DI


```go
func main() {
    me := Person{
        Name:      "Davide Imola",
        Role:      "Software Engineer",
        Company:   "RedCarbon SA",
        Community: "Schrödinger Hat",
        Location:  "Verona, Italy",
        Site:      "https://links.davideimola.dev",
        Interests: []string{
            "Kubernetes",
            "Go",
            "Open Source",
            "BBQ",
            "Cybersecurity",
        },
    }
}
```

Once upon a time…

…an epic journey began. A journey that would take us through the realms of Golang and Domain-Driven Design.

Our hero, the Gopher, during his adventures, one day stumbled upon a mysterious land called Domain-Driven Design.

What secrets would he uncover? What challenges would he face? And most importantly, what would he learn?

Join me (your's today Bard) on this epic quest as we explore the world of Gophers Gone Domain-Driven!

DI

The Domain-Driven Land

A land where the domain is law, and the code is its faithful servant.

Here, the Gopher learned that the key to building great software lies in understanding the domain.

By modeling the domain accurately, we can create software that reflects the real world, making it easier to reason about and maintain.

DI

The Fabled Boundaries

At the edge of the domain lie the Bounded Context, territories clearly marked to divide the land into distinct kingdoms.

Inventoryia

On the other side of the mountains lies the kingdom of Inventoryia. Here, they don’t care about who placed an order or when it was shipped. Instead, they’re focused on managing stock and supplies—making sure items are available, in the right quantities, and restocked as needed.

In Inventoryia, they also have a concept called “Order,” but it means something very different here! In Inventoryia, an “Order” refers to a purchase order—the items they need to bring in from suppliers to keep their inventory full. It has its own set of details: supplier names, quantities, costs, and delivery dates. And while they also have IDs, their “Order ID” refers specifically to supply orders rather than customer purchases.

DI

One Language, to Rule Them All

As any kingdom is free to use its own language, rules, and models that make sense within its borders. They must agree on a common language when they need to communicate.

This common language is called the Ubiquitous Language - a shared vocabulary that everyone in the kingdom understands.

From the dwarves in the mines (developers in their code) to the elves in the forests (business stakeholders in their meetings), everyone uses the same terms and concepts to ensure clear communication and understanding.

The language becomes part of the kingdom's "spellbooks"! It's used in code, documentation, and conversations, ensuring that everyone is on the same page and reducing the risk of misunderstandings.

DI

Mixing the Kingdoms

But what happens when the kingdoms need to work together?

When a visitor from Orderland tries to talk to someone from Inventoryia, they might find themselves speaking different when mentioning the word "Order ID".

Without understanding the cultural context, the Inventoryian may assume this Order ID refers to a purchase order for stocking goods, not a customer purchase.

As you might guess, this could lead to all sorts of chaos—incorrect items being shipped, double restocks, or even shortages, with the two kingdoms misunderstanding each other’s needs and goals.

DI

Boundaries and Translations

To keep peace and clarity, the leaders of Orderland and Inventoryia established a Bounded Context for each kingdom, agreeing that:

- Orderland's "Order" will only ever refer to customer purchases.
- Inventoryia's "Order" will only ever mean supply restocking.

When they need to communicate, they use translators (often known as context mapping in DDD). Translators help convert Orderland's "Order ID" for purchases into Inventoryia's "Supply Order ID" for restocking, so each kingdom can understand the other without confusion.

DI

The Onion Castle

In the heart of the Domain-Driven Land lies the Onion Castle.

The Onion Castle is a fortress of code, with layers upon layers of protection.

Each layer represents a different part of the domain, from the core entities at the center to the outermost services and interfaces.

By organizing the code in this way, we can create a clear separation of concerns and ensure that each part of the domain is properly encapsulated.

DI

The Onion Castle: Domain

At the core of the Onion Castle lies the Domain layer.

Here, we find the heart of the domain, where the core entities and value objects reside.

No business logic or external dependencies are allowed in this layer; it's all about the domain and nothing else.

```go
type Order struct {
    ID        string
    CreatedAt time.Time
    UpdatedAt time.Time
    Status    string
}

func (o *Order) Ship() {
    o.Status = "Shipped"
    o.UpdatedAt = time.Now()
}

func (o *Order) Cancel() {
    o.Status = "Cancelled"
    o.UpdateAt = time.Now()
}
```

DI

The Onion Castle: Application

Surrounding the Domain layer is the Application layer.

Here, we find the use cases and business logic that drive the application.

```go
type OrderService struct {
    orderRepo OrderRepository
}

func (s *OrderService) ShipOrder(id string) error {
    order, err := s.orderRepo.GetOrder(id)
    if err != nil {
        return err
    }

    order.Ship()
    return s.orderRepo.UpdateOrder(order)
}
```

DI

The Onion Castle: Infrastructure

At the outermost layer of the Onion Castle lies the Infrastructure layer.

Here, we find the code that interacts with external systems, such as databases, APIs, and services.

They are passed down to the Application layer, through dependency injection, to ensure that the domain remains pure and free from external concerns.

```
type OrderHTTPService struct {
    orderSvc OrderService
}

func (s *OrderHTTPService) Ship(r *http.Request) error {
    err := s.orderSvc.Ship(r.URL.Query().Get("id"))
    if err ≠ nil {
        return err
    }

    return nil
}
```

And now the magic begins! 🪄
Let’s adventure into the code! ☠️

Pros of DDD

Let’s take a look at some of the benefits of Domain-Driven Design:

- Clearer Communication: By using a common language, everyone can understand each other better.
- Better Modeling: By focusing on the domain, we can create more accurate models that reflect the real world.
- Easier Maintenance: With a clear domain model, it’s easier to maintain and update the codebase.
- Reduced Risk: By focusing on the domain, we can reduce the risk of misunderstandings and miscommunications.
- Scalability: DDD can help us build more scalable systems by focusing on the domain and its boundaries.

Cons of DDD

Let’s take a look at some of the challenges of Domain-Driven Design:

- Complexity: DDD can be complex, especially for large systems with many domains and bounded contexts.
- Learning Curve: DDD has a steep learning curve, and it can take time to fully understand and implement.
- Huge Effort: DDD requires a significant effort to implement correctly, and it may not be suitable for all projects.

A few suggestions from your friendly Bard

If you’re considering embarking on your own journey into the world of Domain-Driven Design, here are a few tips to help

- Start Small: Begin by identifying a single domain and creating a bounded context around it.
- Collaborate: Work closely with domain experts and stakeholders to ensure you’re modeling the domain accurately.
- Iterate: Don’t try to model the entire domain at once. Start with a small part and iterate as you learn more!! All the kingdoms were not built in a day.
- Learn: Take the time to learn about DDD and its concepts. There are many great resources available to help you get started.

Thank you for joining me today on this epic quest!