JOSÉ ANTONIO FUENTES SANTIAGO

DOMAiN-DRIVEN DESiGN IN A ZOMBIE APOCALYPSE!



Two Players Software Development Book

Domain-Driven Design in a Zombie Apocalypse (Demo)

José Antonio Fuentes Santiago

Demo version

Block 1. Survival strategy (strategic design)



Figure 1: The town was taken by the horde…

Now…

An explosion in the middle of the night wakes you up. You hear screams in the street and go to the window. Little by little, you watch chaos take over your town.

The hours drag by while you decide what to do. Someone pounds on your door. You look through the peephole and recognize your friend **Santi**. You open the door as fast as you can. As soon as he's inside, he collapses onto the floor, exhausted. His face is pale, his clothes are torn, and he has a shallow cut on his arm. Once he gets his breath back, he tells you what's going on outside.

He says you both have to leave town and find a safe shelter. Apparently, the military

has set up a shelter a few kilometers away. You pack two backpacks with everything you can carry, mostly food and medicine.

You step out into the street, keeping close to the walls, all your senses on alert. The smell reaches you before you can see the horror. The chaos is total. There are bloodied bodies everywhere, some motionless, others feeding on them. Getting out of there unseen is going to be hard.

You think about taking your car, but a dozen vehicles block one side of the street and an overturned truck blocks the other. So you have no choice but to walk in silence, dodging those things. Luckily, while feeding, they seem oblivious to everything around them.

After a day on the road, several injuries, and a few hasty escapes, you make it to the shelter. **Nora** is the commander who runs it. Several teams work under her: Medical, Logistics, and Command.

They take you to a room for a thorough examination to make sure neither of you has been bitten. After that, they welcome you to what will be your new home… for now.

In this block…

You'll see how to become part of the shelter's team of survivors and learn to speak the same language they do. You'll also learn what each department does and how those responsibilities are kept clearly separated.

- **Chapter 1. Ubiquitous language**: Learn to use the same words in the code and in team conversations to avoid misunderstandings.
- **Chapter 2. Bounded contexts**: Discover how to divide a system along boundaries where each model has a clear, consistent meaning.
- **Chapter 3. Context map**: Analyze how the Contexts communicate, who has authority over whom, and what kinds of relationships exist.
- **Block 1 summary**: Review the main definitions of Ubiquitous Language, Bounded Contexts, and Context Map.

Chapter 1. Ubiquitous language

After a short interview, Nora explains that they lost the members of the shelter's IT team to a failure in the security protocols. Just then, there's a sharp knock at the door of the room. **Joel**, the head of Medical, walks in with the look of someone bringing bad news.

"Nora, we've got someone with a fever in the west wing. I asked for him to be taken to isolation, but the Command team says they haven't received orders from you."

Nora exhales slowly.

"Joel, we've been over this before. If someone has any symptom consistent with infection, they go straight to quarantine, without waiting for my confirmation. That's a fixed rule."
"I know, but they say 'quarantine' and 'preventive isolation' aren't the same thing."
"They're wrong. It's exactly the same thing." She turns toward you before leaving and, after a sigh, adds, "If we want to survive here, we're going to have to avoid these misunderstandings." She points at the terminals of the system that runs the center. "Follow me! I'll explain how all this works, and then we'll see what you're capable of."

What just happened between Joel and the Command team (two different words for the same rule) also happens in the code that describes the shelter. In DDD, everything the software tries to model (who is infected, what supplies are left, which routes are safe) is called the **Domain**. And when the code stops speaking in the same terms as the Domain, it turns into a riddle.

The ubiquitous language

The first pillar of DDD is establishing a *common language* among everyone involved in a project. In DDD, that's called the **Ubiquitous Language**: the same words Nora uses to describe how the shelter works also show up in the code and in the team's conversations. That saves you the constant mental translation between what the business says and what you've coded.

When you use the Ubiquitous Language, you stop using names like `User` or `Person` and pick something more fitting, such as `Survivor`. And you don't define `setHealth()`. You create methods like `takeInjury()` or `applyBandage()` instead. Thanks to that, nobody needs to look at the documentation to understand what each thing does.



Figure 2: Nora and Joel agree to start speaking a common language.

When the code doesn't speak the same language

Once Nora finishes explaining how the shelter works, Santi sits down at the coordination room terminal and starts going through the system the previous team left behind. The shelter has been running for days, but until now nobody had time to put the code in order: the first moments of the apocalypse were all improvisation. A few minutes later, Santi calls you over.

"Hey, look at this," Santi says, pointing at the screen. "`route.level = 0`. What's that supposed to mean?"
"No idea," you answer, coming closer. "The safest one? The shortest? The only one available?"
"And here, `survivor.status = 2`. Infected? In quarantine? Both?"
"Looks like there's no way to know without tracing through the whole code," you reply. "And whoever wrote it doesn't seem to be around to tell the tale."

The direct consequence of not using the Ubiquitous Language is the **loss of Domain knowledge**. The code ends up as a collection of magic numbers and odd names that only the person who wrote them can make sense of. With luck, because sometimes even that person doesn't remember two months later, or has moved on to a better place after a zombie attack.

That's exactly what happened here. Every time Nora talked about an "emergency

resupply," the previous team wrote `supply.priority = 1`. When she said "safe route," it showed up in the code as `route.level = 0`. That team never wrote down their mental dictionary, and now it's gone with them.

From the language to the code

Domain knowledge has to be written down. You don't need a hundred-page document. A simple glossary with the main terms is enough.

To build the glossary, record meetings with the business team and write down every technical term they use. Then confirm your understanding with them. That avoids misunderstandings from the start. In chapter 14 we'll look at this process in more detail.

🪓 Survivor's tip

The glossary isn't something you write once and file away. The Domain keeps evolving: what everyone calls "quarantine" today may be "preventive isolation" tomorrow. When a term changes, the glossary and the code have to move together. Otherwise you're right back where you started, with two words for the same thing. Treat a change in the language with the same discipline as a change in the code.

Santi has spent a few minutes tracking down where the code on his screen came from:

```
route.level = 0;
```

After some digging and a few tests, he refactors the code to use the Ubiquitous Language.

```
route.markAsSafe();
```

With `markAsSafe()` there's no need to ask what happens to the route. What's more, if marking a route as safe meant notifying the patrol team or updating the map, you'd put that logic inside the method, so the callers don't have to change.

💡 Hint

If the business team uses terms that sound odd or technical, don't translate them on autopilot. Ask them what they mean exactly. A single word like "supply" can mean very different things depending on who's saying it. As Inigo Montoya would say: "I do not think it means what you think it means."

Chapter 2. Bounded contexts

After a few days in the shelter, Nora gives Santi and you the job of watching how each department works. You start in the coordination room, where **Octavia** runs Command operations.

A few minutes in, you hear an argument in the hallway. **Andreu**, the head of Logistics, is talking to **Joel**, and he doesn't look friendly.

"I need to know whether Pedro can make the trip to the northern supply point," he says, dropping a folder on the table. "I've been waiting for the report from Medical all morning."
"Pedro is still in quarantine," Joel answers. "I don't care whether he can walk for three hours. What I care about is whether that fever of his means he's caught the zombie virus. If he has, he could put the whole shelter in danger."

Andreu only wants to know whether Pedro can get to the north without collapsing. Joel, on the other hand, wants to know whether half the shelter will be infected in two days. They're both talking about Pedro, but they're trying to solve different problems. It might seem that a single record for Pedro would be the cleanest option. But the data Andreu needs and the data Joel needs have nothing in common.

What is a bounded context?

In large systems, the same word can mean completely different things depending on who uses it. That's where the **Bounded Context** comes in: a part of the project with clear boundaries where each term has a single meaning. Like each department in the shelter, what's said inside it follows its own rules, and outside it, those same words can mean something else.

In the shelter, the word **"Survivor"** doesn't mean the same thing to everyone.

- For **Logistics**, a survivor is someone who consumes resources, has a location, and can carry supplies. They care about stamina and load capacity.
- For **Medical**, a survivor is someone with a medical history, a health level, and an infection risk. They care about body temperature and whether the person has been exposed to bites.
- For **Command**, a survivor is a unit with specific skills that can be assigned to shelters or missions. They care about combat experience and technical knowledge.

If you try to cram all of that into a single `Survivor` model, you'll end up with a gigantic class with dozens of properties, half of which make no sense in any given Context. Worse still, when Logistics wants to change how load capacity is calculated, they could accidentally break something in Medical.

With all that in mind, you sit down at the coordination room terminal and write out both approaches, so the difference is clear:

```
// The "Monster Survivor" that knows everything and must be avoided.
interface Survivor {
  id: string;
  name: string;
  health: number;       // Medical
  vaccines: string[];   // Medical
  maxWeight: number;    // Logistics
  location: Coordinates;// Logistics
  rank: string;         // Command
}

// It's better to have specific models per Context.
// In the Medical Context:
interface MedicalSurvivor {
  id: string;
  infectionLevel: number;
  history: MedicalNote[];
}

// ...And in the Logistics one:
interface LogisticsSurvivor {
  id: string;
  loadCapacity: number;
  isAvailable: boolean;
}
```

One model per context

If you remember the previous chapter, each Context will also have its own Ubiquitous Language: each department models Pedro differently, and it also uses terms, actions, and concepts that don't exist in the others. Andreu's language and Joel's don't compete. They simply belong to different worlds.

The solution is to split the system into Bounded Contexts, each with its own model. An-

dreu talks about stamina and load capacity. Joel talks about medical history, symptoms, and infection risk. Octavia talks about skills, shifts, and assignments. Each department solves a different problem, and its `Survivor` has only the properties that matter to that problem.



Figure 3: The three Bounded Contexts of our apocalypse: Command, Medical, and Logistics.

> **Hint**
>
> Where does one Context end and the next begin? Ask yourself: "Does this concept change meaning in different parts of the business?" If "Survivor" means different things to Logistics and to Medical, there's your boundary. You probably need separate Contexts.

Andreu can change how routes are calculated without touching the medical history the Medical department uses. And Joel can add a new field to the medical history without Logistics ever hearing about it. Each team works on its own problem without having to coordinate every change with everyone else.

🪓 Survivor's tip

I know duplicating `MedicalSurvivor` and `LogisticsSurvivor` is going to sting. We were all taught that duplication is a sin. But sharing one Domain model across Contexts can cost you more in the long run, because every change in one team can hit the other. Duplicate thoughtfully, and don't lose any sleep over it.

Subdomains

While Octavia explains how Command is organized, you look at the map of the shelter pinned to the wall: patrol routes, risk zones, supply points. There are plenty of problems to solve here. But they don't all carry the same weight.

"What's the most important thing during a zombie attack?" Octavia asks, without taking her eyes off the map.

The question catches you off guard. You blink, as if coming back to reality, and after a moment's hesitation you answer:

"Defense?"
"Exactly. And for that, copying what another shelter does is no use to us. Every shelter has its own particular features, whether it sits on high ground, has water around it, and so on. Here we have this military building with a set of protected entrances. Nobody else can design our defense strategy for us." She pauses. "The next priority after defense is scouting. But any experienced team can follow more or less the same reconnaissance protocols. And finally, the night watches…" She points at a sheet of paper with a hand-drawn table pinned to the wall. "Before he got sick, Pedro handled those with a list and a pen. You don't need much more than that."

DDD calls those categories **Subdomains**, and they describe how the real world is divided before a single line of code is written.

- **Core Domain**: what makes this shelter unique and can't be copied from another one. The **Defense** strategy depends on the terrain, the resources, and the specific surroundings: defending a building on top of a mountain is nothing like defending one in a valley. Nobody else can hand you that solution ready-made. This is where the code should reflect the business rules most faithfully, and where it's worth applying DDD in depth.

- **Supporting Subdomain**: necessary, but not a differentiator. **Scouting** matters for survival, but it follows reconnaissance protocols that any organized group could follow. It deserves to be built well, without the same obsession as the Core Domain.
- **Generic Subdomain**: standard problems that everyone solves the same way. Managing the **Watches** (shifts, schedules, handovers) is neither strategic nor unique. Here it makes sense to use ready-made solutions: libraries, external services, or even a sheet of paper on the wall.

🪓 Survivor's tip

Put your biggest design effort into the Core Domain. Applying full DDD to a Generic Subdomain is like fitting an armored door on the bathroom: you can do it, but you're reinforcing a spot that doesn't need it. For a more compact view of Subdomains and Bounded Contexts, see bibliography `[3]`.



Figure 4: The three kinds of Subdomain: Core Domain (Defense), Supporting Subdomain (Scouting), and Generic Subdomain (Watches), with the size of each area representing the recommended design effort

Subdomains and bounded contexts aren't the same thing

You've just seen three Subdomains: Defense, Scouting, and Watches. But earlier, when we talked about Bounded Contexts, "Command" came up too. They could be the same

thing under different names, but they aren't. And understanding why goes a long way toward understanding DDD.

Octavia points at the map in the coordination room. It shows scouting routes, risk zones, and red circles marking the sectors occupied by hordes.

"This is the real shelter. The problem we have to solve." Then she turns toward the three computers stacked in the corner. "That's what we've put together to manage it."

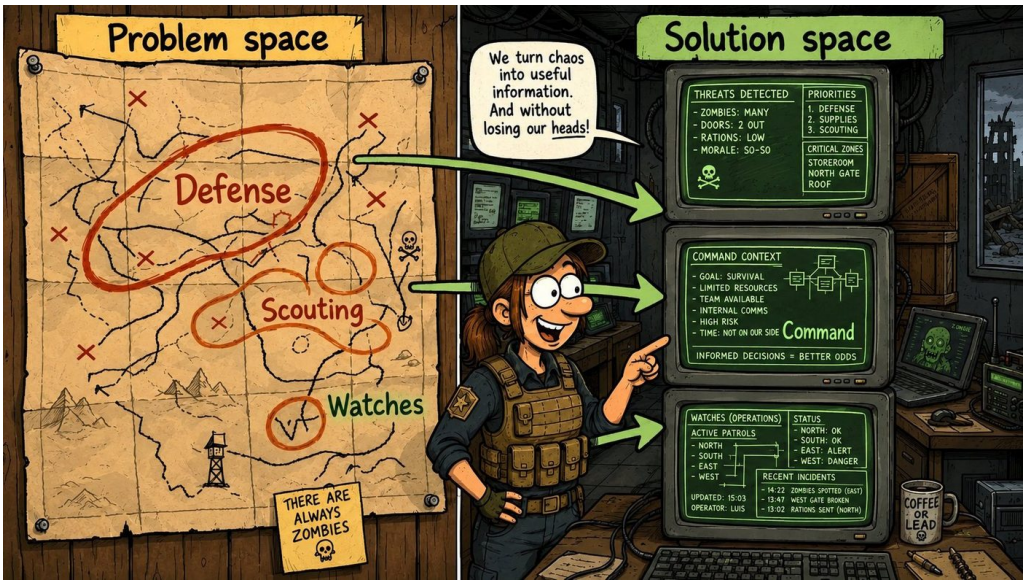


Figure 5: Comparison between the problem space (the Defense, Scouting, and Watches Subdomains on the map) and the solution space (a single Command Context that groups them)

Octavia's distinction helps you get it. The Subdomains are the map: they describe which problems exist in the real world and how strategically important each one is. Defense, Scouting, and Watches are three real problems that have to be solved before writing a line of code. The Bounded Contexts are the computers: the boundaries you define in the system so that each model has a clear meaning and doesn't get mixed up with the others.

They're two separate layers, with no hierarchy between them. A Subdomain describes a problem, while a Context describes how you decide to solve it in code. The relationship between the two follows no fixed pattern. Several small Subdomains can be grouped into a single Context. A very large Subdomain can be split into several Contexts. There can even be an exact one-to-one match. It's a design decision that depends on the team,

the resources, and the complexity of the problem.

"You see how Defense, Scouting, and Watches are all handled by the Command team?" Octavia goes on. "With the people we have, it makes no sense to set up three separate teams for problems so closely related. So Command ends up covering all three."

Knowing what is Core Domain and what isn't lets you decide where to put your focus, and where not to waste it.

But Contexts aren't islands. When Joel detects an outbreak, Octavia needs to know so she can order an evacuation. When Andreu updates the inventory, Command has to know whether there's a critical shortage. How do they communicate without breaking their boundaries or contaminating their models? Because if they get it wrong, they can doom the shelter.

To be continued...

This is where the demo ends. You've reached the shelter, you've learned to speak with the same words as the team defending it, and you've seen why a "survivor" doesn't mean the same thing to Joel as it does to Andreu. With that, you can already draw boundaries between models. What you don't know yet is how to cross them without breaking them: who has the upper hand when Health detects an outbreak and Command needs to know, and what kinds of relationships exist between Contexts. That's what you still have to learn before closing this first block: the context map.

The full book goes much further. In the second block the group takes a shopping mall and turns it into a shelter, and along the way you learn to tell Entities from Value Objects, to protect rules with Aggregates, and to place the logic that fits in neither. In the third, two shelters have to be coordinated at once, and that's where Repositories, the application layer, Domain Events, and Hexagonal Architecture come in. The fourth block is the hardest: the mall falls, it's time to evacuate to a prison and hold out against the final horde. For that you need Sagas, an eye for the antipatterns that sink a project, the nerve to test the Domain, and a way to discover the model with the team before writing code. At the end there are three reference appendices: when to use DDD and when not to, how it applies in other domains, and the questions that come up most often.

If you've enjoyed what you've read so far, you can get the full book:

- Paperback
- eBook

Check the Amazon stores of other countries to find it too.