Table of Contents

The Mechanics of Domain-Driven Design . . . 1
- Strategic design tools . . . 1
  - Business model canvas . . . 2
  - Wardley maps . . . 3
  - Impact maps . . . 3
- Tactical design tools . . . 4
  - BDD and TDD . . . 4
  - Contract testing . . . 6
- Summary . . . 7
- Questions . . . 7

The Mechanics of Domain-Driven Design

> When eating an elephant, take one bite at a time.
>
> — Creighton Abrams

As mentioned in the previous chapter, many things can render a project to veer off-course. The intent of DDD is to decompose complex problems on the basis of clear domain boundaries and the communication between them. In this chapter, we look at a set of tenets and techniques to arrive at a collective understanding of the problem at hand in the face of ambiguity, break it down into manageable chunks and translate it into reliably working software.

The following topics will be covered in this chapter:

- Understanding the problem using strategic design
- Strategic design tools
- Implementing the solution using tactical design
- Tactical design tools

By the end of this chapter, you would have learned how to apply strategic design techniques to break down the problem into different types of subdomains, evolving domain models and bounded contexts as part of the solution, also a high level introduction to popular strategic design tools like Business model canvas, Wardley maps and Impact maps. You also would have learned how to apply tactical design techniques for the solution along with introductions to tactical design tools like Behavior driven development, Test driven development, Contract testing etc.

Strategic design tools

To arrive at an optimal solution, it is important to have a strong appreciation of the business goals and their alignment to support the needs of the users of the solution. We introduce a set of tools and techniques we have found to be useful.

It is pertinent to note that these tools were conceived independently, but when practiced in conjunction with other DDD techniques can accentuate the effectiveness of the overall process and solution. The use of these should be considered to be complementary in your DDD journey.

Business model canvas

As we have mentioned several times, it is important to make sure that we are solving the right problem before attempting to solving it right. The business model canvas is a quick and easy way to establish that we are solving a valuable problem in a single visual that captures nine elements of your business namely:

- *Value propositions*: what do you do?
- *Key activities*: how do you do it?
- *Key resources*: what do you need?
- *Key partners*: who will help you?
- *Cost structure*: what will it cost?
- *Revenue streams*: how much will you make?
- *Customer segments*: who are you creating value for?
- *Customer relationships*: who do you interact with?
- *Channels*: How do you reach your customers?

Here is a sample canvas for a popular movie subscription provider:

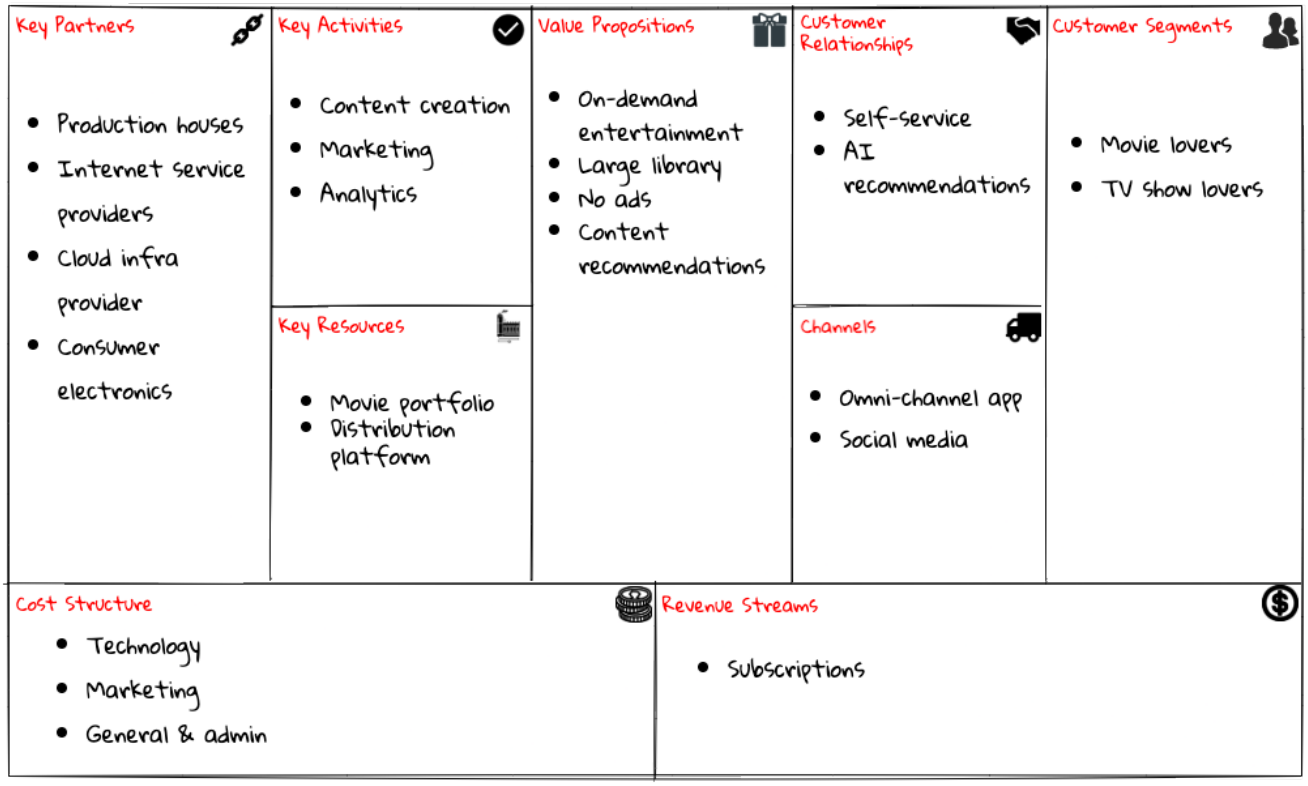

Figure 1. Business model canvas of a popular online movie subscription provider

The business model canvas helps establish a shared understanding of the big picture among a varied set of groups including business stakeholders, domain experts, product owners, architects and developers. We have found it very useful when embarking on both greenfield and brownfield engagements alike.

A variation of the business model canvas is the *lean canvas*, which is a one-page document that has been adapted from Business Model Canvas that is entrepreneur focused and has customer-centric approach that emphasizes on problem, solution, key metrics and competitive advantage.

Wardley maps

The business model canvas can help establish clarity of purpose at a high level. The Wardley map is another tool to help build a business strategy. It provides a sketch of the people that the system is built for, followed by the benefits the system offers them and a chain of needs required to provide those benefits (called the *value chain*). Next the value chain is plotted along an evolution axis which ranges from something that is uncharted and uncertain to something that is highly standardized.

Here is a sample Wardley map for a bank that is looking to provide a suite of next generation credit card products:

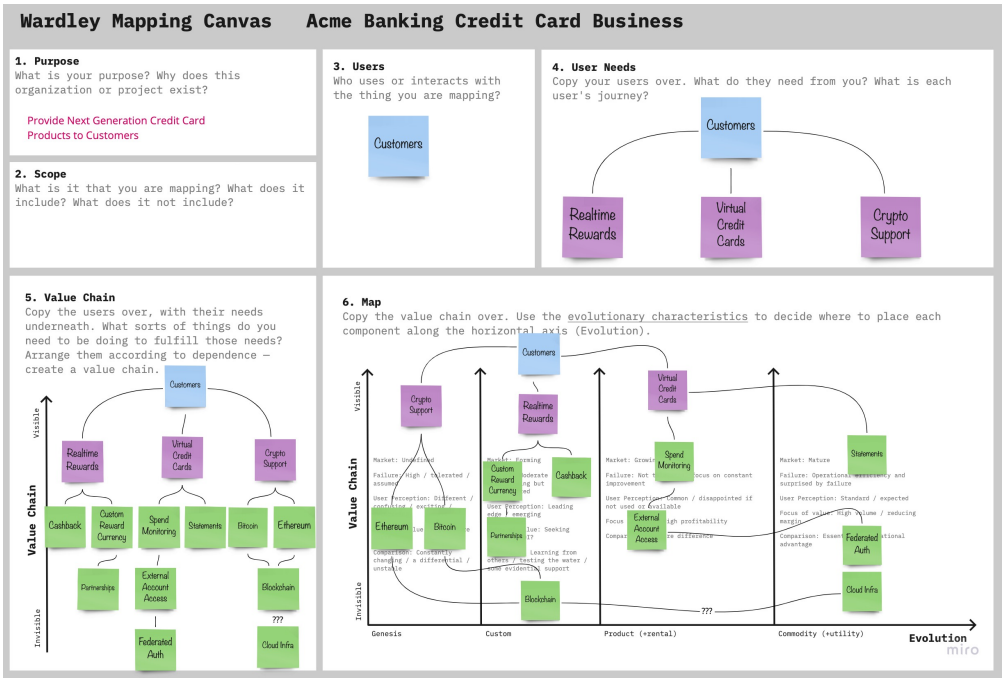

Figure 2. Value chain to a Wardley map

The Wardley map makes it easy to understand the capabilities provided by our solution, their dependencies and how value is derived. It also helps depict how these capabilities play out in comparison to those offered by competitors, allowing you to prioritize attention appropriately and make build versus buy decisions.

Impact maps

An impact map is a visualisation of scope and underlying assumptions, created collaboratively by senior technical and business people. It is a mind-map grown during a discussion facilitated by

considering the following four aspects:

- *Goals*: **Why** are we doing this?
- *Actors*: **Who** are the consumers or users of our product?. In other words, who will be impacted by it.
- *Impacts*: **How** can the consumers' change in behavior help achieve our goals? In other words, the impacts that we're trying to create.
- *Deliverables*: **What** we can do, as an organisation or a delivery team, to support the required impacts? In other words, the software features or process changes required to be realized as part of the solution.

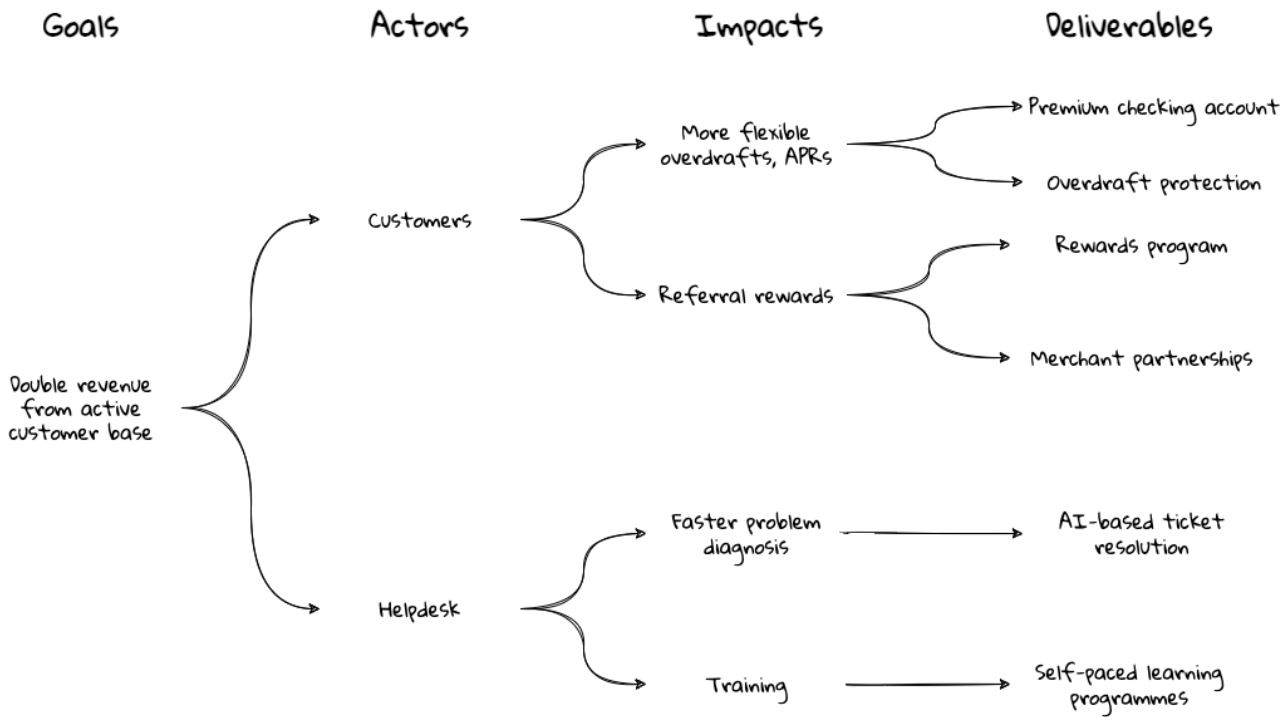

Figure 3. A simple impact map for a retail bank

Impact mapping provides an easy to understand visual representation of the relationship between the goals, the users and the impacts to the deliverables.

Tactical design tools

As we have mentioned a few times, DDD is about making sure that we build the right thing and then build it right. While strategic design tools help on the former, tactical design tools help on the latter. Let us look at set of tools and techniques we have found to be useful.

BDD and TDD

Test-Driven Development (TDD) was conceived by Kent Beck as a means to encourage simple designs and inspire confidence by writing a **test before writing the production code** required to satisfy that requirement. The intent behind this was to arrive at an optimal design iteratively, guided by a set of executable test cases. Unfortunately, it got misconstrued as a unit-testing technique as opposed to a design technique, leading to it being employed in a manner where teams

were unable to derive the benefits of the practice. Behavior-Driven Development (BDD) was conceived by Chris Matts and Dan North as a means to practice TDD the right way by using better terminology (for example, using *specifications* instead of *test suites*, *scenarios* instead of *tests*, *confirming behavior* instead of *testing code*, etc). Consider the example of the overdraft facility for premium customers. A potential test for this feature using **JUnit** may look something like this:

```
class CheckingAccountOverdraftTests {

    @Test
    public void shouldAllowOverdraftForPremiumCustomers() {
        Account account = Account.checking("ABC123")
                              .withCustomer(premiumCustomer())
                              .withBalance(Money.usd(100));

        account.withdraw(Money.usd(150));

        assertThat(account.balance()).isEqualTo(Money.usd(-50));
    }
}
```

The BDD specification for this same feature using **Cucumber** and **Gherkin** looks something like this:

```
Feature: Overdraft capability for premium customers

Scenario: Should allow overdraft facility for premium customers.

Given I am a premium customer with account number "ABC123"
  And I have 100 dollars in my account
 When I attempt to withdraw 150 dollars
 Then I should have a negative balance of 50 dollars
```

Conceptually, there is no difference between the two versions, although it is arguable that less technical domain experts will likely prefer the BDD specification over the TDD test case because it uses the language of the problem domain in an implementation-neutral manner. This has led to BDD being used for more coarse-grained acceptance testing, while TDD gets used for more fine-grained unit testing. In our experience, both these techniques complement each other in achieving better software design. Because BDD tools (like Cucumber[1], JBehave[2]) allow the use of implementation-neutral specification languages like Gherkin, they tend to be more approachable for less technical stakeholders. However, a plethora of tools like easyb[3], Mockito[4], AssertJ[5] etc. make it fairly natural to adopt the BDD style even in Java, proving that they are very complementary as shown here:

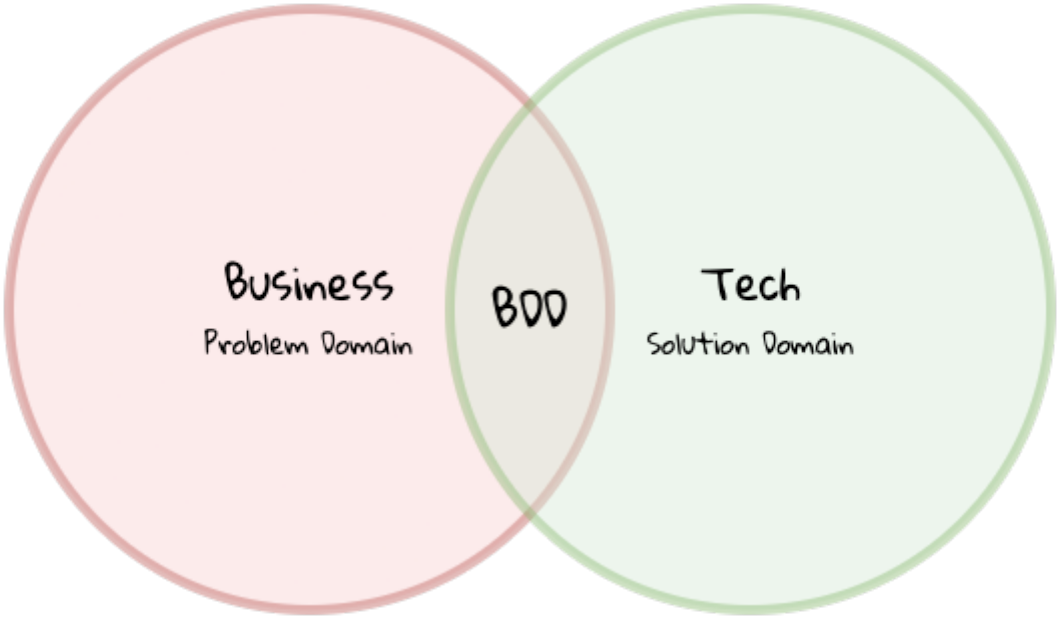

Figure 4. TDD and BDD are complementary concepts.

Both BDD and TDD, when used in close conjunction with DDD's ubiquitous language, can promote closer synergy between the problem and solution domain and in our experience is an indispensable tool when building software solutions.

Contract testing

When implementing a sound test strategy, one encounters two broad classes of tests: **unit** and **end-to-end** (and everything in between). Unit tests tend to be fairly fine-grained, especially when they mimic the behavior of collaborating components using mocks/stubs. This allows us to run a large suite of such tests while consuming very little time. In an ideal world, we would prefer restricting ourselves to running just unit tests. However, unit tests do have a limitation in that the assumptions made while mocking/stubbing collaborator behavior may become inaccurate when the owners of these components make changes to their respective solutions. This may lead to a situation where unit tests work just fine, but the solution as a whole does not work in a formal end-to-end environment, resulting in non-technical stakeholders losing faith in these tests.

To restore confidence, teams then resort to verifying functionality by mostly running time-consuming, error-prone manual end-to-end tests, occasionally throwing in some automation. In our experience, running a stable, non-trivial suite of automated end-to-end tests remains quite a challenge, due to the computational and cognitive expense involved in setup and ongoing upkeep. Because these tests require large portions of the real solution stack to be in place, they tend to often happen very close to the end, causing them to be rushed and coarse-grained.

What we need are tests that both run rapidly, are easy to maintain (like unit tests), and provide a high degree of confidence that the system is functionally correct when they pass (like end-to-end tests). Contract tests can act as the missing link between unit and end-to-end tests by affording a set of *blessed* mocks (those that are always compatible with the real implementation). We will cover contract testing in more detail in Chapter 10.

In addition to the tools and techniques discussed above, we would like to call out domain story telling[6] and eventstorming[7] as two valuable techniques that cover aspects of both strategic and

tactical design. We cover both these techniques with more concrete examples in Chapter 4.

Summary

In this chapter we looked at the differences in perspective that arise due to problem domain and solution domain thinking. We examined how to arrive at a shared understanding of the problem domain using DDD's strategic design elements. We introduced a few tools and techniques that can aid in accelerating the strategic design process.

We also looked at how to build robust solutions using DDD's tactical design elements along with tools and techniques that can enhance our journey of building solutions that can evolve durably.

In the next chapter we will take a closer look at where DDD fits in the scheme of various architecture approaches, patterns and how it is applicable in a wide array of scenarios.

Questions

1. Can you draw a business model canvas for the current ecosystem you are working with? Did this exercise help in enhancing your understanding of the big picture?
2. Are you able to identify the different subdomains in your problem domain? Do your solutions (bounded context) align with these subdomains?
3. Are you able to draw a simple context map of your current ecosystem?
4. Do your bounded contexts align along specific aggregate roots?

[1] https://cucumber.io/docs/installation/java/

[2] https://jbehave.org/

[3] https://easyb.io/

[4] https://site.mockito.org

[5] https://joel-costigliola.github.io/assertj/

[6] https://domainstorytelling.org/

[7] https://www.eventstorming.com/