JavaScript Domain-Driven Design

JavaScript backs some of the most advanced applications. It is time to adapt modern software development practices from JavaScript to model complex business needs.

JavaScript Domain-Driven Design allows you to leverage your JavaScript skills to create advanced applications. You’ll start with learning domain-driven concepts and working with UML diagrams. You’ll follow this up with how to set up your projects and utilize the TDD tools. Different objects and prototypes will help you create models for your business process and see how DDD develops common language for developers and domain experts. Context map will help you manage interactions in a system. By the end of the book, you will learn to use other design patterns such as DSLs to extend DDD with object-oriented design base, and then get an insight into how to select the right scenarios to implement DDD.

Who this book is written for

If you are an experienced JavaScript developer who wants to improve the design of his or her applications, or find yourself in a situation to implement an application in an unfamiliar domain, this book is for you. Prior knowledge of Javascript is required and prior experience with Node.js will also be helpful.

What you will learn from this book

- Make communication with businesses more productive by identifying your requirements
- Analyze a project to build the right features at the right time
- Test the vital parts of your project by implementing test-driven design
- Create application prototypes and utilize UML to enhance developer productivity
- Explore effective means of communicating with non-developers to find out more about their requirements
- Discover how to create manageable applications in JavaScript from scratch
- Learn to quickly build advanced JavaScript applications

Philipp Fehre

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 2 'Finding the Core Problem'
- A synopsis of the book’s content
- More information on JavaScript Domain-Driven Design
About the Author

Philipp Fehre is a software engineer, conference organizer, speaker, and educator. Having seen many different applications as a consultant, he moved on to educate developers as an advocate of the NoSQL technology and is now working as a backend software engineer. He has also been a writer for the German Linux magazine in the past.

For years, he has been fascinated by the possibilities of JavaScript and followed Node.js from its beginnings, using it for research projects and production web services.
Welcome to *JavaScript Domain-Driven Design*. For years, JavaScript has been stuck in the realm of making a website a little more interactive, but nobody would have thought about implementing whole applications in JavaScript. This has changed dramatically over the past few years, and JavaScript has evolved into this omnipresent powerhouse of a language that is present in almost every space of development.

This phenomenal growth has introduced many problems during development, which were previously unknown in the JavaScript world. Projects grow to very large codebases, many developers work simultaneously on these large codebases, and in the end, JavaScript is more often than not a vital part of the overall application. The good thing is that most of those problems have been solved before, and we, as JavaScript developers, can draw from the vast experiences gained over the years in other spaces and adapt them to work for us in JavaScript, leveraging JavaScript’s unique flexibility along the way.

**What this book covers**

*Chapter 1, A Typical JavaScript Project*, introduces a typical business application and how it is developed. It shows how domain-driven design can help steer clear of common issues during the development to create a more problem-tailored application.

*Chapter 2, Finding the Core Problem*, shows how we can effectively explore an application’s problem domain and identify the most important aspects to work on.

*Chapter 3, Setting up a Project for Domain-driven Design*, focuses on setting up a structure for the project that is ready to grow. It not only shows how we can lay out files and folders, but also creates the right testing and build environments.

*Chapter 4, Modelling the Actors*, shows how a project grows using object-oriented techniques together with domain-driven design to really isolate the domain. We also tackle one of the hardest problems of computer science, naming.
Chapter 5, *Classification and Implementation*, is all about the language we use in domain-driven design to make the project understandable and readable. We look at the relationship between domains and sub-domains, and then drill further down into the core of domain objects.

Chapter 6, *Context Map – The Big Picture*, is about not only growing the application from a technical perspective, but also from an organizational perspective. We talk about organizing the different parts that form the application as a whole, either as separate parts or as interlinked pieces.

Chapter 7, *It's Not All Domain-driven Design*, talks about fitting domain-driven design into the space of development techniques, talking about what problems fit where. We also talk about influences such as object-orientation, domain-specific languages, and more.

Chapter 8, *Seeing It All Come Together*, is about how our project fits into a space of projects in JavaScript, referring back to the beginning. We also look at alternative choices for frameworks and development styles.
Finding the Core Problem

Every piece of software is written to solve a problem, and in turn is a perfectly valid solution for this exact problem. Sadly, the problem a piece of software solves so perfectly is not always the problem the software was created for in the first place or even the problem the programmer had in mind when the software was written. The history of programming is full of examples where developers tried various ways to come up with a way to be able to perfectly state a problem first, and then implement a solution. Developing software based on the waterfall model is a great example of having a nice idea that failed to deliver on the promise. When you ask the parties involved about the failure, the reason will most likely be that the problem diverged from the specification, or the specification was misunderstood in—according to one party—a very obvious way. So, why is this?

When starting a software project, especially one motivated by a business need, we set out to model a part of the real world and apply a set of constraints and algorithms to it, to ease the job of one or more parties involved in the business. The problem is that the party that has the issue the developer is trying to solve is most likely not the developer. This means that the developer first has to get an understanding of what the request really is to actually know what is supposed to be developed.

How can we get a deep enough understanding of a certain part of the business without the (most likely) years of experience our clients have ahead of us? The solution to this, and the most likely problem, is communication. We need to find a way to explore the problem deeply enough, and backed with our knowledge of how to model a world in software, to be able to ask the right questions. We need to do this in such a way we don't lose the non-technical people so that we can draw from their understanding. This comes back to the language mismatch between developers and business people, and it is probably the biggest obstacle to overcome. In domain-driven design, this is referred to as the ubiquitous language of the project, a language shared by all parties involved in the project. Establishing this kind of language allow us to communicated clearly across team boundaries, and as mentioned before, this is one of the core ideas in domain-driven design.
Coming back to our example of the orcs running a dungeon, we don't know how this is done; we don't even completely understand the constraints that cultural aspects involve or apply. The world of the orcs is one in which we are an outsider who can only watch, ask questions, and model it according to our understanding. We naturally have to trust the local experts. Even though we aren't as much of an outsider in real-world problems, we should always try to view the problem from the outside as much as possible because, in a business that has taken years to grow, our own assumptions are probably wrong anyway.

In the following, we are going to explore the problem and introduce a set of tools that will help to do this. We will cover several aspects, but most importantly the following:

- Using pen and paper for programming
- Code spikes and throwaway code
- Mapping our actors out—creating a dependency graph for our domain

Exploring a problem

There are not many problems in software development that can be fully specified easily. Even the few that seem like it leave some room for interpretation. When working on a project to implement a database adapter, I recently faced exactly this. There was a specification that needed to be implemented, and a set of unit tests making sure the implementation conforms to the specification. However, as I implemented it, I found myself asking questions along the way. The main question was very similar to what I would have asked if I hadn't had the specification: How are people going to use this piece of code? In a lot of cases, there are multiple ways to implement a certain feature, but picking one often means weighing different tradeoffs against each other, such as speed, extensibility, and readability.

In our orc dungeon, we have to ask the same basic question: How is the client going to use our software? Sadly, this question by itself will not yield the results we have in mind. The problem is that our users don't know our software. Basically, our future users have the same problem we do: they don't know what the software is going to look like when it is finished but can only guess its usage. This really is the catch 22 of software development; thus, so to be successful, we need to find a way around this. We as developers need to find a way to make the process of development possible to grasp for our future users, and our future users need to adapt concepts of the highly descriptive language we use to state intentions as clearly as possible.
Software is really an abstract concept, and most people are not used to talking about abstract things. So, the first step toward a better understanding is to make it more approachable for the users. We need to make the concepts touchable; this can be done in various ways, but the more haptic the better.

Use paper. As developers, we often prefer to go paperless, but writing things on paper makes it easier for most people to understand concepts, so writing things down can be immensely helpful.

**Outlining the problem**

As far as techniques to illustrate and organize pieces of information go, outlining proves useful in many cases. But, how can we outline software? The idea is to keep all the information that comes up when talking to the business experts in an easily searchable format. In a lot of places, this is a wiki, but it can also just be a set of shared text files that are readily accessible whenever information needs to be added or retrieved. Outlining here means to store information nested by topic and drill down as needed.

**Tracking knowledge**

When starting with collecting information, the most important part is to collect as much information as possible, and to do this it needs to be made seamless. It is also important to keep the information organized to be added to as well as to be restructured as needed. As with our software, we don't know the structure of the outline to start out with, so we just add a new piece whenever we identify a new entity, actor, or any important piece of the system. Therefore, don't invest too much time making the current structure perfect, but rather make it just good enough for now.

Make a habit of collecting any information that you come across, and keep the application outline at hand. In a lot of companies, the hallway track is often an immensely valuable source of information, so make sure to use it.

What makes an outline so useful is that you will be able to restructure it easily, and this is also what you should aim for when deciding on the tool to keep these outlined notes. Reordering notes needs to be quick and intuitive. The goal right now is to keep the cost of change as low as possible, so we can easily explore different paths.
Finding the Core Problem

Our dungeon information that we collected so far can be represented like this:

```markdown
# Dungeon
receives prisoners
transfers from other dungeons
new captures
loses prisoners
transfers to other dungeons
fleeing
prisoners might flee during transfer
prisoners might flee from the dungeon itself
```

The important part is that this structure is very easy to modify and keep up-to-date as new information arrives, and we can already see that a new entity emerges from the outline—the prisoner. With this new information, we now add this to the outline to have a place to hold more information about prisoners, since they are obviously a vital concept to our dungeon application.

```markdown
# Prisoner
can flee a dungeon or transport
needs a cell
is transferred between dungeon
```

This is essentially what the outline is about, recording information and drawing quick conclusions.

The medium

Depending on the situation, different mediums are possible or preferable to hold the information. This can stretch from a piece of paper up to a full-blown wiki system. The format I prefer to use for my outlines is Markdown, which has the advantage of being stored as plain text and being very readable without being processed. Also, to generate some documentation to print out, it is useful to process it to HTML first. This is by no means the ultimate choice, and you should choose whatever feels the most natural, as long as it is simple to edit and readily available in as many places as possible. The one thing that is important, is to choose a system that does not lock you in to its way of doing things or as into a data format that is hard to export or change.
Paper programming

In our quest to involve non-programmers in the process of software creations, it is important to make concepts approachable. We need to illustrate interactions as well as actors of the system and make them ready to be moved around. Often, it helps to have something people can actually hold in their hand and move across a table when talking about a subject. The best way to achieve this is to create paper representations of the elements of the system. Create a paper-based, hand-operated version to touch and interact with right there and then. This is often known from UI design, where paper prototypes are a common thing, but it also works well to create a version of non-UI parts of the application.

The idea is to draw out any pieces of the system on cards to be combined, separated, and added to. When this is done, it often ends up being pretty close to the entity representation we will later have in the system. When starting out using this technique, it is important to note that the end result will always be in a certain state. As things get moved across the table, and elements are modified, we need to keep track of the information that is generated. Make sure to keep notes along the lines of how certain actions evolved during the discussion as a single picture of the end result is just reflecting one state.

So how does such a paper program work?

When starting out, we lay out all the information we currently have, drawing out boxes for all the elements, and name them. In our case, we will draw out the dungeon, prisoners, cells, and a transport. For now, these are the entities we interact with. At this point, we think about a specific interaction and try to represent it with the entities and other objects we currently have. Let’s transfer a prisoner from our dungeon into another; to do this, we need to think what we have to do:

- The dungeon keeper notifies the other dungeon
- The prisoner is transferred from a cell onto the transport
- An orc is assigned to guard the transport
- The transport travels to the other dungeon
Finding the Core Problem

When drawn it out on a sheet of paper, the result may look a little like this, where the numbers are the order in which the steps appeared:

![Diagram showing interactions between Dungeon Keeper, Orc, Cell, Transport, Other Dungeon, and Prisoner]

At this point, we already notice that multiple pieces are missing, mainly the dungeon keeper and some way to notify the other dungeon. So, how can these be added? The dungeon keeper clearly is an entity that manages the dungeon, so a separate card should be added. Also, the notification is done via messages, so we add a messaging system. This is a new subsystem, but we can for now consider it a black box we can drop messages into have them arrive at the other side.

Now that the systems are in place, we can add the needed methods to the actors of our system: The dungeon keeper, to request a transfer, needs a way to send a message; the cell needs to give up the ownership of the prisoner; the transport needs to take ownership; and so on. As we move through this interaction we can clearly see one possible way this can be modeled and this is also more approachable for non-developers as they see actual boxes moving across the table. As this model is in constant flux, make sure to keep notes in your outline along the way, to not lose any of the newly acquired information.

Not so scary UML

Our paper, a prototype, gives us a nice picture of the interaction, and our outline captures a lot of information about how the program should behave in various cases. It also captures details on the naming side of things from a business perspective. All in all, this gives us a lot of good insight, but there is still a part missing. This makes the information out of our paper prototype durable enough, so we can more easily reference it as we move along. The prototype we drew earlier is missing some information that is important to the implementation. We need to capture more of the structure of the application.
This is the point where Unified Modelling Language (UML) comes into play, yes this scary piece of waterfall-infused practice that most of us never thought of as being useful. When talking about UML is it often referenced as the idea to encode all modeling information in a diagram; so ultimately code could be generated and filled out by basically everybody with some amount of coding skills. Of course, this does not work, but UML still has some interesting properties that make it useful. What we are setting out to do is leveraging one property of UML, and this is the ability to capture interactions in a concise form. UML defines multiple categories of diagrams:

- Structure diagrams
- Behavior diagrams
- Interaction diagrams

A **structure diagram** focuses mostly on the actors in the system and their relationships. In our case, it would express the relationship of the keeper toward the dungeon and other orcs for example. This can be helpful when many actors are involved, but is not necessarily the most important information to start out with.

![Structure Diagram](image)

A use case diagram gives a slightly more detailed picture of the actors in the system, and their interaction with each other. A use case diagram is part of the behavior diagram family and therefore focuses on the behaviors of the actors. This is not only useful information for our system, but also too coarse grained at the moment to express the flow of information and actions.

![Behavior/Use Case Diagram](image)
Finding the Core Problem

As our feature does involve interaction between the defined actors of our system, a useful thing to explore is the sequence of events as they happen in our system. For this, we can use a sequence diagram, which is a type of interaction diagram in UML. This kind of diagram focuses on the sequence of events that need to happen to achieve a certain goal. Some of this may be asynchronous, some needs to await a response; all this is captured in a single diagram:

With this kind of illustration, it is easy to distinguish between synchronous and asynchronous messaging, so we can be sure to model the methods accordingly. Also, naming things is known as one of the hardest problems of computer science, so be sure to show this to your domain experts to draw from their language to name the now exposed messages and methods.

So far, the idea of every part has been to have the tools to explore the problem from different view perspectives, but don't drink too much of the Kool-Aid! We don't try to create a complete description of the whole system, but rather explore one part just deeply enough so that we can get a sense of what its core functionality is going to be and how it makes sense to implement it. We can then remove the uncertainties, by asking the right questions as we know the domain well enough so that we are able to explore the business domain together with the experts.
Involving the experts

As we explore the domain from every angle, it is important to talk to the people who
know as much as one can about it. One of the core ideas of domain-driven design is
to create a language around the domain that can be spoken by each party involved.
When talking about the tools, we set out to create them in such a way developers as
well as domain experts can take part on an equal footing, so each can draw from the
other's knowledge to approach a problem.

The spoken language is a problem in itself, so for developers it needs to be as
unambiguous as possible, because very concrete and specific ideas need to be
expressed. There should be no room for misinterpretation. For business people, on
the other hand, it needs to be understandable for such a non-technical audience.
Now comes the important part, where we actually are going to see whether we
have achieved this goal so far, and how we are able to communicate the ideas of the
domain back and forth.

When involving the experts of a domain, we should first have a clear idea of what
we are trying to achieve, such as gaining knowledge about the system we are
currently developing. It is a natural tendency for developers to make their system
shine in the best light, but our goal is to expose misunderstandings and uncertainties
in our design and understanding so far. We actually want to get caught off-guard,
so to speak. For the current stage of the project, this should be considered an
achievement. Right now, change is as cheap as it is going to get, so if we expose
a certain gap in our knowledge, we make our lives easier down the road. Getting
a misunderstanding exposed right now also means that we were able to ask all
the right questions such that we were able to communicate this abstract idea of a
software system successfully; thus, the business side was able to dive into our system
and correct the flaws. If we get to this point, non-developers are actually involved in
the development and we can move forward developing a very well-suited system.
So, how do we get there?

Finding the gaps

The first thing we now have to do is to start the conversation flowing. As with
most problems, it is best to think about them in a diverse group, so we get the most
viewpoints. To get there, we want to create an environment where the business
domain experts can explain to us what is going on. We can use all the different
techniques now to talk about our software in an accessible fashion. The idea of paper
programming can come in very handy at this stage.
Finding the Core Problem

So first we need to prepare, make sure all the units that have been identified are prepared. Have cards ready for everybody to move around and write on them as the actions are illustrated and gaps are identified in the knowledge. It is also helpful to take a picture of the current state with notes attached to save the state for later reference as the ideas evolve. The conversation can start out with the developers explaining how they think the system works, encouraging the business experts to interject whenever there is something unclear or just plain wrong. This can really become a kind of game. How can an action we want to express be expressed with the pieces available? Of course, this is not a puzzle, so you are able to create new pieces at will and change them as needed. Being guided through the process in such a way will most likely expose several valuable properties in the system.

![image]

Being precise is what it is all about; make sure to ask questions such as And this is how it is done 100% of the time? as often as possible.

So, let's walk through an example feature of our software: transferring a prisoner to another dungeon.

Talking business

The process of transferring a prisoner has been described in three steps:

1. The dungeon keeper notifies the other dungeon.
2. The prisoner is transferred from a cell onto the transport.
3. The transport travels to the other dungeon.

So, we have some cards prepared:

- The notification service identified by an envelope
- The dungeon cell
- The prisoner
- The transport

With the available cards, we can let the orc master describe precisely what needs to happen when a prisoner is transferred.

The orc master identifies the problem, as follows, and sends out a raven with the notification of the transfer request to the dungeon. He then goes to the cell to move the prisoner out and on to the transport, assigning an orc to guard the transport and sending it off to the other dungeon.
In this short description, we see multiple differences from our model that need to be addressed.

1. The order of one and two does not actually matter, as long as there is at least one prisoner in the dungeon, which we can check at notification time.
2. There is going to be another scarce resource involved, and these are the guards to man the transport; they need to be available, and their flow in and out will need to be tracked.

Given the new insights, we can now model this event as actors in our system pretty accurately. It is important to note that our system of course does not need to represent the flow directly in code but, from a high-level point of view, having a consistent flow makes sense since it has established itself through (possibly) years of practical use. Thus, it is at least a good point to start after all.

**Talking about the actors**

When talking about how to implement a certain feature, several forms of objects are involved, all of which have certain distinct roles in the system. A lot of these roles exist in many systems, even though they may be named differently. In a domain-driven design, the classification of these roles makes a big difference. The reason is that, if we classify something, there is a certain set of patterns that can be applied right away, since it has proven itself to be useful. This is very similar to the idea of naming the patterns that have emerged in enterprise applications and are by now almost basic knowledge to most application developers.

In domain-driven design, we have multiple building blocks to choose from:

- Entity
- Value object
- Aggregate
- Domain event
- Service
- Repository
- Factory

Most of the elements in this list probably make sense to you as a developer already but, if not, we are going to define each of these more explicitly later. For now, let's focus on the ones we need and we are already using in the system: aggregate, value-object, and domain events.
One important distinction is the difference between an entity and a value object. While an entity is defined by its identity, a value object is defined by its properties. Going back to our prisoners and cells, we can see that it is possible to use either classification, but it changes the focus. If a prisoner is an entity, each prisoner is clearly defined, and two prisoners will always be different. Classifying them like this makes prisoners traceable throughout the system, as they move from dungeon to dungeon and cell to cell. This may be really useful, but may be an overkill as well. This is what the current stage is all about—finding the focus of the project from a domain point of view. So let’s walk through the whole process step by step.

Starting from the outside in, we first have to think about our domain event. As the name suggests, this is the event that triggers a certain reaction by the domain; in our case, this is the transfer of prisoners. To handle the events, we have to move one level down and think about the parts of our system that handle the transactions on our resources, the aggregates. They are, so to speak, the actors in the system as they aggregate all the needed entities, value objects, and everything else needed to present a consistent view to the outside world. Aggregates are also responsible for changing the state of the world in our system according to the domain’s need. As far as aggregates go, there are multiple ones that are responsible for the action: the dungeon keeper managing cells, prisoners, and keepers, as well as the transport being a mobile cell, the prisoner, and the keeper. Notifications to other dungeon live somewhat outside the system, so classifying these as a service seems like the natural thing to do. OK, this wasn't too hard, and thinking about the classification of different object is quite natural.

Using the provided domain terms lets us state the intended focus and level of the parts clearly. Other developers, even if they are unfamiliar with the system, are now able to assume a given feature set from each named entity. For us, the naming is a form of documentation that allows us to notice quickly when we start to intermix concepts.

**Identifying the hard problem**

Over the last sections, we started to gain a solid understanding of the interactions in the system. Now it is time to leverage this understanding and move on to implementing our software solution. So, what should we start with when developing the software?
Often when a project is started, we like to start with the easy part, maybe create a project from a template— for example, running a framework code generator, such as Node.js Express, in a new folder to set us up with the scaffold structure for our project. At first, this seems like a really good option as it creates a lot of the boilerplate code we would have to write to create an Express project otherwise. But, does it move us closer to solving a business problem? We now have a code base to explore but, as it is auto-generated, we don't have any domain-specific code obviously. On the other hand, we have locked ourselves in a fixed structure. For some projects, this is a good thing; it means that there are fewer things to think about. However, if we try to solve a lower-level problem, it is arguably bad to lock yourself into a certain mindset.

We need to identify the problem and determine how to deliver value to the business as fast as possible. This will drive user adoption and development of the software further. So far, we explored one part of the domain, which seemed important enough to our business to explore implementing it as our first feature. Now, it is time to drill down into it to see where the core problem lies, seeing the objects that will be involved and their interaction with our software.

**Mapping the dependencies**

From our previous work, we have a pretty clear understanding of the objects involved, at least on a high level:

- Notification Service
- Cell
- Prisoner
- Keeper
- Orc master
- Transport

With these in mind, our task is now to find a place to start. When laying out these objects, it is clear that they all have some dependency on other parts, and we can leverage this. We draw up each object, using arrows to demonstrate which objects it depends on. This is known as a **dependency graph**:

![Dependency Graph](image)
The graph shows us the dependencies for each of the actors we identified. The keeper, for example, is a necessary dependency for the transport as well as the orc master. On the other hand, the orc master depends not only on the keeper, but also on the transport, prisoner, and cell as well. Looking at the graph, we can see in which order the elements need to be implemented. Elements we identified as aggregates before are of course going to have the most dependencies. They, as their name suggest, aggregate multiple objects into one unit for common access and modification.

One way to attack the problem is to start out in the following order:

1. Keeper.
2. Cell.
3. Prisoner.
4. Transport.
5. Notification service.
6. Orc Master.

The nice thing is that, along the way, we can present an intermediate state back as soon as one of the aggregates is in working order. We can talk about our idea of a transport, and align it with the expected feature for example. The working condition is an important point here, since it is really hard for people to judge a certain piece if it does not satisfy the requirements in multiple ways. Of course, “working condition” does not mean that we actually see something but, as the software gets more complex, we are able to use those aggregates to play the role in the operations they are designed for. We may, for example, create a quick prototype that replays some interactions specified by the business team. This of course goes hand in hand with testing and feature acceptance tests or behavior-driven development.

Presenting an intermediate state to the domain experts needs to involve guidance on the feature, as well as asking questions along the way. Throwing partially implemented software over the fence is hardly of any use.
Chapter 2

Drawing with code – spiking

After we now have an idea where to start developing, we can finally explore how to actually do it. When we think about the problem, we may have some idea how it works, but there are also going to be pieces where, though we know how the high level operates, we are unclear about the lower levels. When a developer does not know how something will work in reality, the best way to figure out what to do is by actually trying it and exploring the libraries and tools that are deemed useful along the way.

This approach is referred to as spiking. We create a throwaway code, which is just there to explore a certain difficult part, without the intention of having this code ever make it into production. This frees us from a lot of the intricacies that are often involved in creating production-ready code. The code is there just to solve a specific case and help us gain knowledge about how to solve the same problem later. Most developers know that the first approach is hardly ever the perfect solution to a problem, so let’s just deal with this fact by creating a first version that we intend to throw away. A spike is all about the knowledge gain and not about the code, so be ready to write something quick and dirty just to make it work. This can actually be a really fun exercise, by the way!

A common area to spike is the interaction with external services, such as our notification service where the interface is known on a high level but the developer has actually never worked with it. Since we don’t have any idea how to interface with a Raven, I’m going to leave this for now. We will need to revisit this when the situation comes up but, as we learned from our UML diagram, the process is asynchronous anyway. Therefore, we don’t expect that a response can hide behind a Mock in our first prototype.

The other interesting problem is going to be creating the interface between our code and the users. We can’t be sure how the users want to interact with the software, since there is no experience in using anything like it. Interfaces tend to give us more insight into what the users want out of the software as well. The way a user wants to use a software teaches a lot about his focus and expected features, so spiking it is a great way to learn more about the system. This spike can be done in multiple ways, but it is actually useful to have real interface elements to build on and that can later be filled with more interactions. One way of doing this is to create the interface in HTML, which provides the basic structure but without the interactivity and fill in the gaps with JavaScript as we move along. For the sake of brevity, the code is omitted. If you are interested, visit the code repository for the book and check it out.

Keep in mind that this is not actually an interface we intend to keep, but we can now show something to our users and explain how they will interact.
Getting started, it's about time

With the previous work done, we are now at a point where we can start to work on the first feature of the application. We explored our ideas as far as we needed, and are in a position where we can actually talk about the details with our domain experts. I simplified the steps to get here a bit in the sense that we only talked about one iteration. In reality, this process most likely takes several iterations where your own understanding of the problem evolves. Sometimes not only your understanding changes, but often the business side also refines its own understanding along the way as well.

Creating value without creating code

As programmers, we often feel that the value we create is tied to the code we create, but this is not true, I will even go as far as to say that our value lies in the code we don't create. The simpler we can make a problem, the easier it is to move the project forward, and simplicity is based on a solid understanding of working together with the business as a team.

There is nothing easier to create than complexity, so watch out!
Solving a problem in the simplest way possible is what every piece of software should aim to do.

When we walk through the process as we have done earlier and let people explain what they do each day, it is not rare to discover how something can be simplified and improved. It is part of the software development process to try to improve the process itself as well. As we explore the idea behind a feature, and let the business side talk about its own actions, it is common that they themselves notice unnecessary overhead or even process inherit complications that don't need to exist. This is the reason why we try to explore in the way we have done earlier in a textual format. Don't begrudge the time it takes to explore, but keep in mind that right now you are already creating value for the business, and an improvement at this stage is a great success.
Deciding on the first feature

Even though we have already been moving the business forward, it is now time to actually do what developers do best—write code. The exploration we performed now points us towards starting out with the following feature set.

We want to automate the process of moving prisoners out of our dungeon and keep a record of the prisoners moved at the same time. This seems really valuable, since an overflowing dungeon is a major problem for the orc master. This is also a part of the larger problem of keeping a record of the prisoners inside the dungeon and that we saw as part of our outline. Ultimately, this is what we set out to do. After this first feature is done, the moving of prisoners will be almost completely automated and will therefore save time we can invest into other elements of running the dungeon.

We flushed out a basic interface to handle this, and it seems to be great to work with. So, let’s start coding and set up a project using the techniques of domain-driven design to move the project along.

Summary

In this chapter, we learned how we can get started with a project prior to writing code. We focused on the interaction with the business experts, providing them with feedback by illustrating our thinking. We covered the importance of gathering knowledge and how to organize that knowledge so we can leverage it later on in the project to understand the goals of the application we are building.

As we moved forward, we looked into how to identify the core feature set and choose a good starting point to not only provide value to the business early on, but also to help us further understand the business domain. This process is similar to the goals of agile methodologies, trying to cover the core problems early on and provide quick value and feedback to the business.

In the next chapter, we are going to get started setting up the project and covering the important details to get a good grip on managing the process throughout development.
Where to buy this book

You can buy JavaScript Domain-Driven Design from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.