1

What Is Domain-Driven Design

Software development is most often applied to automating processes that exist in the real world, or providing solutions to real business problems; The business processes being automated or real world problems that the software is the domain of the software. We must understand from the beginning that software is originated from and deeply related to this domain.

Software is made up of code. We might be tempted to spend too much time with the code, and view the software as simply objects and methods.

Consider car manufacturing as a metaphor. The workers involved in auto manufacturing may specialize in producing parts of the car, but in doing so they often have a limited view of the entire car manufacturing process. They start viewing the car as a huge collection of parts which need to fit together, but a car is much more than that. A good car starts with a vision. It starts with carefully written specifications. And it continues with design. Lots and lots of design. Months, maybe years of time spent on design, changing and refining it until it reaches perfection, until it reflects the original vision. The processing design is not all on paper. Much of it includes doing models of the car, and testing them under certain conditions to see if they work. The design is modified based on the testing results. The car is sent to production eventually, and the parts are created and assembled together.

Software development is similar. We can't just sit down and type code. We can do that, and it works well for trivial cases . But we cannot create complex software like that.

In order to create good software, you have to know what that software is all about. You cannot create a banking software system unless you have a good understanding of what banking is all about, one must understand the *domain* of banking.

Is it possible to create complex banking software without good domain knowledge? No way. Never. Who knows banking? The software architect? No. He just uses the bank to keep his money safe and available when he needs them. The software analyst? Not really. He knows to analyze a given topic, when he is given all the necessary ingredients. The developer? Forget it. Who then? The bankers, of course. The banking system is very well understood by the people inside, by their specialists. They know all the details, all the catches, all the possible issues, all the rules. This is where we should always start: the domain.

When we begin a software project, we should focus on the domain it is operating in. The entire purpose of the software is to enhance a specific domain. To be able to do that, the software has to fit harmoniously with the domain it has been created for. Otherwise it will introduce strain into the domain, provoking malfunction, damage, and even wreak chaos.

How can we make the software fit harmoniously with the domain? The best way to do it is to make software a reflection of the domain. Software needs to incorporate the core concepts and elements of the domain, and to precisely realize the relationships between them. Software has to model the domain.

Somebody without knowledge of banking should be able to learn a lot just by reading the code in a domain model. This is essential. Software which does not have its roots planted deeply into the domain will not react well to change over time.

So we start with the domain. Then what? A domain is something of this world. It cannot just be taken and poured over the keyboard into the computer to become code. We need to create an abstraction of the domain. We learn a lot about a domain while talking with the domain experts. But this raw knowledge is not going to be easily transformed into software constructs, unless we build an abstraction of it, a blueprint in our minds. In the beginning, the blueprint is always incomplete. But in time, while working on it, we make it better, and it becomes more and more clear to us. What is this abstraction? It is a model, a model of the domain. According to Eric Evans, a domain model is not a particular diagram; it is the idea that the diagram is intended to convey. It is not just the knowledge in a domain expert's head; it is a rigorously organized and selective abstraction of that knowledge. A diagram can represent and communicate a model, as can carefully written code, as can an English sentence.

The model is our internal representation of the target domain, and it is very necessary throughout the design and the development process. During the design process we remember and make lots of references to the model. The world around us is way too much for our heads to handle. Even a specific domain could be more than a human mind can handle at one time. We need to organize information, to systematize it, to divide it up in smaller pieces, to group those pieces into logical modules, and take one at a time and deal with it. We even need to leave some parts of the domain out. A domain contains just too much information to include it all into the model. And much of it is not even necessary to be considered. This is a challenge by itself. What to keep and what to throw away? It's part of the design, the software creation process. The banking software will surely keep track of the customer's address, but it should not care about the customer's eye color. That is an obvious case, but other examples might not be so obvious.

A model is an essential part of software design. We need it in order to be able to deal with complexity. All our thinking process about the domain is synthesized into this model. That's good, but it has to come out of our head. It is not very useful if it remains in there, is it? We need to communicate this model with domain experts, with fellow designers, and with developers. The model is the essence of the software, but we need to create ways to express it, to communicate it with others. We are not alone in this process, so we need to share knowledge and information, and we need to do it well, precisely, completely, and without ambiguity. There are different ways to do that. One is graphical: diagrams, use cases, drawings, pictures, etc. Another is writing. We write down our vision about the domain. Another is language. We can and we should create a language to communicate specific issues about the domain. We will detail all these later, but the main point is that *we need to communicate the model.*

When we have a model expressed, we can start doing code design. This is different from software design. Software design is like creating the architecture of a house, it's about the big picture. On the other hand, code design is working on the details, like the location of a painting on a certain wall. Code design is also very important, but not as fundamental as software design. A code design mistake is usually more easily corrected, while software design errors are a lot more costly to repair. It's one thing to move a painting more to the left, and a completely different thing to tear down one side of the house in order to do it differently. Nonetheless the final product won't be good without good code design. Here code design patterns come handy, and they should be applied when necessary. Good coding techniques help to create clean, maintainable code.

There are different approaches to software design. One is the waterfall design method. This method involves a number of stages. The business experts put up a set of requirements which are communicated to the business analysts. The analysts create a model based on those requirements, and pass the results to the developers, who start coding based on what they have received. It's a one way flow of knowledge. While this has been a traditional approach in software design, and has been used with a certain level of success over the years, it has its flaws and limits. The main problem is that there is no feedback from the analysts to the business experts or from the developers to the analysts.

Another approach is the Agile methodologies, such as Extreme Programming (XP). These methodologies are a collective movement against the waterfall approach, resulting from the difficulties of trying to come up with all the requirements upfront, particularly in light of requirements change. It's really hard to create a complete model which covers all aspects of a domain upfront. It takes a lot of thinking, and often you just cannot see all the issues involved from the beginning, nor can you foresee some of the negative side effects or mistakes of your design. Another problem Agile attempts to solve is the so called "analysis paralysis", with team members so afraid of making any design decisions that they make no progress at all. While Agile advocates recognize the importance of design decision, they resist upfront design. Instead they employ a great deal of implementation flexibility, and through iterative development with continuous business stakeholder participation and a lot of refactoring, the development team gets to learn more about the customer domain and can better produce software that meets the customers needs.

The Agile methods have their own problems and limitations; they advocate simplicity, but everybody has their own view of what that means. Also, continuous refactoring done by developers without solid design principles will produce code that is hard to understand or change. And while the waterfall approach may lead to over-engineering, the fear of over-engineering may lead to another fear: the fear of doing a deep, thoroughly thought out design.

This book presents the principles of domain driven design, which when applied can greatly increase any development process' ability to model and implement the complex problems in the domain in a maintainable way. Domain Driven Design combines design and development practice, and shows how design and development can work together to create a better solution. Good design will accelerate the development, while feedback coming from the development process will enhance the design.

Building Domain Knowledge

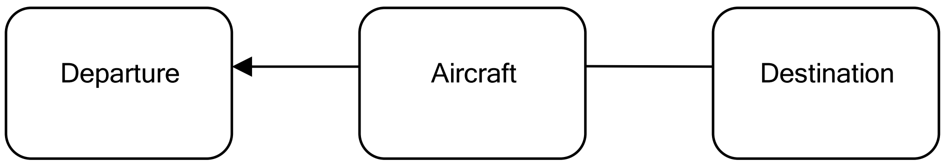
Let’s consider the example of an airplane flight control system project,and how domain knowledge can be built.

Thousands of planes are in the air at a given moment all over the planet. They are flying their own paths towards their destinations, and it is quite important to make sure they do not collide in the air. We won’t try to elaborate on the entire traffic control system, but on a smaller subset which is a flight monitoring system. The proposed project is a monitoring system which tracks every flight over a certain area, determines if the flight follows its supposed route or not, and if there is the possibility of a collision.

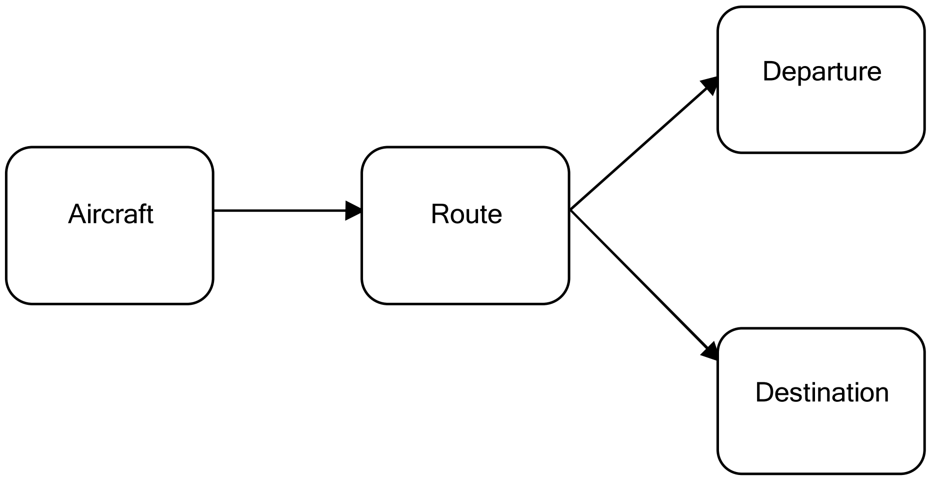
Where do we start from a software development perspective? In the previous section we said that we should start by understanding the domain, which in this case is air traffic monitoring. Air traffic controllers are the specialists of this domain. But the controllers are not system designers or software specialists. You can’t expect them to hand you a complete description of their problem domain.

The air traffic controllers have vast knowledge about their domain, but in order to be able to build up a model you need to extract essential information and generalize it. When you start talking to them, you will hear a lot about aircrafts taking off, and landing, aircrafts in midair and the danger of collision, planes waiting before being allowed to land, etc. To find order in this seemingly chaotic amount of information, we need to start somewhere.

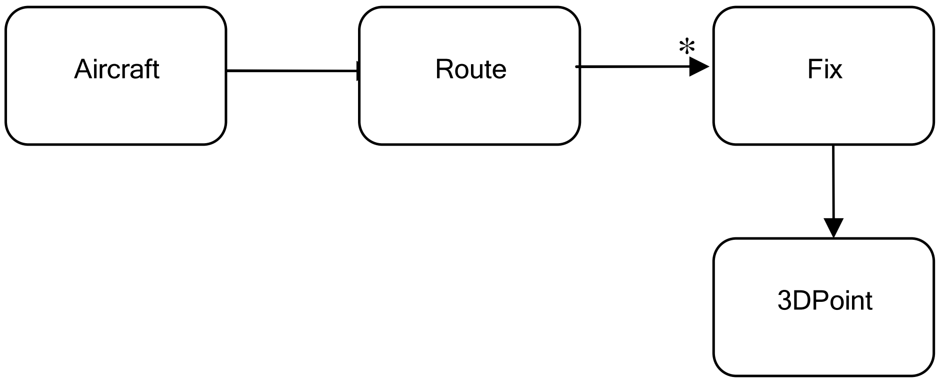
The controller and you agree that each aircraft has a departure and a destination airfield. So we have an aircraft, a departure and a destination, as shown in the figure below.



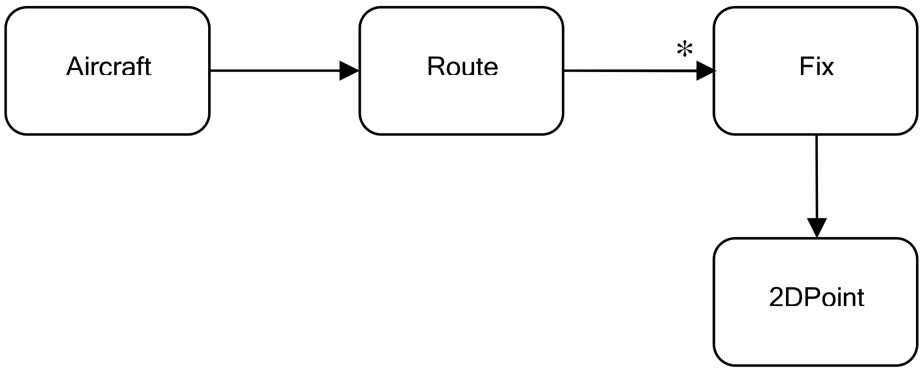
OK, the plane takes off from some place and touches down in another. But what happens in the air? What path of flight does it go? Actually we are more interested in what happens while it is airborn. The controller says that each plane is assigned a flight plan which is supposed to describe the entire air travel. While hearing about a flight plan, you may think in your mind that this is about the path followed by the plane while in the air. After further discussion, you hear an interesting word: route. It instantly catches your attention, and for a good reason. The route contains an important concept of flight travel. That’s what planes do while flying, they follow a route. It is obvious that the departure and destination points of the aircraft are also the starting and ending points of the route. So, instead of associating the aircraft with the departure and destination points, it seems more natural to associate it with a route, which in turn is associated with the corresponding departure and destination.



Talking with the controller about the routes airplanes follow, you discover that actually the route is made up of small segments, which put together constitute some sort of a crooked line from departure to destination. The line is supposed to pass through predetermined fixed points. So, a route can be considered as a series of consecutive fixes. At this point you no longer see the departure and destination as the terminal points of the route, but just another two of those fixes. This is probably quite different from how the controller sees them, but it is a necessary abstraction which helps later. The resulting changes based on these discoveries are:



The diagram shows another element, the fact that each fix is a point in space followed by the route, and it is expressed as a three dimensional point. But when you talk to the controller, you will discover that he does not see it that way. Actually he sees the route as the projection on earth of the plane flight. The fixes are just points on Earth surface uniquely determined by their latitude and longitude. So the correct diagram is:



What is actually happening here? You and the domain experts are talking, you are exchanging knowledge. You start asking questions, and they respond. While they do that, they dig essential concepts out of the air traffic domain. Those concepts may come out unpolished and disorganized, but nonetheless they are essential for understanding the domain. You need to learn as much as possible about the domain from the experts. And by putting the right questions, and processing the information in the right way, you and the experts will start to sketch a view of the domain, a domain model. This view is neither complete nor correct, but it is the start you need. Try to figure out the essential concepts of the domain.

This is an important part of the design. Usually there are long discussions between software architects or developers and the domain experts. The software specialists want to extract knowledge from the domain experts, and they also have to transform it into a useful form. At some point, they might want to create an early prototype to see how it works so far. While doing that they may find some issues with their model, or their approach, and may want to change the model. The communication is not only one way, from the domain experts to the software architect and further to the developers. There is also feedback, which helps create a better model, and a clearer and more correct understanding of the domain. Domain experts know their area of expertise well, but they organize and use their knowledge in a specific way, which is not always the best to be implemented into a software system. The analytical mind of the software designer helps unearth some of the key concepts of the domain during discussions with domain experts, and also help construct a structure for future discussions as we will see in the next chapter. We, the software specialists (software architects and developers) and the domain experts, are creating the model of the domain together, and the model is the place where those two areas of expertise meet. This might seem like a very time consuming process, and it is, but this is how it should be, because in the end the software's purpose is to solve business problems in a real life domain, so it has to blend perfectly with the domain.