Welcome

\*\*\*

**Inspired by:**

**Many presentations** about microservices are about **framework** X or Y, but the question that kept bugging me was **how to split** a problem into pieces. **Last year**, I attended a workshop by **Erik and Koen**, which seemed to **promise** to answer that question. But apart from a mention of the single responsibility principle **in the title**, there was no more explanation about how the actual splitting should occur. I also **lacked** the assumed **knowledge** of creating a microservice from scratch. And so I came up with the idea of **complementing their idea** with a bit more **theory** and **process**, the use of a **simpler game**, and a **hello-world microservice** for people like me who don't know how to conjure one out of a hat.

\*\*\*

**Possible criteria**

monolith first is often easier, as it saves a lot of refactoring as the model still has to crystalize

\*\*\*

**DDD terms explained**

* domain  
  The **subject area** for which the user wants a piece of software
* model  
  A **system of abstractions** that describes selected aspects of a domain and can be **used to solve problems** related to that domain.
* ubiquitous language  
  A **language** structured around the domain model and **used by all** team members to **connect** all the activities of the team **with** the **software**.
* context  
  The **setting** in which a word or statement has meaning. **Statements** about a model can only be understood in a context.
* bounded context  
  A **description of** a **boundary** (typically a subsystem, or the work of a particular team) within which a particular model is defined and applicable.

\*\*\*

**Building blocks**

* Often **business logic gets spread** throughout all layers of the code, meaning a change to the UI can accidentally change the business logic. Therefor you should **isolate** the expression of the **domain model and the business logic**, and eliminate any dependency on infrastructure and user interface.
* Many objects represent a thread of **continuity and identity**, going through a **lifecycle**, though their **attributes may change**. If that is the case, make sure this is reflected in the model by **keeping** the class **definition simple** and focused on identity and lifecycle.  
  **Example: user**
* There are also elements in the model where you care more about the **attributes and logic** than the identity. These are the value objects. Their **values** are **immutable**, and their **operations** should be **side-effect-free**.  
  **Example: address**
* Sometimes you care more about the **activities that** have **occured** than the current state of the system. In this case you should model information about these activities in the domain as a **series of discrete events**. Typically these events will be immutable and contain a timestamp.  
  **Example: bank transactions**
* Some concepts from your domain **aren't natural to model as objects**. In such a case it might make more sense to add an operation to the model as a **standalone interface declared as a service**.  
  **Example: InvoiceNumberGenerator**

\*\*\*

**Large scale structure**

* A large-scale structure is a **language** that lets you **discuss and understand the system in broad strokes**. A **set** of **high-level concepts or rules**, or both, establishes a pattern of design for an entire system. However, an **ill-fitting structure is worse than none**, so try to find a minimal set that solves the problems that have emerged.
* As a software **design** tends to be **very abstract** and hard to grasp, it is a good idea to find a **tangible way to understand the system**. So if there is a **concrete analogy** that captures the imagination of the team, adopt it as a large-scale structure. But all metaphors are inexact, so **keep reexamining** if it is still a good fit, and drop it, if it gets in the way.
* We've all been taught that **each class** should have **only one responsibility**. But responsibility-driven design **also applies to larger scales**. When you look at the conceptual dependencies in your model and the varying rates and sources of change of the different parts of your domain, you might be able to **identify natural strata**. These should tell a story of the **high-level purpose and design** of your system.
* In an application in which the roles and relationships between entities **vary in different situations**, complexity can explode. Neither fully general models nor highly customized ones serve the users' needs. In such cases it is useful to create a **distinct set of objects** that can be used to describe and contain the structure and behavior of the basic model. Keep these concerns separate as two levels, one very concrete, the other reflecting the rules and knowledge that a user or super-user is able to customize.

\*\*\*

**References:**

The **first two** books are definitely **recommended** reading. The **third** is more **aimed at managers** and other people who look more at the high-level overview. However it does contain a great chapter about **whether** microservices **will fit** your problem or not. The **fourth** book was a **free download** and gives you a nice boost in **how to program** microservices with three different **frameworks** (Spring Boot, Dropwizard and WildFly Swarm).