Hands-On Domain-Driven Design with .NET Core

Tackling complexity in the heart of software by putting DDD principles into practice

Alexey Zimarev

Packt
www.packt.com

Hands-On Domain-Driven Design with .NET Core

Tackling complexity in the heart of software by putting DDD principles into practice

Alexey Zimarev

Packt>

BIRMINGHAM - MUMBAI

Hands-On Domain-Driven Design with .NET Core

Commissioning Editor: Richa Tripathi
Acquisition Editor: Chaitanya Nair
Content Development Editor: Rohit Singh
Technical Editor: Gaurav Gala
Copy Editor: Safis Editing
Project Coordinator: Vaidehi Sawant
Proofreader: Safis Editing
Indexer: Tejal Daruwale Soni
Graphics: Alishon Mendonsa
Production Coordinator: Nilesh Mohite

First published: April 2019

Production reference: 1300419

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78883-409-4

www.packtpub.com

To my wonderful family: my wife Olga, our sons Denis and Miklail, and our Siberian husky Taiga. Thank you for your patience and for giving me time and space to complete this big work. Without your support, this book would have never been published.

To my friend and colleague Sérgio Silveira Vaqueiro. We learned a lot from each other and I am very happy to have an opportunity to work with you. Nearly all the code in this book is based on the code we've prepared for the Hands-on Event Sourcing workshop that we deliver together.

mapt.io

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

Packt.com

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the author

Alexey Zimarev is a software architect with a present focus on domain models, **Domain-Driven Design** (**DDD**), event sourcing, message-driven systems and microservices, coaching, and mentoring. Alexey is also a contributor to several open source projects, such as RestSharp and MassTransit, and is the organizer of the DDD Norway meetup.

About the reviewers

Marcin Budny is a software developer with over a decade of experience in designing and building systems. He specializes in getting to the bottom of things and finding the worst-case scenarios. Mostly focused on the .NET ecosystem, he likes to venture into other territories to steal good ideas.

Marcin works mostly with Poland-based companies that cooperate with partners around the globe. That has resulted in a broad spectrum of challenges that he has had to face, which he has now turned into topics to share with his local software development community.

In this book, Marcin debuts as a reviewer.

Nick Tune is the coauthor of two books, P*atterns, Principles and Practices of Domain-Driven Design* (Wrox) and *Designing Autonomous Teams and Services* (O'Reilly), and frequently writes about technical leadership at NT.Coding().

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface

This book will help you solve complex business problems by understanding users better, finding the right problem to solve, and building lean, event-driven systems to give your customers what they really want. You will be taken through the fundamentals of **Domain-Driven Design** (**DDD**) principles and how it can be applied using modern tools such as EventStorming, Event Sourcing, and CQRS. Through this book, you will learn how DDD applies directly to various architectural styles, such as REST, reactive systems, and microservices. You will empower teams to work flexibly with improved services and decoupled interactions.

Who this book is for

This book is for .NET developers who have an intermediate level understanding of C#, and for those who seek to deliver value, not just write code. An intermediate level of competence in JavaScript will be helpful for the UI chapters.

What this book covers

`Chapter 1`, *Why Domain-Driven Design?*, covers the concepts of problem and solution spaces, requirements specifications, complexity, knowledge, and ignorance. These topics have a significant impact on how and what we deliver.

`Chapter 2`, *Language and Context*, deep dives into the importance of language and explains Ubiquitous Language.

`Chapter 3`, *EventStorming*, explores one of the most popular techniques for domain modeling and goes through some practical tips on how to organize useful workshops between domain experts and developers.

`Chapter 4`, *Designing the Model*, goes deeper into the modeling process, with more of a focus on artifacts that can help us to start writing code and deliver initial prototypes as soon as possible.

`Chapter 5`, *Implementing the Model*, forms the basis for our domain model implemented in code. We will go through different styles of performing the behavior in domain entities and also write some tests.

`Chapter 6`, *Acting with Commands*, shows how to implement commands, and how commands are the glue between our domain model and the world outside it. We will learn how to make our model useful by letting people interact with it.

`Chapter 7`, *Consistency Boundary*, takes a closer look at entity persistence, and its scope will be our focus. We will learn what types of consistency we need to deal with and how important it is to understand consistency boundaries.

`Chapter 8`, *Aggregate Persistence*, takes a deep dive into the topic of aggregate persistence. We will find a way to store our domain objects in a database and see our application working for the first time.

`Chapter 9`, *CQRS - The Read Side*, covers the read side of CQRS and explains what the read models are. You will learn how to use Ubiquitous Language for queries and see how to implement CQRS with one database.

`Chapter 10`, *Event Sourcing*, shows how events can be used to persist the state of an object, instead of using traditional persistence mechanisms. We will cover the concept of event streams and see how streams relate to aggregates. We will use the Event Store to persist our aggregates in streams and load them back.

`Chapter 11`, *Projections and Queries*, takes you through the challenges of querying the Event Sourced system and solving these challenges by using separate read models and projections.

`Chapter 12`, *Bounded Context*, makes you familiar with the concept of Bounded Contexts. We will identify contexts in our project and separate the system into pieces. We will also learn about the Context Map, which shows the landscape of Bounded Contexts for the entire system and their relationships.

`Chapter 13`, *Splitting the System*, gives practical advice about identifying Bounded Contexts and implementing more than one context in the sample application. This chapter is available as an online chapter at: `https://www.packtpub.com/sites/default/files/downloads/Splitting_the_System.pdf`.

To get the most out of this book

In order to follow the instructions in this book, you need to have intermediate-level understanding of C#. Other requirements are mentioned at the relevant instances in the respective chapters.

Diagrams for this book are created using Miro, an online collaboration tool, and `draw.io`, the free online service for creating diagrams and wireframes. The line style and the font are used intentionally, to embrace the temporal nature and usefulness of all models and diagrams

Download the example code files

You can download the example code files for this book from your account at `www.packtpub.com`. If you purchased this book elsewhere, you can visit `www.packtpub.com/support` and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at `www.packtpub.com`.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Hands-On-Domain-Driven-Design-with-.NET-Core`. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://www.packtpub.com/sites/default/files/downloads/9781788834094_ColorImages.pdf`

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Mount the downloaded `WebStorm-10*.dmg` disk image file as another disk in your system."

A block of code is set as follows:

```
html, body, #map {
 height: 100%;
 margin: 0;
 padding: 0
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
[default]
exten => s,1,Dial(Zap/1|30)
exten => s,2,Voicemail(u100)
exten => s,102,Voicemail(b100)
exten => i,1,Voicemail(s0)
```

Any command-line input or output is written as follows:

```
$ mkdir css
$ cd css
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Select **System info** from the **Administration** panel."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email `feedback@packtpub.com` and mention the book title in the subject of your message. If you have questions about any aspect of this book, please email us at `questions@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/submit-errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packtpub.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packtpub.com`.

1
Why Domain-Driven Design?

The software industry appeared back in the early 1960s and has been growing ever since. There have been predictions that one day all software will be written and software developers will no longer be needed, but this prophecy has never become reality, and the growing army of software engineers is working hard to satisfy the continually increasing demand.

However, from the very early days of the industry, the number of projects that were delivered very late and massively over budget, plus the number of failed projects, was overwhelming. The 2015 CHAOS report by the Standish Group (`https://www.projectsmart.co.uk/white-papers/chaos-report.pdf`) suggests that from 2011 to 2015, the percentage of successful IT projects remained unchanged at a level of just 22%. Over 19% of projects failed, and the rest experienced challenges. Although the report might set somewhat controversial expectations for project success, it still paints a picture that is familiar to many. These numbers are astonishing. Over four decades, a lot of methods have been developed and advertised as silver bullets for software project management, but there has been little or no change in the number of successful projects.

One of the critical factors that define the success of any IT project is understanding the problem that the system is supposed to solve. We are all very familiar with systems that do not solve the problems they claim to answer or do it very inefficiently. Both the SCRUM and XP software development methodologies embrace interacting with users and understanding their problems.

The term **Domain-Driven Design** (**DDD**) was coined by Eric Evans in his now-iconic book *Domain-Driven Design: Tackling Complexity in the Heart of Software* published by Addison-Wesley back in 2004. More than a decade after the book was published, interest in the practices and principles described in the book started to grow exponentially. Many factors influenced this growth in popularity, but the most important one is that DDD explains how people from the software industry can build an understanding of their users' needs and create software systems that solve the problem and make an impact.

In this chapter, we will discuss how understanding the business domain, building domain knowledge, and distinguishing essential complexity from accidental complexity can help in creating software that matters.

The objective of this chapter is to explore the following topics:

- Problem space versus solution space
- What went wrong with requirements
- Understanding complexity
- The role of knowledge in software development

Understanding the problem

We rarely write software to just write some code. Of course, we can create a pet project for fun and to learn new technologies, but professionally we build software to help other people to do their work better, faster, and more efficiently. Otherwise, there is no point in writing any software in the first place. It means that we need to have a *problem* that we intend to solve. Cognitive psychology defines the issue as a restriction between the current state and the desired state.

Problem space and solution space

In their book *Human Problem Solving*, Allen Newell and Herbert Simon outlined the problem space theory. The theory states that humans solve problems by searching for a solution in the *problem space*. The problem space describes the initial and desired states, as well as possible intermediate states. It can also contain specific constraints and rules that define the context of the problem. In the software industry, people operating in the problem space are usually customers and users.

Each real problem demands a solution, and if we search properly in the problem space, we can outline which steps we need to take to move from the initial state to the desired state. This outline and all the details about the solution form a *solution space*.

The classic story of problem and solution spaces, which get completely detached from each other during the implementation, is the story of writing in space. The story goes that in the 1960s, space-exploring nations realized that the usual ballpoint pens wouldn't work in space due to the lack of gravity. NASA then spent a million dollars to develop a pen that would work in space, and the Soviets decided to use the good old pencil, which costs almost nothing.

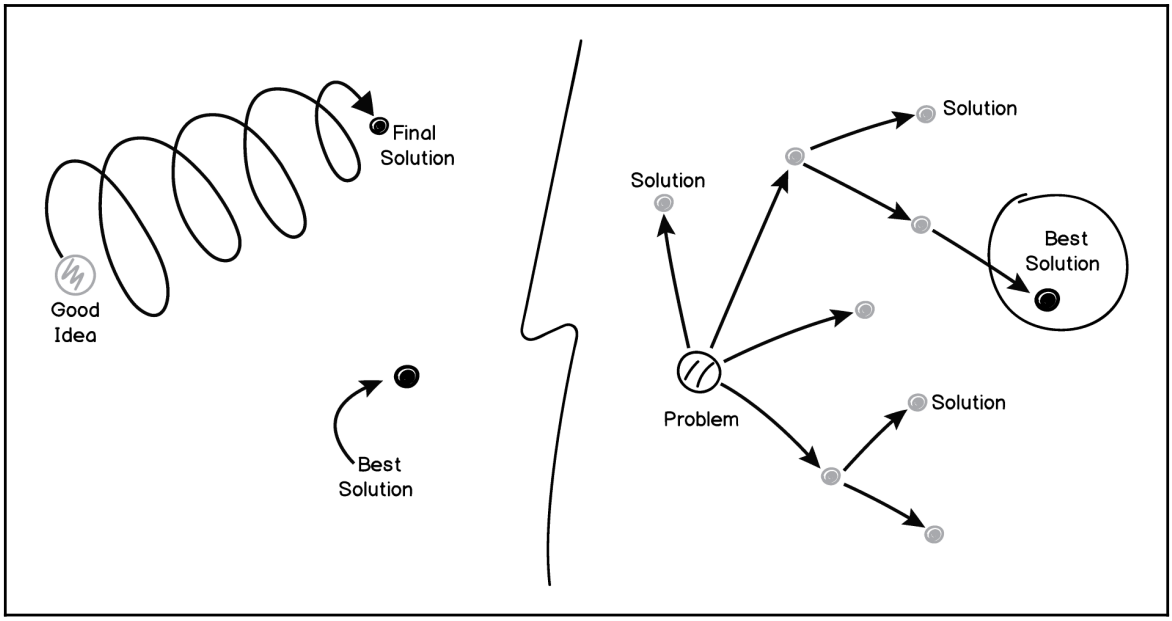
This story is so compelling that it is still circulating, and was even used in the TV show *The West Wing*, with Martin Sheen playing the US president. It is so easy to believe, not only because we are used to wasteful spending by government-funded bodies, but mostly because we have seen so many examples of inefficiency and misinterpretation of real-world issues, adding enormous unnecessary complexity to their proposed solutions and solving problems that don't exist.

This story is a myth. NASA also tried using pencils but decided to get rid of them due to the production of microdust, tips breaking, and the potential flammability of wooden pencils. A private company called Fisher developed what is now known as a **space pen**. Later, NASA tested the pen and decided to use it. The company also got an order from the Soviet Union, and pens were sold across the world. The price was the same for everyone, $2.39 per pen.

Here you can see the other part of the problem space/solution space issue. Although the problem itself appeared to be simple, additional constraints, which we could also call **non-functional requirements**, or, to be more precise, operational requirements, made it more complicated than it looked at first glance.

Jumping to a solution is very easy, and since most of us have a rather rich experience of solving everyday problems, we can find solutions for many issues almost immediately. However, as Bart Barthelemy and Candace Dalmagne-Rouge suggest in their article *When You're Innovating, Resist Looking for Solutions* (2013, Harvard Business Review `https://hbr.org/2013/09/when-youre-innovating-resist-l`), thinking about solutions prevents our brain from thinking about the problem. Instead, we start going deeper into the solution that first came to our mind, adding more levels of detail and making it the most ideal solution for a given problem.

There's one more aspect to consider when searching for a solution to a given problem. There is a danger of fixating all your attention on one particular solution, which might not be the best one at all but it was the first to come to mind, based on your previous experiences, your current understanding of the problem, and other factors:



Refinement versus exploration

The exploratory approach to find and choose solutions involves quite a lot of work to try out a few different things, instead of concentrating on the iterative improvement of the original *good idea*. However, the answer that is found during this type of exploration will most probably be much more precise and valuable. We will discuss fixation on the first possible solution later in this chapter.

What went wrong with requirements

We are all familiar with the idea of requirements for software. Developers rarely have direct contact with whoever wants to solve a problem. Usually, some dedicated people, such as requirements analysts, business analysts, or product managers, talk to customers and generalize the outcomes of these conversations in the form of functional requirements.

Requirements can have different forms, from large documents called a **requirements specification** to more **agile** means such as user stories. Let's have a look at this example:

> *"Every day, the system shall generate, for each hotel, a list of guests expected to check in and check out on that day."*

As you can see, this statement only describes the solution. We cannot possibly know what the user is doing and what problem our system will be solving. Additional requirements might be specified, further refining the solution, but a description of the problem is never included in functional requirements.

In contrast, with user stories, we have more insight into what our user wants. Let's review this real-life user story: *"As a warehouse manager, I need to be able to print a stock-level report so that I can order items when they are out of stock."* In this case, we have an insight into what our user wants. However, this user story already dictates what the developers need to do. It is describing the *solution*. The real problem is probably that the customer needs a more efficient procurement process, so they never run out of stock. Or, they need an advanced purchase forecasting system, so they can improve throughput without stockpiling additional inventory in their warehouse.

We should not think that the requirements are a waste of time. There are many excellent analysts out there who produce high-quality requirements specifications. However, it is vital to understand that these requirements almost always represent the understanding of the actual problem from the point of view of the person who wrote them. A misconception that spending more and more time and money on writing higher-quality requirements prevails in the industry.

However, lean and agile methodologies embrace more direct communication between developers and end users. Understanding the problem by all stakeholders, from end users to developers and testers, finding solutions together, eliminating assumptions, building prototypes for end users to evaluate—all these things are being adopted by successful teams, and as we will see later in the book, they are also closely related to DDD.

Dealing with complexity

Before writing about complexity, I tried to find some fancy, striking definition of the word itself, but it appeared to be a complex task on its own. Merriam-Webster defines the word **complexity** as the quality or state of being complex and this definition is rather obvious and might even sound silly. Therefore, we need to dive a bit deeper into this subject and understand more about complexity.

In software, the idea of complexity is not much different from the real world. Most software is written to deal with real-world problems. Those problems might sound simple but be intrinsically complex, or even wicked. Without a doubt, the problem space complexity will be reflected in the software that tries to solve such a problem. Realizing what kind of complexity we are dealing with when creating software thus becomes very important.

Types of complexity

In 1986, the Turing Award winner Fred Brooks wrote a paper called *No Silver Bullet – Essence and Accident in Software Engineering* in which he made a distinction between two types of complexity—essential and accidental complexity. Essential complexity comes from the domain, from the problem itself, and it cannot be removed without decreasing the scope of the problem. In contrast, accidental complexity is brought to the solution by the solution itself—this could be a framework, a database, or some other infrastructure, with different kinds of optimization and integration.

Brooks argued that the level of accidental complexity decreased substantially when the software industry became more mature. High-level programming languages and efficient tooling give programmers more time to work on business problems. However, as we can see today, more than 30 years later, the industry still struggles to fight accidental complexity. Indeed, we have power tools in our hands, but most of those tools come with the cost of spending the time to learn the tool itself. New JavaScript frameworks appear every year and each of them is different, so before writing anything useful, we need to learn how to be efficient when using the framework of choice. I wrote some JavaScript code many years ago and I saw Angular as a blessing until I realized that I spend more time fighting with it than writing anything meaningful. Or take an example of containers that promised us to bring an easy way to host our applications in isolation, without all that hassle with physical or virtual machines. But then we needed an orchestrator, and we got quite a few, spent time learning to work with them until we got Kubernetes to rule them all and now we spend more time writing YAML files than actual code. We will discuss some possible reasons for this phenomenon in the next section.

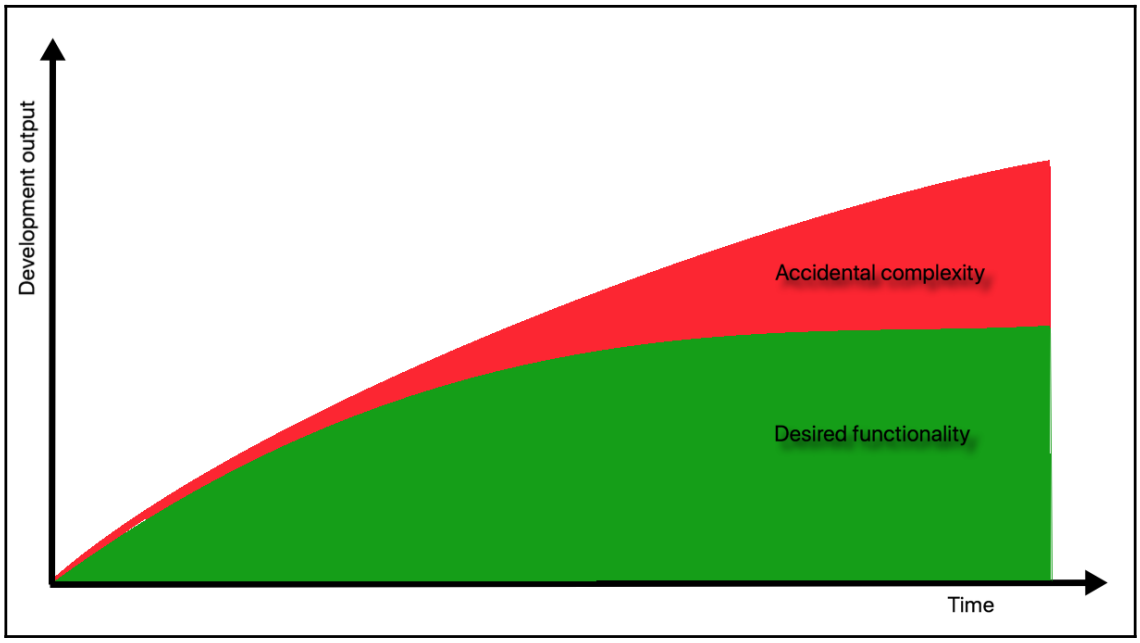
You probably noticed that essential complexity has a strong relation to the problem space, and accidental complexity leans towards the solution space. However, we often seem to get problem statements that are more complex than the problems themselves. Usually, this happens due to problems being mixed with solutions, as we discussed earlier, or due to a lack of understanding.

Gojko Adžić, a software delivery consultant and the author of several influential books, such as *Specification by Example* and *Impact Mapping*, gives this example in his workshop:

> *"A software-as-a-service company got a feature request to provide a particular report in real time, which previously was executed once a month on schedule. After a few months of development, salespeople tried to get an estimated delivery date. The development department then reported that the feature would take at least six more months to deliver and the total cost would be around £1 million. It was because the data source for this report is in a transactional database and running it in real time would mean significant performance degradation, so additional measures such as data replication, geographical distribution, and sharding were required.*
>
> *The company then decided to analyze the actual need that the customer who requested this feature had. It turned out that the customer wanted to perform the same operations as they were doing before, but instead of doing it monthly, they wanted it weekly. When asked about the desired outcome of the whole feature, the customer then said that running the same report batched once a week would solve the problem. Rescheduling the database job was by far an easier operation that redesigning the whole system, while the impact for the end customer was the same."*

This example clearly shows that not understanding the problem can lead to severe consequences. We as developers love principles like DRY. We seek abstraction that will make our code more elegant and concise. However, often that might be entirely unnecessary. Sometimes we fall to the trap of using some tool or framework that promises to solve all issues in the world, easily. Again, that never comes without a cost. As a .NET developer, I can clearly see this when I look at the current obsession with dependency injection among the community.

True enough, Microsoft finally made a DI container that makes sense, but when I see it being used in a small console app just to initialize the logger, I get upset. Sometimes, more code is being written just to satisfy the tool, the framework, the environment, than the code that delivers the actual value. What seemed to be the essential complexity in this example turned out to be a waste:



Complexity growth over time

The preceding graph shows that with the ever-growing complexity of the system, the essential complexity is being pushed down and the accidental complexity takes over. You might have doubts about the fact that accidental complexity keeps growing over time when the desired functionality almost flatters out. How could this happen, definitely we can't spend time only creating more accidental complexity? When systems become more prominent, a lot of effort is required to make the system work as a whole and to manage large data models, which large systems tend to have. Supportive code grows and a lot of effort is being spent to keep the system running. We bring cache, optimize queries, split and merge databases, the list goes on. In the end, we might actually decide to reduce the scope of the desired functionality just to keep the system running without too many glitches.

DDD helps you focus on solving complex domain problems and concentrates on the essential complexity. For sure, dealing with a new fancy front-end tool or use a cloud document database is fun. But without understanding what problem are we trying to solve, it all might be just waste. It is much more valuable to any business to get something useful first and try it out than getting a perfect piece of state-of-the-art software that misses the point entirely. To do this, DDD offers several useful techniques for managing complexity by splitting the system into smaller parts and making these parts focus on solving a set of related problems. These techniques are described later in this book.

The rule of thumb when dealing with complexity is—embrace essential, or as we might call it, domain complexity, and eliminate or decrease the accidental complexity. Your goal as a developer is not to create too much accidental complexity. Hence, very often, accidental complexity is caused by over-engineering.

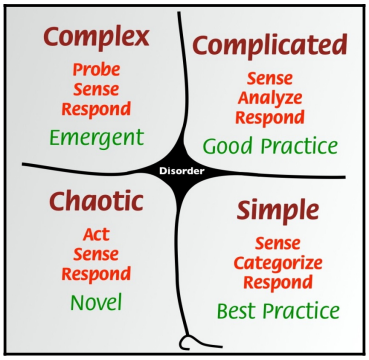
Categorizing complexity

When dealing with problems, we don't always know whether these problems are complex. And if they are complex, how complex? Is there a tool for measuring complexity? If there is, it would be beneficial to measure, or at least categorize, the problem's complexity before starting to solve it. Such measurement would help to regulate the solution's complexity as well, since complex problems also demand a complex solution, with rare exceptions to this rule. If you disagree, we will be getting deeper into this topic in the following section.

In 2007, Dave Snowden and Mary Boone published a paper called *A Leader's Framework for Decision Making* in Harvard Business Review, 2007. This paper won the **Outstanding Practitioner-Oriented Publication in OB** award from the Academy of Management's Organizational Behavior division. What is so unique about it, and which framework does it describe?

The framework is **Cynefin**. This word is Welsh for something like *habitat,* accustomed, familiar. Snowden started to work on it back in 1999 when he worked at IBM. The work was so valuable that IBM established the Cynefin Center for Organizational Complexity, and Dave Snowden was its founder and director.

Cynefin divides all problems into five categories or complexity domains. By describing the properties of problems that fall into each domain, it provides a *sense of place* for any given problem. After the problem is categorized into one of the domains, Cynefin then also offers some practical approaches to deal with this kind of problem:



Cynefin framework, image by Dave Snowden

These five realms have specific characteristics, and the framework provides attributes for both, identifying to which domain your problem belongs, and how the problem needs to be addressed.

The first domain is Simple, or Obvious. Here, you have problems that can be described as *known knowns*, where best practices and an established set of rules are available, and there is a direct link between a cause and a consequence. The sequence of actions for this domain is *sense-categorize-response*. Establish facts (sense), identify processes and rules (categorize), and execute them (response).

Snowden, however, warns about the tendency for people to wrongly classify problems as *simple*. He identifies three cases for this:

- **Oversimplification**: This correlates with some of the cognitive biases described in the following section.
- **Entrained thinking**: When people blindly use the skills and experiences they have obtained in the past and therefore become blinded to new ways of thinking.
- **Complacency**: When things go well, people tend to relax and overestimate their ability to react to the changing world. The danger of this case is that when a problem is classified as simple, it can quickly escalate to the chaotic domain due to a failure to adequately assess the risks. Notice the *shortcut* from Simple to Chaotic domain at the bottom of the diagram, which is often being missed by those who study the framework.

For this book, it is important to remember two main things:

- If you identify the problem as obvious, you probably don't want to set up a complex solution and perhaps would even consider buying some off-the-shelf software to solve the problem, if any software is required at all.
- Beware, however, of wrongly classifying more complex problems in this domain to avoid applying the wrong best practices instead of doing more thorough exploration and research.

The second domain is Complicated. Here, you find problems that require expertise and skill to find the relation between cause and effect, since there is no single answer to these problems. These are *known unknowns*. The sequence of actions in this realm is *sense-analyze-respond*. As we can see, *analyze* replaces *categorize* because there is no clear categorization of facts in this domain. Proper analysis needs to be done to identify which good practice to apply. Categorization can be done here too, but you need to go through more choices and analyze the consequences as well. That is where previous experience is necessary. Engineering problems are typically in this category, where a clearly understood problem requires a sophisticated technical solution.

In this realm, assigning qualified people to do some design up front and then perform the implementation makes perfect sense. When a thorough analysis is done, the risk of implementation failure is low. Here, it makes sense to apply DDD patterns for both strategic and tactical design, and to the implementation, but you could probably avoid more advanced exploratory techniques such as EventStorming. Also, you might spend less time on knowledge crunching, if the problem is thoroughly understood.

Complex is the third complexity domain in Cynefin. Here, we encounter something that no one has done before. Making even a rough estimate is impossible. It is hard or impossible to predict the reaction to our action, and we can only find out about the impact that we have made in retrospect. The sequence of actions in this domain is *probe-sense-respond*. There are no right answers here and no practices to rely upon. Previous experience won't help either. These are *unknown unknowns*, and this is the place where all innovation happens. Here, we find our core domain, the concept, which we will get to later in the book.

The course of action for the complex realm is led by experiments and prototypes. There is very little sense in creating a big design up front since we have no idea how it will work and how the world will react to what we are doing. Work here needs to be done in small iterations with continuous and intensive feedback.

Advanced modeling and implementation techniques that are lean enough to respond to changes quickly are the perfect fit in this context. In particular, modeling using EventStorming and implementation using event-sourcing are very much at home in the complex domain. A thorough strategic design is necessary, but some tactical patterns of DDD can be safely ignored when doing spikes and prototypes, to save time. However, again, event-sourcing could be your best friend. Both EventStorming and event-sourcing are described later in the book.

The fourth domain is Chaotic. This is where hellfire burns and the Earth spins faster than it should. No one wants to be here. Appropriate actions here are *act-sense-respond*, since there is no time for spikes. It is probably not the best place for DDD since there is no time or budget for any sort of design available at this stage.

Disorder is the fifth and final realm, right in the middle. The reason for it is that when being at this stage, it is unclear which complexity context applies to the situation. The only way out from disorder is to try breaking the problem into smaller pieces that can be then categorized into those four complexity contexts and then deal with them accordingly.

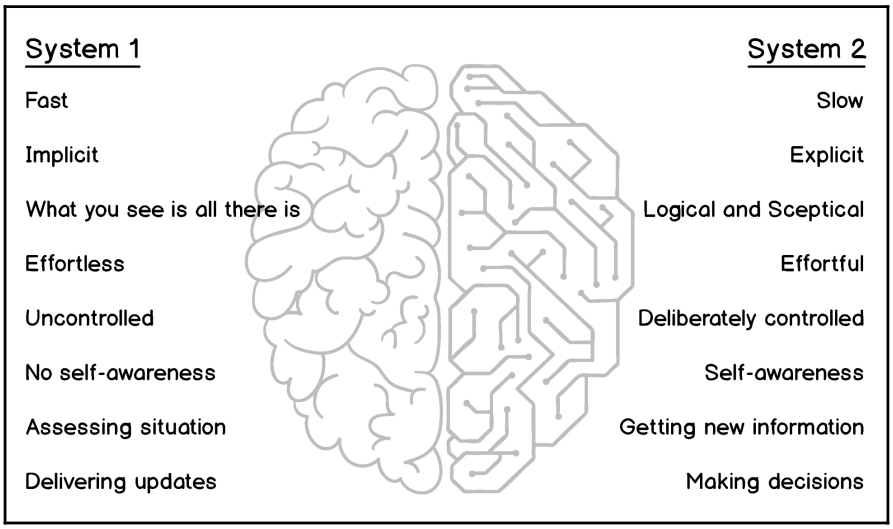
This is only a brief overview of the categorization of complexity. There is more to it, and I hope your mind gets curious to see examples, videos, and articles about the topic. That was the exact reason for me to bring it in, so please feel free to stop reading now and explore the complexity topic some more. For this book the most important outcome is that DDD can be applied almost everywhere, but it is of virtually no use in obvious and chaotic domains. Using EventStorming as a design technique in complex systems would be useful for both complicated and complex domains, along with event-sourcing, which suits complex domains best.

Decision making and biases

The human brain processes a tremendous amount of information every single second. We do many things on autopilot, driven by instinct and habit. Most of our daily routines are like this. Other areas of brain activity are thinking, learning, and decision-making. Such actions are performed significantly more slowly and require much more power than the automatic operations.

Dual process theory in psychology suggests that these types of brain activity are indeed entirely different and there are two different processes for two kinds of thinking. One is the implicit, automatic, unconscious process, and the other one is an explicit conscious process. Unconscious processes are formed over a long time and are also very hard to change because changing such a process would require developing a new habit, and this is not an easy task. Conscious processes, however, can be altered through logical reasoning and education.

These processes, or systems, happily co-exist in one brain but are rather different in the way they operate. Keith Stanovich and Richard West coined the names implicit system, or **System 1** and explicit system, or **System 2** (*Individual difference in reasoning: implications for the rationality debate*? Behavioral and Brain Sciences 2000). Daniel Kahneman, in his award-winning book *Thinking Fast and Slow* (New York: Farrar, Straus and Giroux, 2011), assigned several attributes to each system:



System 1 and System 2

What does all this have to do with DDD? Well, the point here is more about how we make decisions. It is scientifically proven that all humans are biased, one way or another. As developers, we have our own ways of solving technical problems and of course we're ready to pick up the fight when being challenged by the business about the way we write software to solve their problems. At the other hand, our customers are also biased towards their ways, they probably already were earning money without our software or, they might have some other system created twenty years ago by ancient Cobol programmers and it somehow works, so they just want a *modern* or even *cloud-based* version of the same thing. The point I am trying to make here is that we should strive to mitigate our biases and be more open to what other people say and still not fall into a trap of their own biases. It was not without a reason for Google People Operations team to create the *Unconscious Bias @ Work* workshop to help their colleagues to become aware of their biases and show some methods to deal with them.

The Cynefin complexity model requires us to at least categorize the complexity we are dealing with in our problem space (and also sometimes in the solution space). But to assign the right category, we need to make a lot of decisions, and here we often get our System 1 responding and making assumptions based on our biases and experiences from the past, rather than engaging System 2 to start reasoning and thinking. Of course, every one of us is familiar with a colleague exclaiming *yeah, that's easy!* before you can even finish describing the problem. We also often see people organizing endless meetings and conference calls to discuss something that we assume to be a straightforward decision to make.

Cognitive biases are playing a crucial role here. Some biases can profoundly influence decision-making, and this is definitely System 1 speaking. Here are some of the biases and heuristics that can affect your thinking about system design:

- **Choice-supportive bias*:*** If you have chosen something, you will be positive about this choice even though it might have been proven to contain significant flaws. Typically, this happens when we come up with the first model and try to stick to it at all costs, even if it becomes evident that the model is not optimal and needs to be changed. Also, this bias can be observed when you choose a technology to use, such as a database or a framework.
- **Confirmation bias**: Very similar to the previous one, confirmation bias makes you only hear arguments that support your choice or position and ignore arguments that contradict the arguments that support your choice, although these arguments may show that your opinion is wrong.
- **Band-wagon effect**: When the majority of people in the room agree on something, this *something* begins to make more sense to the minority that previously disagreed. Without engaging System 2, the opinion of the majority gets more credit without any objective reason. Remember that what the majority decides is not the best choice by default!

- **Overconfidence**: Too often, people tend to be too optimistic about their abilities. This bias might cause them to take more significant risks and make the wrong decisions that have no objective grounds but are based exclusively on their opinion. The most obvious example of this is the estimation process. People invariably underestimate rather than overestimate the time and effort they are going to spend on a problem.
- **Availability heuristic**: The information we have is not always all the information that we can get about a particular problem. People tend to base their decisions only on the information in hand, without even trying to get more details. This often leads to an over-simplification of the domain problem and an underestimation of the essential complexity. This heuristic can also trick us when we make technological decisions and choose something that has *always worked* without analyzing the operational requirements, which might be much higher than our technology can handle.

The importance of knowing how our decision-making process works is hard to overestimate. The books referenced in this section contain much more information about human behavior and different factors that can have a negative impact on our cognitive abilities. We need to remember to turn on System 2 in order to make better decisions that are not based on emotions and biases.

Knowledge

Many junior developers tend to think that software development is just typing code, and when they become more experienced in typing, get to know more IDE shortcuts, and know frameworks and libraries by heart, they will be ninja developers, able to write something like Instagram in a couple of days.

Well, the reality is harshly different. In fact, after getting some experience and after deliberately spending months and maybe years in death-marches towards impossible deadlines and unrealistic goals, people usually slow down. They begin to understand that writing code immediately after receiving a specification might not be the best idea. The reason for this might already be apparent to you if you have read all the previous sections. Being obsessed with solutions instead of understanding the problem, ignoring essential complexity and conforming to biases—all these factors influence us when we are developing software. As soon as we get more experience and learn from our own mistakes and, preferably, from the errors of others, we will realize that the most crucial part of writing useful, valuable software is the knowledge about the problem space, for which we are building a solution.

Domain knowledge

Not all knowledge is equally useful when building a software system. Knowing about writing Java code in the financial domain might not be very beneficial when you start creating an iOS app for real-estate management. Of course, principles such as clean code, DRY, and so on are helpful no matter what programming language you use. But knowledge of one domain might be vastly different from what you need for another domain.

That is where we encounter the concept of domain knowledge. Domain knowledge is knowledge about the domain in which you are going to operate with your software. If you are building a trading system, your domain is financial trading, and you need to gain some knowledge about trading to understand what your users are talking about and what they want.

This all comes to getting into the problem space. If you are not able to at least understand the terminology of the problem space, it would be hard (if not impossible) to even speak to your future users. If you lack domain knowledge, the only source of information for you would be the **specification**. When you do have at least some domain knowledge, conversations with your users become more fruitful since you can understand what they are talking about. One of the consequences of this is building trust between the customer and the developer. Such trust is hard to overestimate. A trusted person gets more insight and mistakes are forgiven more easily. By speaking the *domain language* to *domain experts* (your users and customers), you also gain credibility, and they see you and your colleagues as more competent people.

Obtaining domain knowledge is not an easy task. People specialize in their domains for years, they become experts in their domains, and they do this kind of work for a living. Software developers and business analysts do something else, and that particular problem domain might be little known or completely unknown when they need to obtain domain knowledge.

The art of obtaining domain knowledge is through effective collaboration. Domain experts are the source of ultimate truth (at least, we want to treat them like this). However, they might not be. Some organizations have fragmented knowledge; some might just be wrong. Knowledge crunching in such environments is even harder, but there might be bits and pieces of information waiting to be found at the desks of some low-level clerks, and your task is to see it.

The general advice here is to talk to many different people from inside the domain, from the management of the whole organization, and from adjacent domains. There are several ways to obtain domain knowledge, and here are some of them:

- Conversations are the most popular method, formalized as meetings. However, conversations often turn into a mess without any visible outcome. Still, some value is there, but you need to listen carefully and ask many questions to get valuable information.
- Observation is a very powerful technique. Software people need to fight their introversion, leave the ivory tower and go to a trading floor, to a warehouse, to a hotel, to a place where business runs, and then talk to people and see how they work. Jeff Patton gave many good examples in his talk at the DDD Exchange 2017 (`https://skillsmatter.com/skillscasts/10127-empathy-driven-design`).
- Domain Storytelling, a technique proposed by Stefan Hofer and his colleagues from Hamburg University (`http://domainstorytelling.org/`), advocates using pictograms, arrows, and a little bit of text, plus numbering actions sequentially, to describe different interactions inside the domain. This technique is easy to use, and typically there is not much to explain to people participating in such a workshop before they start using it to deliver the knowledge.
- EventStorming was invented by Alberto Brandolini. He explains the method in his book *Introducing EventStorming* (2017, Leanpub), and we will also go into more detail later in this book when we start analyzing our sample domain. EventStorming uses post-it notes and a paper roll to model all kinds of activities in a straightforward fashion. Workshop participants write facts from the past (events) on post-its and put them on the wall, trying to make a timeline. It allows the discovery of activities, workflows, business processes, and more. Very often, it also uncovers ambiguities, assumptions, implicit terminology, confusion, and sometimes conflicts and anger. In short—everything that the domain knowledge consists of.

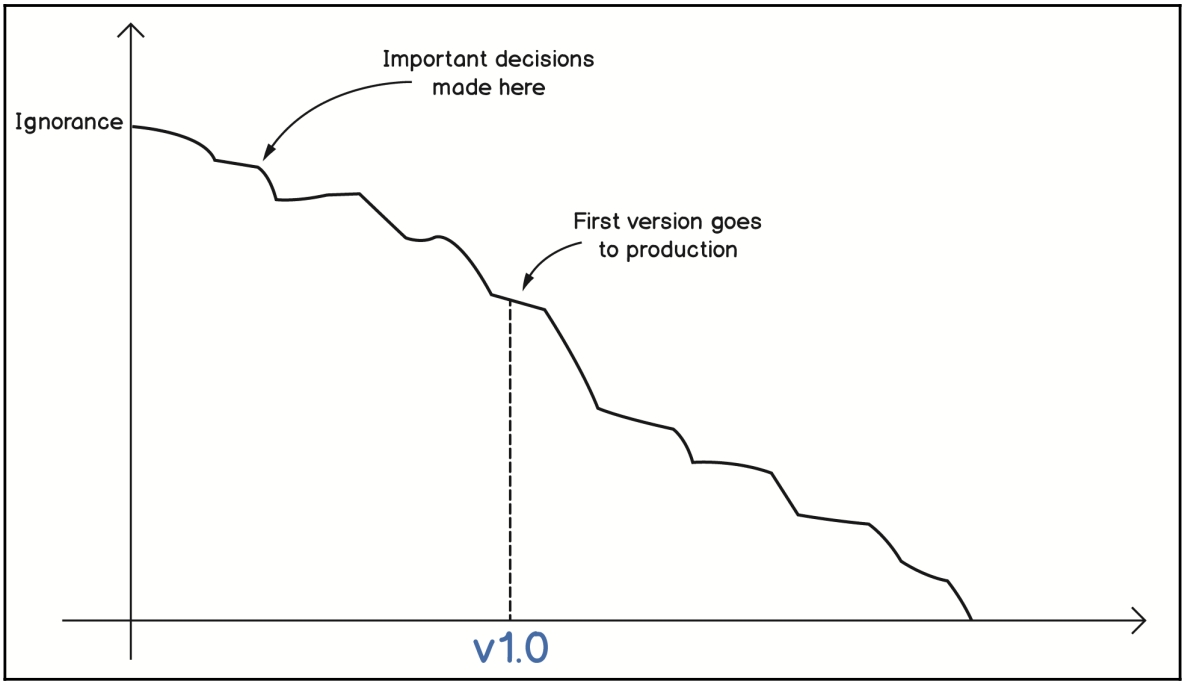
Avoiding ignorance

Back in 2000, Philip Armour published an article called *Five Orders of Ignorance* (Communications of the ACM, Volume 43 Issue 10, Oct. 2000), with the subtitle *Viewing software development as knowledge acquisition and ignorance reduction*. This message very much correlates with Alberto's quote from the previous section, although it is somewhat less catchy but by no means less powerful. The article argues that increasing domain knowledge and decreasing ignorance are two keys to creating software that delivers value.

The article concentrates on ignorance and identifies five levels of it:

- The zero ignorance level, which authors call *the lack of ignorance*, is the lowest. On this level, you have no ignorance because you have most of the knowledge and know what to do and how to do it.
- The first level is the *lack of knowledge*. It is when you don't know something, but you realize and accept this fact. You want to get more knowledge and decrease your ignorance to level zero, so you have channels to obtain the knowledge.
- The second level also called the *lack of awareness*, is when you don't know that you don't know. Most commonly, this occurs when you get a specification that describes a solution without specifying which problem this solution is trying to solve. This level can also be observed when people pretend to have competence they do not possess, and at the same time are ignorant of it. Such people might be lacking both business and technical knowledge. A lot of wrong decisions are made at this level of ignorance.
- The third level is the *lack of process*. On this level, you don't even know how to find out about your lack of awareness. Literally, you don't have a way to figure out you don't know that you don't know, which sounds like inception, but that's exactly what it is. It is tough to do anything on this level since apparently there is no way to access end users, even to ask if you understand their problem or not, in order to get down to level two. Essentially, with the lack of process, it is nearly impossible to find out if the problem you're trying to solve even exists. Building a system might be the only choice in this case, since it will be the only way to get any feedback.
- The fourth and last level of ignorance is meta-ignorance. It is when you don't know about the five degrees of ignorance.

As you can see, ignorance is the opposite of knowledge. The only way to decrease ignorance is to increase understanding. A high level of ignorance, conscious or subconscious, leads to a lack of knowledge and a misinterpretation of the problem, and therefore increases the chance of building the wrong solution:



Ignorance is highest at the earliest stages

Eric Evans, the father of DDD, describes the upfront design as *locking in our ignorance*. The issue with the upfront design is that we do it at the beginning of a project, and this is when we have the least knowledge and the most ignorance. It has become the norm to make most of the important decisions about the design and architecture of the software at the very beginning of a project when there is virtually nothing to base such decisions on. This practice is quite obviously not optimal.

In the article *Introducing Deliberate Discovery* (`https://dannorth.net/2010/08/30/introducing-deliberate-discovery/`), Dan North suggests that we realize our position of being on at least the second level of ignorance when we start any project. In particular, the following three risks need to be taken into account:

- A few *unpredictable bad things* will happen during the project.
- *Being unpredictable*, these *things* are unknown in advance.
- *Being bad*, these *things* will negatively impact the project.

To mitigate these risks, Dan recommends using *INTRODUCING DELIBERATE DISCOVERY*, that is, seeking knowledge from the start. Since not all knowledge is equally important, we need to try to identify those sensitive areas where ignorance is creating the most impediments. By raising knowledge levels in these areas, we enable progress. At the same time, we need to keep an eye on new troublesome areas and resolve them too; and this process is continuous and iterative.

Summary

In this chapter, we briefly touched on the concepts of problem and solution spaces, requirements specifications, complexity, knowledge, and ignorance. Although at first, these topics don't seem to be directly related to software development, they have a significant impact on how and what we deliver.

Don't fall into the trap of thinking that you can deliver valuable solutions to your customers just by writing code and that you can deliver faster and better by typing more characters per second and writing cleaner code. Customers do not care about your code or how fast you type. They only care that your software solves their problems in a way that hasn't been done before. As Gojko Adžić wrote in his sweet little book about impact mapping (*Impact Mapping: Making a Big Impact With Software Products and Projects*, 2012, published by Provoking Thoughts), you cannot only formulate user stories like this:

- As a *someone*
- To *do something*
- I need to *use some functionality*

Your user, *someone*, might be already doing *something* by executing *some functionality* even without your software: using a pen and paper, using Excel, or using a system from one of your competitors. What you need to ensure is that you make a difference, make an impact. Your system will let people work faster, more efficiently, allow them to save money or even not to do this work at all if you completely automate it.

To build such software, you must understand the problems of your users. You need to crunch the domain knowledge, decrease the level of ignorance, accurately classify the problem's complexity, and try to avoid cognitive biases on the way to your goal. This is an essential part of DDD, although not all of these topics are covered in the original *Domain-Driven Design: Tackling Complexity in the Heart of Software* by Eric Evans, although known by the DDD community as the *Blue Book*.

In the next chapter, we will do a deep dive into the importance of language and discover the definition of Ubiquitous Language.

Further reading

Here is a list of information you can refer to:

- *A Leader's Framework for Decision Making*, Snowden D J, Boone M E. (2007), Harvard Business Review 2007 November issue
- *Thinking, Fast and Slow* (First edition), Kahneman, Daniel (2011), New York: Farrar, Straus, and Giroux
- *Impact Mapping: Making a Big Impact With Software Products and Projects*, Adžić, G. (2012), Provoking Thoughts.