron Jeffries, one of the founders of Extreme Programming, separates user stories into three aspects, the three C's [13]:

- Card: user stories are written in cards, that contain just enough text to serve as a reminder to the team of what the requirement is. The card is a token that represents a requirement.
- Conversation: using the cards just as the basis for the discussion, it is in the conversation that the team ultimately communicates business requirements. Documents can be used as an auxiliary, but verbal communication properly defines the features.
- Confirmation: to assess that the features make sense and are consistent with the problem space, examples usually describe a specific user's journey. These examples can then be transformed into acceptance tests that, not only, give shape and justify abstract ideas, but also give a definition of done to the features to be implemented.

User Story Mapping [14] is a visual exercise idealized by Jeff Patton to help the team describe the different possible user journeys as user stories. Mapping the interactions as user stories help to find out what is really necessary for the system and which features should be prioritized. Figure 3.1 portrays an example story map.

The workshop is organized with the team using a big canvas, where different sticky notes will be the cards that define the story.

First, the team writes the major user activities that will be performed in the system and the type of user that will do it, without details of how this task should be done or alternative paths it could take. At the same time, the team organizes the tasks,

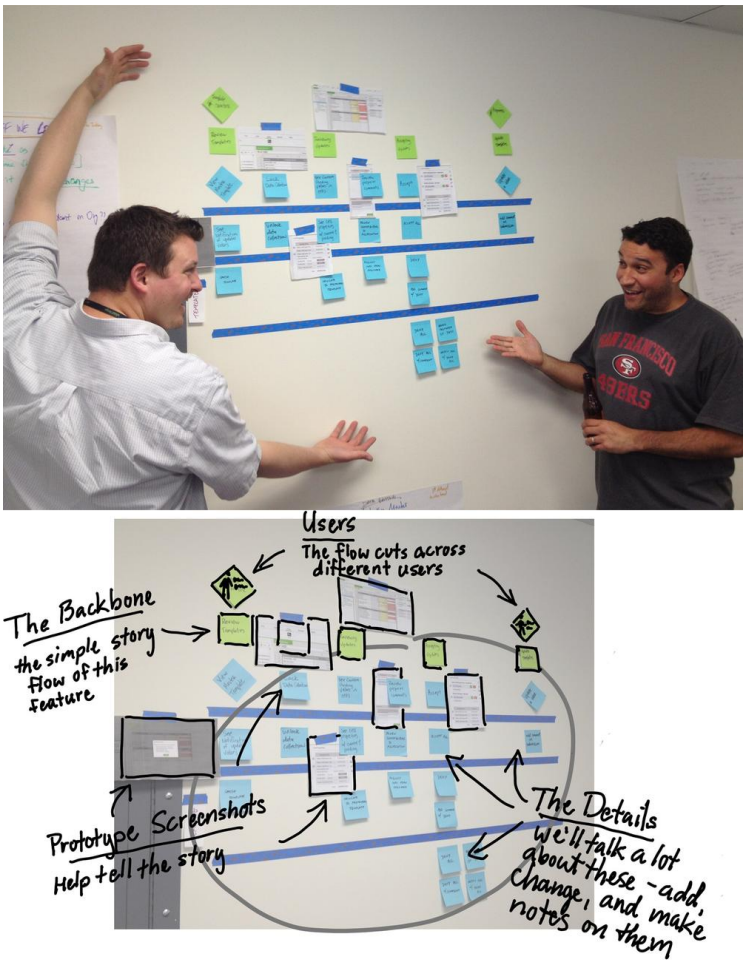

Figure 3.1: Example of a story map [14]

trying to establish a priority between them. Once a substantial number of tasks – relative to the system's complexity – are delineated, the team should start to outline a primary function of the system, a simple story describing the most crucial feature the system must implement. The objective at this stage is to find functionality that can aggregate the most value to its customers. The process steps are not linear at any point, so if the team finds out that the story lacks middle steps to make sense, they must add this activity to the story.

Once the story is established, its cards are stuck in the canvas from left to right in chronological order. Those cards set out the backbone of the story map. This backbone provides a functional description of the feature aggregating the most value to the business. The users will be represented above the backbone to highlight which customers will be affected by the functionality and to which part they are related.

The design strategy is iterative and incremental. Based on the backbone, the details necessary to accomplish the functionality are outlined. They are sliced out vertically into sections that increase in granularity and decrease in priority. In the first segment, there will be the minimum number of features necessary to see it working below the part of the backbone that needs it; this is the minimum viable product (MVP). Below are the features that could improve the product or are nice to have, decreasing the level of importance at every division. Every section, following horizontally, must depict a proper functional product that could be used by the users illustrated on the top of the canvas.

The major benefit of this approach is that it enables the team to discover the product as it tries to describe visually the customer journey based on a real situation, providing a good model of the problem's domain. As Gene Kim highlights:*"When engineers think of "the customer" in the abstract instead of as a real person, you rarely get the right outcomes"* [15].

Once a discussion over a specific story map is completed, a new story could be selected from those written previously. It continues until the team is satisfied with the number of functionalities described. The meeting is repeated as new demands emerge.

Event storming

A different workshop on extracting requirements from the business domain is Event Storming [16]. The method was invented by Alberto Brandolini, based on the principles of Domain-Driven Design, aiming to extract all relevant events of the domain and to establish a common language between domain experts and the technical staff.

This approach uses the same materials as the User Story Mapping: a big canvas and colored sticky notes. The meeting is guided by a facilitator that ensures that it's following the proper steps and the discussion is not diverting from its objective. The process usually involves the following steps:

1. Identify the focus of the event storming session: The first step is to define the scope of the event storming session. This might be a specific business process, a customer journey, or some other aspect of the domain.

2. Gather the necessary participants: To be effective, an event storming session should include domain experts, developers, and other relevant stakeholders. It

is essential to have a diverse group of people to get a wide range of perspectives and ideas.

3. Set the stage: Event storming sessions typically take place on a large wall or whiteboard, with sticky notes or other visual aids to represent events and processes. The group should also decide on the rules and conventions for the session, such as how to identify different types of events and how to represent them on the wall.

4. Start brainstorming: The group should begin by listing all of the events within the scope of the event storming session. These events might be actions taken by users, system events, or other types of occurrences.

5. Organize the events: Once the group has identified a list of events, they should work together to organize them into a logical flow. This might involve grouping related events or creating sub-flows for different parts of the process.

6. Identify relationships: As the group organizes the events, they should also look for relationships and dependencies between the events. For example, one event might trigger another, or specific events might be conditional on the completion of other events.

7. Refine and iterate: As the group works through the Event Storming process, they will likely uncover new insights and ideas. They should use this information to refine and iterate on the model, adding or removing events as needed.

8. Overall, the goal of event storming is to create a comprehensive and holistic view of the domain, which can help to identify opportunities for improvement and inform the design of the software being developed.

3.4 Bounded contexts

In Domain-Driven Design, bounded contexts divide a domain, or subject area, into smaller, self-contained parts. Each bounded context has its own language and model and operates independently of the others within the domain. Bounded contexts can help manage complexity and ensure that the model accurately reflects the underlying domain.

During an event storming session, one way to identify bounded contexts is to look for clusters of events related to a specific part of the domain. Those events' spatial

Figure 3.2: Event Storming canvas of a shopping cart system where the association of events revealed bounded contexts [17]

separation and adjacency give us a clue about which parts of the system can be treated as a bigger abstraction unit.

In Figure 3.2, there is a canvas obtained after a session of event storming of a shopping cart system. The events were associated organically in the workshop and slowly started to create clusters that could be identified as bounded contexts.

In the contemporary context of microservices architecture, bounded contexts can help identify the boundaries between different services. Each microservice or group of microservices can be designed to handle a specific bounded context. This can help to ensure that each microservice has a clear purpose and is focused on a specific use case of the domain.

Chapter 4

THE NECESSITY OF DOMAIN MODELING

Enterprise applications are particular because they usually do not involve CPU-intensive tasks but are primarily data-driven. Those information-rich systems mainly consist of data processing pipelines with generally simple operations.

4.1 Layered architecture

Seeking reduced cognitive load, better reusability, modularity, and testing capacity, the industry converged to divide the application responsibilities following a layered architecture. The essential principle is that elements of the layers depend exclusively on their components or from the layer beneath them. It embodies the separation of concerns design rule [18]. Usually, the application is divided into four layers [2]:

- User Interface: render views used to interact with the application, responsible for associating user inputs with application commands.
- Application Layer: orchestrate the required calls to business objects based on the instructions given by the user interface. It acts as the public API of the system. However, it doesn't contain business logic.
- Domain Layer: enforce the business rules of the domain. It's the heart of the application, including a collection of domain objects, their specific operations, and dynamics.
- Infrastructure Layer: wraps up the system-to-system interactions of the application. Controls input and output operations besides those originating from the user interface. Examples are abstractions over database interactions, messaging systems, and third-party APIs.

Variations of this structure exist in many forms, but a central piece of a layered architecture is separating the domain logic from other application concerns. The domain objects remain free of presentation, persistence, or technology-specific concerns. This way, the business rules could evolve at their pace without being affected by technical decisions and focusing on business demands.

Embedding business logic into presentation elements or persistence concerns into business objects is sometimes the fastest way to build software. Still, it leads to a dangerous path of domino effects that span all application layers for even the slightest change in business rules, called the ripple effect [19]. Testing logic becomes difficult, and changes to the UI or database can influence how business rules are implemented, yielding hard-to-trace bugs and tightly coupled code.

4.2 The domain layer

The core layer of an enterprise application is the domain layer, which contains the main business rules of the system, representing the domain problems through abstract entities.

There are two dominant approaches to organizing the domain layer: transaction scripts and domain model [3].

Transaction scripts

Each request from the user interface or other system can be modeled as a query for some data or a command to perform some action. Those interactions are implemented as transactions, where each is handled by a different procedure containing the whole logic of a specific use case. The transaction scripts provide a thin service layer between the user interface and the data layer, usually making direct calls to the storage system or through a simple Data Access Object.

In a simple example, a transfer between two bank accounts can be described by the following procedure:

```
func transferFunds(fromAccountNumber string,
        toAccountNumber string, amount Money) error {
    from, err = AccountDAO.findByAccountNumber(fromAccountNumber)
    if err != nil {
        return fmt.Errorf("couldn't find account number: %s",
                                            fromAccountNumber)
    }

    to, err = AccountDAO.findByAccountNumber(toAccountNumber)
    if err != nil {
        return fmt.Errorf("couldn't find account number: %s",
                                            toAccountNumber)
```

```
    }

    if from.balance < amount:
        return errors.New("insufficient Funds");

    from.balance = Money.Sub(from.balance, amount)
    to.balance = Money.Add(to.balance, amount)
    return nil
}
```

This approach may be ideal if the system only requires a small amount of logic once its linear and procedural style allows for easy code comprehension by other developers and fast pace feature development.

The downsides of this approach show up when business complexity starts to rise. In those contexts, the transaction scripts' lack of reusability and flexibility becomes evident. A financial application may not only have a `transferFunds` procedure but a lot of different operations related to the account's balance, like `billPayment`, `creditCardPayment`, and `withdraw`. To apply validation to input data and balance state, each one of those procedures may duplicate the code to implement it. Moreover, the scattering of common logic among various procedures can lead to inconsistencies between them, generating a lot of possible sources of bugs.

Domain modeling

The domain modeling pattern uses a more object-oriented way of modeling logic to avoid the drawbacks of the procedural approach of transaction scripts. The domain is divided among entities from the domain represented as objects. Those domain objects not only store data in memory but also include the behaviors related to domain logic, which include validation, calculations, and business rules.

Incorporating behaviors into domain objects is essential to the benefits of domain modeling. The absence of behavior to implement business rules and the delegation of this logic to a service layer's external procedures will generate Transactions Scripts without the advantages of a procedural approach. This anti-pattern receives the name of Anemic Domain Model [20], in contrast to the Rich Domain Model that binds data and behavior.

The previous example of a transfer between two bank accounts can be modeled in

conformance with domain modeling this way:

```
type Account struct {
    balance Money
}

func (from *Account) transferTo(to Account, amount Money)
                                                    error {
    if from.balance < amount:
        return errors.New("insufficient Funds");
    from.balance -= Money.Sub(from.balance, amount)
    to.balance += Money.Add(from.balance, amount)
}
```

The encapsulation of data and behavior in the class `Account` helps to ensure that the Account object domain always maintains a valid state regarding the operation. It also complies with the Separation of Concerns principle, once the only items present in the method are the business rules and the entities involved in the operation are abstracted as real Domain Entities, such as `Account` instead of an account number. The data access layer is separated from the domain logic and focuses on representing the domain's concepts rather than on specific use cases.

Effective domain modeling does not come without costs. It requires previous knowledge crunching to design the domain objects correctly. Moreover, the concerns about data storage and communication with external systems are not gone. They are just abstracted to new layers of patterns such as Repositories, Ports, and Adapters [21]. With more patterns to follow comes a more complex architecture at first hand, with more classes and layers, but overall, with changes in business rules and the addition of features, domain modeling tends to pay off fast.

The great triumph of Domain Modeling is to isolate domain logic from all other concerns, creating a solid core that could be easily modified, extended, and used in different contexts and infrastructures.

Chapter 5

TACTICAL PATTERNS

The domain model obtained from the workshops describes the bounded contexts, events, and main domain objects, as well as the business processes associated with them. Establishing a common language contributes to the correspondence between domain objects and their real implemented objects in the system. The question that remains is: "Who about the implementation?". The alignment between business experts and the technical staff is an important step toward the consistency of the application with the real world. Still, the effective realization of those concepts as code involves another variety of technical challenges.

Tactical patterns are a series of design patterns from Domain-Driven design that provide specific solutions to common technical problems while modeling the domain. These patterns help developers to implement the domain model in the codebase, and make it more maintainable, flexible, and extensible.

They include different forms of organization of the codebase through abstraction layers, encapsulating complexity into smaller and more manageable pieces, helping to maintain consistency and integrity of data, and improving the scalability of development by adding new functionality without having to change existing code.

5.1 Domain object representation

In the enterprise world, business processes typically involve some flow of information. Different components of the system request and give commands to each other; they are associated. In an object-oriented approach, this translates to relationships between different objects. Those relations must describe a behavior that is consistent with the association in the model.

There are two essential types of domain objects: Entities (Reference objects) and Value Objects.

Entities

To represent a person in an application, it is not possible to describe it only with name or birthdate. A person is not defined by their attributes. Two persons can have the same name and age, or even the same person could adopt a different name. Each

person has an identity associated with it. Even in the case of a description through a social security number, this number is essentially the correspondent of the person's identity in a government system.

An object that needs to be distinguishable from others even though it has the same attributes is called an entity. An interesting characteristic of those objects is that they have a lifetime associated with them. The objects' state and attributes could change over time but remain with the same identity. Examples are customers and orders in e-commerce, university students, and social media posts.

In OOP languages, an identity operation typically identifies the same object in memory. Still, that form of identity tracking is too fragile when considering persistence and a distributed system. Establishing a property that correlates to the object's entity is necessary. For a person, it could be the SSN, a government-backed identifier. Nevertheless, instead of relying on external identifiers, each system typically delegates its own internal ID to its entities, which is guaranteed to be unique among all instances, generally an increasing sequence of integer IDs or a random UUID.

A bank account in a financial system would be an entity, once it could not only be identified by its attributes, and it has a lifecycle defined by the transactions executed from or to it, changing the object state. An example of a bank account implementation is presented below:

```
type Account struct {
    ID uuid.UUID
    Balance float64
    Owner string
}

func (a *Account) Deposit(amount float64) {
    a.Balance += amount
}

func (a *Account) Withdraw(amount float64) error {
    if a.Balance < amount {
        return fmt.Errorf("Insufficient funds")
    }
    a.Balance -= amount
```

```
    return nil
}

func (a *Account) Transfer(toAccount *Account,
                                    amount float64) error {
    if a.Balance < amount {
        return fmt.Errorf("Insufficient funds")
    }
    a.Balance -= amount
    toAccount.Balance += amount
    return nil
}
```

Value objects

Although entities are an important aspect of domain objects, there are objects which do not need an identity. Even if instantiated as different memory objects, those objects are considered equal if they have the same attributes. They are concerned only with what they are, not who or which they are. This kind of object is called a Value Object.

The nature of value objects is intrinsically transient in the application's runtime. They are rapidly created, transmitted, and discarded. One important point in modeling value objects is that they must be immutable. The only way to change an attribute that contains a value object is to replace it with a newly constructed value object. In the OOP world, value objects don't have a setter; they only have full constructors. It follows copy-on-write semantics.

The immutability of value objects prevents problems related to aliasing. Aliasing happens when two or more variables or attributes access the same memory-allocated object. Modifying data from the perspective of one variable implies a change in the state of another variable. This could happen when an object is passed as an argument to be processed by other functions, or two entities share the same object as its attributes. By being immutable, a change in a variable storing a value object would necessarily incur the instantiation of another value object derived from it, avoiding the risk of an inconsistent state in the system.

When space is critical, and the system has a huge memory footprint, the numerous redundant value objects could be reduced to a reference to just one object shared

between different entities without much concern about aliasing.

For the most part, when declaring entities' attributes, the use of value objects is encouraged in contrast to the use of primitive types. Value objects allow for easy property extension and permit attributes to implement their validations and have a consistent state.

The previous `Account` struct uses a `float64` to store balance and amounts. Representing them as a `Money` value object would be better. One potential extension to a financial system is for it to accept different currencies when it evolves. When the attribute `Balance` is wrapped in a value object declaration, the concerns of checking the validity of operations, and internal state consistency are encapsulated inside the `Money`, instead of being scattered through the different procedures of the entity using it.

An example of a Money value object would be:

```
type Money struct {
    amount   float64
    currency string
}

func NewMoney(amount float64, currency string) *Money {
    return &Money{amount: amount, currency: currency}
}

func (m *Money) Add(money *Money) (*Money, error) {
    if m.currency != money.currency {
        return nil,
        fmt.Errorf("Cannot operate with different currencies")
    }
    return &Money{amount: m.amount + money.amount,
                  currency: m.currency}, nil
}

func (m *Money) Subtract(money *Money) (*Money, error) {
    if m.currency != money.currency {
```

```
        return nil,
        fmt.Errorf("Cannot operate with different currencies")
    }
    return &Money{amount: m.amount - money.amount,
                  currency: m.currency}, nil
}

func (m *Money) Multiply(multiplier float32) *Money {
    return &Money{amount: m.amount * float64(multiplier),
                  currency: m.currency}
    }

func (m *Money) String() string {
    return fmt.Sprintf("%.2f %s", m.amount, m.currency)
}
```

Aggregates

A domain object is represented by associating multiple entities and value objects. Each object has references to the other, and the state of the overall system must remain consistent.

Invariants are business constraints that must apply to the system to be a reliable model of the domain. One example is that purchases on a credit card should never surpass an established limit, independent if the operation is paid by installments or on sight.

In a relational context, it's common to use transactions to maintain the data's consistency. In this manner, every change to the database is treated as an encapsulated unit of work, either totally applied or not.

Aggregates are a pattern that clusters associated objects as a unit for data operations.

The aggregate has a root entity object that spans a tree of referenced objects. Those objects could be value objects or entities with a local identity concept. But the only manner of referring and making changes to all objects pertaining to some aggregate is to do it by the root object. Outside references to the data contained in the aggregate could only be made through the root entity. This includes database operations, references in other aggregates, and use as a dependency by other system modules. When a change to any component of the aggregate is committed, all

business invariants must be verified and satisfied. Figure 5.1 shows the example of two aggregates.

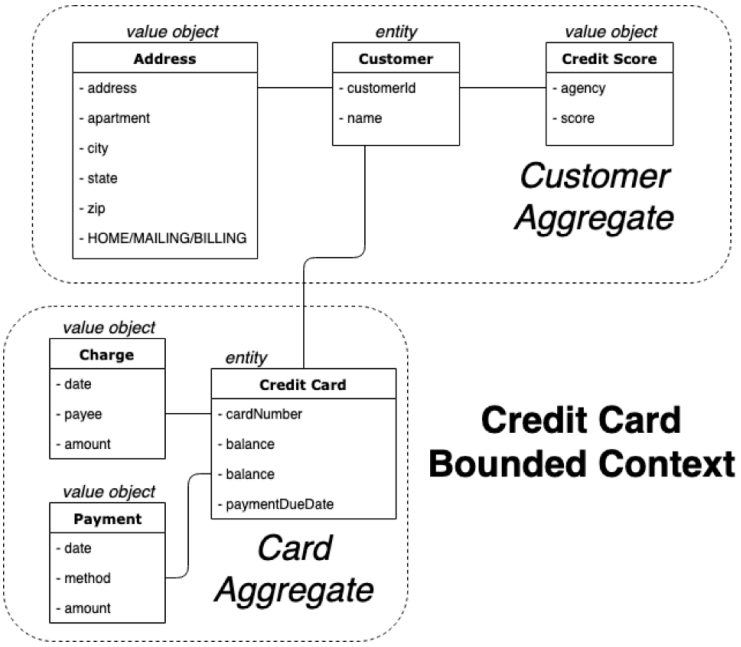

Figure 5.1: Example of two aggregates composing the domain objects of a credit card application [22]

5.2 The life of a domain object

One of the intrinsic characteristics of an entity is that its identity gives it a lifecycle. An object is created, modified, and eventually deleted. Those processes could be quite complex, and many options and possible configurations can be adopted for each situation. In those situations, two useful abstractions could better manage the object's lifecycle: factories and repositories.

Factories

The creation of complex objects could encompass a variety of configurations and different options for their attributes and concrete dependencies. This creation logic could quickly clutter the business rules and the effective state change operations of the domain. Moreover, the responsibility of object creation shouldn't be delegated to the client code, as it could obscure the client's logic. The responsibility of object creation remains at the domain layer. Still, this task could be passed on to a new element of the domain, a factory.

A factory encapsulates all logic and options involved in the domain object instan-

tiation. It isolates the creation logic from the core domain object and the client code. This abstraction is described by the Gang of Four as three types of creational patterns: factory method, abstract factory, and builder [23]. Independently from the implementation strategy adopted, the factory must only allow the creation of an aggregate in a valid state and is responsible for enforcing that inconsistent parameters are rejected.

Repositories

To avoid muddling the domain code with persistency concerns and effectively enforcing that access to any object internal to an aggregate must be done through the root, a repository could be used. A repository abstracts the storage concerns as a collection in which objects are added and removed, and the machinery behind the repository hides the implementation concerns of those operations. The client code uses only terms of the domain model to retrieve aggregates, the number of objects, or even statistics about objects that meet some criteria.

Repositories can be implemented using hard-coded queries with specific parameters, like:

```
type AccountRepository interface {
    AddAccount(account Account) error
    GetAccount(id uuid.UUID) (Account, error)
    UpdateAccount(id uuid.UUID, account Account) error
    RemoveAccount(id uuid.UUID) error
    GetAccounts() ([]Account, error)
    GetAccountsByName(name string) ([]Account, error)
    GetAccountsWithBalanceGreaterThan(balance Money)
                                        ([]Account, error)
}
```

A specification-based approach can be used if many queries are necessary, and the hard-coded queries could generate a cluttered interface. It requires more code to describe each possible specification but allows for a very high level of flexibility and extensibility of the repository. An example of a repository that allows many different filter criteria to retrieve objects is:

```
type AccountRepository interface {
    Find(spec Specification) ([]Account, error)
}

type Specification interface {
    IsSatisfiedBy(account Account) bool
}

type NameSpecification struct {
    Name string
}

func (n NameSpecification) IsSatisfiedBy(account Account)
                                                        bool {
    return account.Name == n.Name
}

type BalanceSpecification struct {
    MinBalance Money
}

func (b BalanceSpecification) IsSatisfiedBy(account Account)
                                                        bool {
    return account.Balance >= b.MinBalance
}
```

5.3 Working with domain events

The brainstorming workshops presented in Chapter 3 allow the team to discover the events characteristic of the problem domain. Event storming models the domain flow of information as a series of discrete events, that is, changes in the domain state.

One of the main advantages of modeling domain events is the ability to decouple system components. By modeling domain events, the components of the system can react to changes in the domain state without having to be directly connected to the source of the change. It leads to a more flexible and scalable system, as new components can be added without changing existing components.

Domain events can be used as a source of reactive data as the principles of Reactive Programming [24]. In a reactive system, changes in the system's state can be stated as a fact to the rest of the system. Other parts of the system can then monitor these changes, allowing the system to respond to those events dynamically and efficiently.

When a purchase is made in a specific account, suppose an event is published to other parts of the system with this information. In that case, notification systems can send the information via e-mail, SMS, or push notifications to the account holder. Moreover, credit analysis, anti-fraud, or recommendation services can be triggered depending on the value of the transaction. Any system that needs to react to this purchase event must only register itself as a listener to this event, decoupling the system's modules.

The observer pattern can be used to implement this schema. It can be an in-memory publish subscriber system or a messaging system with Kafka, RabbitMQ, or AWS SQS, depending on the scale of the application. Different bounded-context can communicate and exchange information in the form of immutable units (domain events). Domain events are essentially value objects.

Adopting domain events as stating facts about the system allows using two interesting techniques for state storage: event sourcing and CQRS.

Event sourcing

Event sourcing is a software design pattern that involves storing a log of changes made to an application's data rather than storing the current state of the data itself [21]. This log of changes, known as an event store, can be used to reconstruct the data's current state at any given time.

This pattern is particularly efficient for data-intensive applications because writing single units of immutable domain events require minimal locking mechanisms. The system always updates the state by appending information to the database, never updating or deleting it, so many concurrency problems can be avoided.

There are several technical benefits of using event sourcing in application design:

- Auditability: Provides a clear, chronological record of all changes made to an application's data. This can be useful for audit purposes, as it allows for reconstructing the exact sequence of events that led to a particular data state. Helping especially with debugging.

- Data recovery: If data is lost or corrupted, it may be possible to recover the data by replaying the event stream up to the point of the loss or corruption. It facilitates rollback commands.
- Regulatory compliance: Regulatory requirements such as the General Data Protection Regulation (GDPR) or the *Lei Geral de Proteção de Dados* (LGPD) may require companies to maintain a record of all changes made to personal data, and event sourcing can help to meet these requirements.
- Decoupling: Event sourcing can help decouple the various components of an application, as the event store serves as a single source of truth for the data rather than requiring each component to maintain its copy. It can make adding or modifying components easier without affecting the rest of the application.

CQRS

It's common to combine the event sourcing pattern with Command and Query Responsibility Segregation (CQRS) to avoid replaying events to achieve the system's current state. The CQRS pattern separates commands, which change data, and queries that only retrieve information about the system's current state.

The principle behind this pattern is that applications usually have different writing and reading necessities, with variable loads for them. The technologies used to store events and to read data from the system can be completely different. Denormalization and heavy indexing can be applied to the reading system, which will not directly impact the writing operations.

Separating the typical model of a unique database into two also allows scaling those different responsibilities to be better optimized.

There are essential trade-offs when considering Event Sourcing and CQRS. The CQRS implies that a queue of events separates the command and querying databases. The direct consequence of this is that the data consistency of the system will now be only eventual. Changes written to the database will only be visible by queries after milliseconds or even seconds, depending on the system's load. In eventual consistency, it's possible to guarantee that the system will always converge to an agreement about its state, but there is always some inconsistency window of time.

The complexity of storage is increased, so it must be important to assess when the benefits will compensate for this infrastructure complexity. Those approaches are especially relevant for high-load and real-time systems. CQRS and event sourcing

should be only used on specific bounded contexts and not to store information about the system as a whole. Each bounded context needs its design considerations. Figure 5.2 exemplifies a CQRS and Event Sourcing architecture.

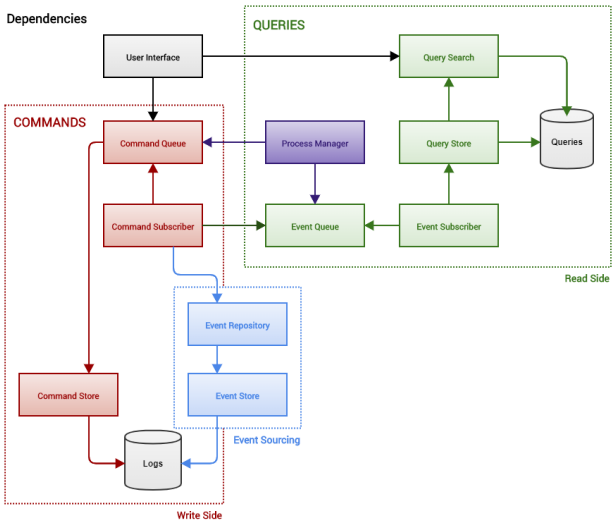

Figure 5.2: Example of a CQRS and Event Sourcing architecture [25]

Chapter 6

CONCLUSION

The architecture of enterprise applications is an always-evolving topic. New demands and necessities derived from now-unknown business aspects will continually provide new techniques and approaches to software architecture. Nevertheless, the majority of topics discussed in previous chapters have almost two decades of establishment and have been applied to various enterprise applications needs.

Two fundamental aspects of modeling those applications are:

- The necessity to establish a ubiquitous language that binds real-world business requirements to the application as a domain model. This domain needs to be found in the collaboration between the technical staff and the business stakeholders (domain experts). The model is aimed to correctly distillate the essential concepts from the problem domain and serves as the basis and source of truth to model the software system. From it, the high-level modules of the system will be derived, its bounded contexts, and the description of the main processes as a series of events.
- The isolation of the domain layer from other application concerns is the central point of a layered architecture. The domain must be free from infrastructure and representation concerns, focusing only on enforcing business rules.

Moreover, once a domain model is established, abstractions, such as entities, value objects, repositories, and factories, can help preserve the cleanness of the domain layer. Event sourcing and CQRS can be used in high-scale applications to perform better and decouple different bounded contexts.

Further improvements to the system can be made by adopting other high-level concepts derived from DDD, such as domain-centric architectures. Those architectures include Alistair Cockburn's Hexagonal Architecture [26], and Robert R. Martin's Clean Architecture [27].

BIBLIOGRAPHY

[1] N. Forsgren, J. Humble, and G. Kim, *Accelerate: The Science Behind DevOps : Building and Scaling High Performing Technology Organizations* (G - Reference,Information and Interdisciplinary Subjects Series). IT Revolution, 2018, ISBN: 9781942788331. [Online]. Available: https://books.google.com.br/books?id=85XHAQAACAAJ.

[2] Evans, *Domain-Driven Design: Tacking Complexity In the Heart of Software*. USA: Addison-Wesley Longman Publishing Co., Inc., 2003, ISBN: 0321125215.

[3] M. Fowler, D. Rice, M. Foemmel, E. Hieatt, R. Mee, and R. Stafford, *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002.

[4] D. J. Snowden and M. E. Boone, *A leader's framework for decision making*, Dec. 2007. [Online]. Available: https://hbr.org/2007/11/a-leaders-framework-for-decision-making.

[5] E. Codd, "Relational database: A practical foundation for productivity," in *Readings in Artificial Intelligence and Databases*, J. Mylopolous and M. Brodie, Eds., San Francisco (CA): Morgan Kaufmann, 1989, pp. 60–68, ISBN: 978-0-934613-53-8. DOI: https://doi.org/10.1016/B978-0-934613-53-8.50009-1. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780934613538500091.

[6] P. C. KANELLAKIS, "Chapter 17 - elements of relational database theory," in *Formal Models and Semantics*, ser. Handbook of Theoretical Computer Science, J. VAN LEEUWEN, Ed., Amsterdam: Elsevier, 1990, pp. 1073–1156, ISBN: 978-0-444-88074-1. DOI: https://doi.org/10.1016/B978-0-444-88074-1.50022-6. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780444880741500226.

[7] P. Douglas and M. Douglas, *Purity and Danger: An Analysis of Concepts of Pollution and Taboo*. Taylor & Francis, 2013, ISBN: 9781136489273. [Online]. Available: https://books.google.com.br/books?id=CjFHQGiXsVcC.

[8] E. Evans, *Domain-Driven Design Reference: Definitions and Pattern Summaries*. Dog Ear Publishing, 2014, ISBN: 9781457501197. [Online]. Available: https://books.google.com.br/books?id=ccRsBgAAQBAJ.

[9] K. Beck, M. Beedle, A. van Bennekum, *et al.*, *Manifesto for agile software development*, 2001. [Online]. Available: http://www.agilemanifesto.org/.

[10] R. C. Martin, *Agile software development : principles, patterns, and practices*. Prentice Hall, 2003, ISBN: 9780132760584. DOI: 10.1002/pfi.21408. [Online]. Available: http://dx.doi.org/10.1002/pfi.21408.

[11] M. Kersten, *Project to Product: How to Survive and Thrive in the Age of Digital Disruption with the Flow Framework* (G - Reference,Information and Interdisciplinary Subjects Series). IT Revolution Press, 2018, ISBN: 9781942788393. [Online]. Available: https://books.google.com.br/books?id=G1aQtAEACAAJ.

[12] M. Cohn, *User Stories Applied: For Agile Software Development*. USA: Addison Wesley Longman Publishing Co., Inc., 2004, ISBN: 0321205685.

[13] R. Jeffries, A. Anderson, and C. Hendrickson, *Extreme Programming Installed* (XP series). Addison-Wesley, 2001, ISBN: 9780201708424. [Online]. Available: https://books.google.com.br/books?id=l4z030WkdIsC.

[14] J. Patton, *User Story Mapping*. Beijing: O'Reilly, 2014, ISBN: 978-1-4919-0490-9. [Online]. Available: https://www.safaribooksonline.com/library/view/user-story-mapping/9781491904893/.

[15] G. Kim, *The Unicorn Project: A Novel about Developers, Digital Disruption, and Thriving in the Age of Data*. IT Revolution, 2019, ISBN: 9781942788768. [Online]. Available: https://books.google.com.br/books?id=o0z1tgEACAAJ.

[16] A. Brandolini, *Introducting EventStorming: An act of Deliberate Collective Learning*. https://leanpub.com/, 2022. [Online]. Available: https://leanpub.com/introducing_eventstorming.

[17] R. Laszczak, *Software dark ages — threedots.tech*, https://threedots.tech/post/software-dark-ages/, [Accessed 12-Out-2022], June 2021.

[18] E. W. Dijkstra, "On the role of scientific thought," in *Selected Writings on Computing: A personal Perspective*. New York, NY: Springer New York, 1982, pp. 60–66, ISBN: 978-1-4612-5695-3. DOI: 10.1007/978-1-4612-5695-3_12. [Online]. Available: https://doi.org/10.1007/978-1-4612-5695-3_12.

[19] S. Black, "Computing ripple effect for software maintenance," *Journal of Software Maintenance*, vol. 13, no. 4, p. 263, Sep. 2001, ISSN: 1040-550X.

[20] M. Fowler, *Bliki: Anemicdomainmodel*, Nov. 2003. [Online]. Available: https://martinfowler.com/bliki/AnemicDomainModel.html.

[21] V. Vernon, *Implementing Domain-Driven Design*, 1st. Addison-Wesley Professional, 2013, ISBN: 0321834577.

[22] *The patterns - domain-driven design*. [Online]. Available: https://doc.rust-cqrs.org/intro.html.

[23] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed. Addison-Wesley Professional, 1994, ISBN: 0201633612. [Online]. Available: http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612/ref=ntt_at_ep_dpi_1.

[24] T. Nurkiewicz and B. Christensen, *Reactive Programming with RxJava: Creating Asynchronous, Event-Based Applications*. O'Reilly Media, 2016, ISBN: 9781491931608. [Online]. Available: https://books.google.com.br/books?id=gYY1DQAAQBAJ.

[25] D. Miller, *A fast and lightweight solution for cqrs and event sourcing*, Apr. 2020. [Online]. Available: https://www.codeproject.com/Articles/5264244/A-Fast-and-Lightweight-Solution-for-CQRS-and-Event.

[26] A. O. Cockburn, *Hexagonal architecture*, Oct. 2021. [Online]. Available: https://alistair.cockburn.us/hexagonal-architecture/.

[27] R. C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*, 1st. USA: Prentice Hall Press, 2017, ISBN: 0134494164.