Thursday, 2 September, 2021

10:31 PM

**Architecture and Design**

Designing is fundamentally about taking things apart... in such a way that they can be put back together. ...Separating things into things that can be composed that's what design is.

— Rich Hickey. [**Design Composition and Performance**](https://www.infoq.com/presentations/Design-Composition-Performance/)

System design, says the quote in the epigraph, is the system separation so that it can be reassembled later. And most importantly, be assembled easily, without too much work.

|  |  |  |
| --- | --- | --- |
| **class My-Class extends My-Interface {**    **Function(var1, var2) { }**  **var1 function1() { }**    **}** |  | **Interface My-Interface {**    **void function(var1, var2);**  **var1 function1();**    **}** |

**Domain Layer**

The **data structure** of **domain entities** and the essence of their transformations are independent from the outer world. External events *trigger domain transformations*, but *do not determine* how they will occur.

**Application Layer**

This layer describes [**use cases**](https://en.wikipedia.org/wiki/Use_case), i.e. user scenarios. They are responsible for what happens after some **event** occurs.

Also, in the application layer there are [**ports**](https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together/#ports)—the specifications of how *our application wants* the outside world to communicate with it. Usually a port is an [**interface**](https://en.wikipedia.org/wiki/Interface_(computing)#Software_interfaces), a behavior contract.

**Ports** serve as a **“buffer zone”** between our application's wishes and the reality. **Input Ports** tell us how the application *wants* to be contacted by the outside world. **Output Ports** say how the application *is going to* communicate with the outside world to make it ready.

**Adapters Layer**

The outermost layer contains the [**adapters**](https://herbertograca.com/2017/09/14/ports-adapters-architecture/) to external services. Adapters are needed to turn incompatible APIs of external services into those compatible with our application's wishes.

Adapters are a great way to lower the[**coupling**](https://en.wikipedia.org/wiki/Coupling_(computer_programming)) between our code and the code of third-party services. Low coupling reduces needs to change one module when others are changed.

**Adapters are often divided into:**

* **driving—which *send signals to* our application;**
* **driven—which *receive the signals from* our application.**

*The user* interacts most often with driving adapters. For example, the UI framework's handling of a button click is the work of a driving adapter. It works with the browser API (basically a third-party service) and converts the event into a signal that our application can understand.

Driven adapters interact with the ***infrastructure***. In the **frontend**, most of the **infrastructure** is the **backend server**, but sometimes we may interact with some other services directly, such as a **search engine**.

Note that the farther we are from the **center**, the more **“service-oriented”** the code functionality is, the *farther it is from the* ***domain knowledge*** of our application. This will be important later on, when we decide which layer any **module** should belong to.

**Costs of Clean Architecture**

1. **Takes Time**
2. **Sometimes Overly Verbose (** In general, a **canonical** کَلیسائی قَواعِد و ضَوابِط سے مُتَعلِق **implementation** of the clean architecture is not always convenient, and sometimes even harmful. If the project is small, a full implementation will be an overkill that will increase the entry **threshold** حَدِ حِس , حَدِ تاثَر for newcomers. You may need to make design tradeoffs to stay within budget or deadline. I'll show you by example exactly what I mean by such tradeoffs.)
3. **Can Make Onboarding More Difficult**
4. **Can Increase the Amount of Code (**maybe access the domain functionality directly from the adapter, bypassing the use case**)**

|  |  |
| --- | --- |
| **Designing Application Layer**  The application layer contains the use cases. A **use case** always has an **actor**, an **action**, and a **result**. Also, in the application layer there are **ports**—**interfaces** for communicating with the outside world. | **Designing Adapters Layer**  In the adapters layer, we declare **adapters** to **external services**. Adapters make incompatible APIs of third-party services compatible to our system. |
|  |  |

**Validate Data Flow Diagram**

|  |  |  |
| --- | --- | --- |
| Let's now validate how the user will communicate with the application during the created use case. |  | The user interacts with the UI layer, which can only access the application through **ports(interface)**. That is, we can change the **UI** if we want to. |

**Split Code by Features, not Layers**

|  |  |  |
| --- | --- | --- |
| The code can be split in **folders** |  | not “by **layers**” but “by **features**”. One feature would be a piece of the pie from the **schematic** تَصُّوری below. This structure is even more preferable, because it allows |

you to deploy certain features separately, which is often useful.

|  |  |
| --- | --- |
|  | **Conclusions**  It's not a gold standard, but rather a compilation تَرتیب of experience with different **projects**, **paradigms**, and **languages**. I find it a convenient **scheme** (چال , منصوبہ , تجویز ) that allows you to decouple code and make independent **layers**, **modules**, **services**, which not only can be deployed and published separately, but also transferred from project to project if needed. |

**<============================DDD, Hexagonal, Onion, Clean, CQRS, … How I put it all together============================>**

|  |  |
| --- | --- |
|  | [**Ports & Adapters**](https://herbertograca.com/2017/09/14/ports-adapters-architecture/) architecture explicitly identifies **three** fundamental blocks of code in a system:   * 1. What makes it possible to run a **user interface**, whatever type of user interface it might be;   2. The system **business logic**, or **application core**, which is used by the user interface to actually make things happen;   3. **Infrastructure** code, that connects our application core to tools like a database, a search engine or 3rd party APIs.   (Data Access, I/O, and Web Services are all infrastructure. **Infrastructure** is any code that is a **commodity and does not give your application a competitive advantage**. This code is most likely to change frequently as the application goes through years of maintenance.)    Data access changes every two years or so, so we definitely don’t want to be tightly coupled to it.  For long-life, we would want our business logic to be independent of these infrastructure concerns so that as infrastructure changes, the business logic doesn’t have to. As a result we can integrate latest infrastructure code with our application when it's available in the industry.    The **application core** is what we should really care about. It is the code that allows our code to do what it is supposed to do, it is our application. It might use several user **interfaces** (**progressive web app, mobile, CLI, API, …**) but the code actually doing the work is the same and is located in the application core, it shouldn’t really matter what **UI** triggers it. |
|  | As you can imagine, the typical **application flow** goes from the code in the **user interface**, through the application core to the **infrastructure code**, back to the **application core** and finally deliver a response to the **user interface**. |
|  | **Tools**  Far away from the most important code in our system, the **application core**, we have the **tools** that our application uses, for example, a database engine, a search engine, a Web server or a CLI console (although the last two are also delivery mechanisms).    While it might feel **weird** پُر اسرار , انہونی , غیر طبعی to put a CLI console in the same “bucket” as a database engine, and although they have different types of purposes, they are in fact tools used by the application. The key difference is that, while the CLI console and the web server are used to **tell our application to do something**, the database engine is **told by our application to do something**. This is a very relevant distinction, as it has strong implications اثر , اِشارہ on how we build the code that connects those tools with the application core. |
| **Primary or Driving Adapters**  The Primary or **Driver Adapters wrap around a Port** and use it to tell the Application Core what to do. **They translate whatever comes from a delivery mechanism into a method call in the Application Core.** | **Connecting the tools and delivery mechanisms to the Application Core:**  **------------------------------------------------------------------------------**  The code units that connect the tools to the application core are called **adapters (**[**Ports & Adapters Architecture**](https://herbertograca.com/2017/09/14/ports-adapters-architecture/)**).** The **adapters** are the ones that effectively implement the **code** that will allow the **business logic** to communicate with a specific **tool** and vice-versa.    The adapters that **tell** our application to do something are called **Primary or Driving Adapters** while the ones that are **told** by our application to do something are called **Secondary or Driven Adapters**.    **Ports**  These ***Adapters***, however, are not randomly created. They are created to fit a very specific entry point to the Application Core, a ***Port***. A port **is nothing more than a specification** of how the tool can use the application core, or how it is used by the Application Core. In most languages and in its most simple form, this specification, the Port, will be an **Interface**, but it might actually be composed of **several Interfaces and DTOs.**  It’s important to note that **the Ports (Interfaces) belong inside the business logic**, while the adapters belong outside. For this pattern to work as it should, it is of utmost importance that the **Ports** are created to fit the **Application Core** needs and not simply mimic نَقَل اُتارنا the tools APIs. |
|  | Web  Controllers  Entities  External Interfac  Enterprise Busines  Application  Interface  Frameworks &  use Case  Presenter  Output Port  use Case  Interaetor  How ot control  use Case  Controller  Input Port  In other words, our **Driving Adapters** are **Controllers or Console Commands** who are injected in their **constructor** with some **object** whose **class** implements the **interface (Port)** that the **controller or console command** requires.    In a more concrete example, a **Port** can be a **Service interface or a Repository interface** that a **controller** requires. The concrete implementation of the **Service**, **Repository** or **Query** is then injected and used in the Controller.    Alternatively, a **Port** can be a **Command Bus or Query Bus interface**. In this case, a concrete implementation of the Command or Query Bus is injected into the **Controller**, who then constructs a Command or Query and passes it to the relevant Bus. |

|  |  |
| --- | --- |
|  | **Secondary or Driven Adapters**    **-----------Observe (connectors) in diagram Carefully----------**  Unlike the **Driver Adapters**, who **wrap around** a **port**,  **the Driven Adapters implement a Port**, an **interface**, and are then injected into the Application Core, wherever the port is required (type-hinted).  **-----------Observe (connectors) in diagram Carefully----------**    For example, let’s suppose that we have a **naive** معصُومانہ , سِیدھا سادہ  application which needs to persist data. So, we create a persistence interface that meets its needs, with a method to *save* an array of data and a method to *delete* a line in a table by its ID. From then on, wherever our application needs to save or delete data we will require in its constructor an object that implements the persistence interface that we defined.    Now we create an adapter specific to MySQL which will implement that interface. It will have the methods to save an array and delete a line in a table, and we will inject it wherever the persistence interface is required.    If at some point we decide to change the database vendor, let’s say to PostgreSQL or MongoDB, we just need to create a new adapter that implements the persistence interface and is specific to PostgreSQL, and inject the new adapter instead of the old one. |
|  | **Inversion of control**  A characteristic to note about this pattern is that the adapters depend on a specific tool and a specific port (by implementing an interface). But our business logic only depends on the port (interface), which is designed to fit the business logic needs, so it doesn’t depend on a specific adapter or tool.    This means the direction of dependencies is towards the center, it’s the **inversion of control principle at the architectural level**.  Although, again, **it is of utmost importance that the Ports are created to fit the Application Core needs and not simply mimic the tools APIs.** |

**Application Core organization**

The [**Onion Architecture**](https://herbertograca.com/2017/09/21/onion-architecture/) picks up the **DDD layers** and incorporates شامِل کرنا , اِدارہ تشکیل کرنا them into the [**Ports & Adapters Architecture**](https://herbertograca.com/2017/09/14/ports-adapters-architecture/). Those **layers** are intended **مطلب ہونا , مُراد ہونا**

to bring some organization to the **business logic**, the interior of the **Ports & Adapters “hexagon”**, and just like in Ports & Adapters, the dependencies direction is towards the center.

**Application Layer**

The **use cases** are the processes that can be triggered **حَرکَت میں لانا** in our Application Core by one or several **User Interfaces** in our application. For example, in a **CMS** we could have the actual application UI used by the common users, another independent UI for the CMS administrators, another CLI UI, and a web API. These UIs (applications) could trigger use cases that can be specific to one of them or reused by several of them.

The **use cases** are defined in the **Application Layer**, the first layer provided by **DDD** and used by the **Onion Architecture**.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | This layer contains **Application Services** (& their **interfaces**) as first class citizens, but it also contains the Ports & Adapters interfaces (ports) which include ORM interfaces, search engines interfaces, messaging interfaces and so on. In the case where we are using a Command Bus and/or a Query Bus, this layer is where the respective Handlers for the Commands and Queries belong.  The Application Services and/or Command Handlers contain the logic to unfold a use case, a business process. Typically, their role is to:   |  | | --- | | * 1. use a repository to find one or several entities;   2. tell those entities to do some domain logic;   3. and use the repository to persist the entities again, effectively saving the data changes. | | The **Command Handlers** can be used in **two different ways:**   * 1. They can contain the actual logic to perform the **use case**;   2. They can be used as mere wiring pieces in our architecture, receiving a Command and simply triggering logic that exists in an Application Service. |   Which approach to use depends on the context, for example:   |  | | --- | | * + Do we already have the Application Services in place and are now adding a Command Bus?   + Does the Command Bus allow specifying any class/method as a handler, or do they need to extend or implement existing classes or interfaces? |   This layer also contains the triggering of **Application Events**, which represent some outcome of a use case. These events trigger logic that is a side effect of a use case, like sending emails, notifying a 3rd party API, sending a push notification, or even starting another **use case** that belongs to a different **component** of the application. |
| [**https://dzone.com/articles/what-is-spring-aop-1**](https://dzone.com/articles/what-is-spring-aop-1)  AOP can be done using 3-ways:   1. Pragmatic 2. Declarative using XML 3. Annotations   AOP is not related to OOP,  The key unit of **modularity** in OOP is the **class**; whereas in **AOP**, the unit of **modularity** is the **aspect**.  **AIM** of **AOP**: separate **business** from **services** | **Spring boot** makes **spring cloud** (micro-services & IOT) effort less. |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | [**https://www.youtube.com/watch?v=FBt31etvDZw&list=PLd3UqWTnYXOlc93disyBjyFv-r1Vq-5zh&index=69&ab\_channel=DurgaSoftwareSolutions**](https://www.youtube.com/watch?v=FBt31etvDZw&list=PLd3UqWTnYXOlc93disyBjyFv-r1Vq-5zh&index=69&ab_channel=DurgaSoftwareSolutions)   1. Method before Advice (MBA), is an interface class    1. 1st **Service** (POJO, Models) execute, 2nd **business logic** executes       1. 1st **Transaction code** executes then **deposit() code** executes       2. 1st **Security code** executes then **deposit() code** executes       3. Etc. 2. After returning Advice (ARA), is an interface class    1. 1st Service (POJO, Models) execute, 2nd **business logic** executes, 3rd again Service executes       1. 1st **Security code** executes then          1. **deposit() code** executes at last             1. **Transaction code** executes 3. Method interceptor for around (MIR), is an interface class    1. Executes before business logic & after business logic       1. 1st **Mail code** executes then          1. **deposit() code** executes at last             1. **Transaction code** executes 4. Throws Advice , is an interface class    1. While doing **deposit() code** execution if your database records are locked by another user, Database will throw (transaction exception) back into **deposit() code**, whenever you get transaction exception, how you will send error message to the user? Instead sending error message to user, we again process it, how?       1. We keep the data into **JMS**, later I can process **JMS data** into database tables. | **Terminologies of AOP:**   |  | | --- | | * + **Aspect** = a service.   + **Advice** = service provider (services can be provided using these advice)   + **Point cut** = a **point or condition** to execute aspects for business   + **Advisor** = point cut + advice   + **Proxy** = a weaver | | |  | |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | |  |  | | --- | --- | | class **Bank {**  **Int amount = 5000;**  **String accountNumber = 56i123** |  | | **int deposit(int amount, String accountNumber) {**  **If (accountNumber.equal(this.accountNumber))**    **}** | Requires transactions, you may **DML** operations on DB | | **Withdraw()** | Requires transactions, you may **DML** operations on DB | | **Find-balance()** | This is **DRL** operation, so transaction may not require | | **Calculate interest** | This is **DRL** operation, so transaction may not require | | **}** |  |   So if you write a service for Bank class it will execute for all methods, but you if you want to execute a service for a particular method then you use point cut (conditions) | | |

|  |  |  |
| --- | --- | --- |
|  | **Domain Layer (is not Domain Model)**  Further inwards, we have the Domain Layer. The objects (classes) in this layer contain the data and the logic to manipulate that data, that is specific to the Domain itself and it’s independent of the business processes that trigger that logic, they are independent and completely unaware of the Application Layer.  **Domain Services**  As I mentioned above, the role of an Application Service is to:   |  | | --- | | * 1. use a **repository** to find one or several **entities**;   2. tell those **entities** to do some **domain logic**;   3. and use the **repository** to **persist** the **entities** again, effectively saving the data changes. |   However, sometimes we encounter some **domain logic** that involves **different entities**, of the same **type** or not, and we feel that that **domain logic** does not belong in the **entities** themselves, we feel that that logic is not their direct responsibility.    So our first reaction might be to place that logic outside the **entities**, in an **Application Service**. However, this means that that **domain logic** will not be reusable in other **use cases: domain logic** should stay out of the **application layer**!  The solution is to create a **Domain Service**, which has the role of receiving a **set of entities** and performing some **business logic** on them. A Domain Service belongs to the Domain Layer, and therefore it knows nothing about the **classes** in the **Application Layer**, like the **Application Services** or the **Repositories**. In the other hand, it can use other **Domain Services** and, of course, the **Domain Model objects**. |

**Domain Model ( is not Domain Layer)**

In the very center, depending on nothing outside it, is the **Domain Model**, which contains the **business objects** that represent something in the **domain**. Examples of these objects are, first of all, **Entities** but also **Value Objects**, **Enums** and any **objects** used in the **Domain Model**.

The **Domain Model** is also where **Domain Events “live”**. These **events** are triggered when a specific set of **data** changes and they carry those changes with them. In other words, when an **entity** changes, a **Domain Event** is triggered and it carries the **changed properties new values**. These **events** are perfect, for example, to be used in **Event Sourcing**.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | |  | |  |  | | --- | --- | | **Jdepend.jar** | * 1. This **(.jar)** can help you avoiding importing/depending from **entities--to--> use-cases layer**   2. Importing/depending from **use-cases --to--> entities layer is allowed** | | **ArcUnit.jar** | Is a new/latest (.jar) that helps you achieve architectural characteristics something same like Jdepend or even better. |   [**Lesson 68 - Automating Architecture Governance**](https://www.youtube.com/watch?v=nFZePnD0AFQ&ab_channel=MarkRichards)   |  |  | | --- | --- | |  | [**ArchUnit - Unit Testing Architecture and Design**](https://www.youtube.com/watch?v=_ZUtb_hsm4Q&ab_channel=JavaUserGroupSwitzerland) | |   **What to Keep in Mind When You Write Tests**  If you have a large codebase, you should cache class file imports for your tests. This step can greatly reduce the tests’ execution time. Caching of the imported classes, for example, can be achieved with the ArchUnitRunner when you are using the JUnit support. For small codebases, the reimport overhead of the classes is negligible.    It is important that you write the concepts being tested as precisely as possible and that the concepts communicate clearly what the point of a rule is. For complex rules, it may make sense to use explicit and descriptive [rule text](https://www.archunit.org/userguide/html/000_Index.html#_controlling_the_rule_text) instead of relying on the generated rule text. And sometimes a long rule definition can be split into multiple short ones that are easier to understand.    Furthermore, you should not follow the implemented rules blindly. Likewise, you should not change them to pass the tests quickly without thinking beforehand. Sometimes new components do not fit the existing concepts; therefore, existing rules need to be changed or extended. That is especially true in an **evolving architecture**. But, as mentioned earlier, you should not overuse the options to ignore tests that ArchUnit offers. This is especially true in **greenfield** projects. |

**Components**

So far we have been segregating the code based on layers, but that is the fine-grained code segregation **الگ کرنے کا عمل**. The coarse-grained segregation of code is at least as important and it’s about segregating the code according to sub-domains and [***bounded contexts***](http://ddd.fed.wiki.org/view/welcome-visitors/view/domain-driven-design/view/bounded-context), following Robert C. Martin ideas expressed in [***screaming architecture***](https://8thlight.com/blog/uncle-bob/2011/09/30/Screaming-Architecture.html). This is often referred to as **“*Package by feature*”** or “*Package by component*” as opposed to **”*Package by layer*“**, and it’s quite well explained by Simon Brown in his blog post “[Package by component and architecturally-aligned testing](http://www.codingthearchitecture.com/2015/03/08/package_by_component_and_architecturally_aligned_testing.html)“:

|  |  |  |
| --- | --- | --- |
| **Package By Layer (Horizontal Slicing)** | **Package By Feature (Vertical Slicing)** | **Package By Component (Hybrid approach)** |
| [**Below mentioned is an example of implementation.**](https://www.youtube.com/watch?v=WOEYxPM0ljQ&list=LL&index=3&ab_channel=MarkRichards) | [**Below mentioned is an example of implementation.**](https://www.youtube.com/watch?v=WOEYxPM0ljQ&list=LL&index=3&ab_channel=MarkRichards) | [**Lesson 120 - Domain vs Technical Partitioning**](https://www.youtube.com/watch?v=WOEYxPM0ljQ&list=LL&index=3&ab_channel=MarkRichards)    [**Lesson 68 - Automating Architecture Governance**](https://www.youtube.com/watch?v=nFZePnD0AFQ&ab_channel=MarkRichards) |
| Let's assume that we're building a **web application** based upon the **Web-MVC pattern**. Packaging code by layer is typically the default approach because, after all, that's what the books, tutorials and framework samples tell us to do. Here we're organizing code by grouping things of the same type.    There's one top-level package for controllers, one for services (e.g. "business logic") and one for data access. Layers are the primary organization mechanism for the code. **Terms** such as **"separation of concerns"** are thrown around to justify this approach and generally layered architectures are thought of as a "good thing". Need to switch out the data access mechanism? No problem, everything is in one place. Each layer can also be tested in isolation to the others around it, using appropriate mocking techniques, etc. The problem with layered architectures is that [**they often turn into a big ball of mud**](http://tech.pro/blog/1498/why-the-n-layer-approach-is-bad-for-us-all) because, in Java anyway, you need to mark your classes as public for much of this to work. | Instead of organizing code by horizontal slice, package by feature seeks to do the opposite by organizing code by vertical slice.  Now everything related to a single feature (or feature set) resides in a single place. [**You can still have a layered architecture, but the layers reside inside the feature packages**](https://www.youtube.com/watch?v=WOEYxPM0ljQ&list=LL&index=3&ab_channel=MarkRichards)**.** In other words, layering is the secondary organization mechanism. The often-cited benefit is that it's "easier to navigate the codebase when you want to make a change to a feature", but this is a minor thing given the power of modern **IDEs**.  What you can do now though is hide feature specific classes and keep them out of sight from the rest of the codebase. For example, if you need any feature specific view models, you can create these as package-protected classes. The big question though is what happens when that new feature set C needs to access data from features A and B? Again, in Java, you'll need to start making classes publicly accessible from outside of the packages and the big ball of mud will again emerge.  Package by layer and package by feature both have their advantages and disadvantages. | This is a hybrid approach with increased modularity and an architecturally-evident coding style as the primary goals.  The basic premise **مُقَدّمَہ , واضح ہونا , بَیان کرنا** here is that I want my codebase to be made up of a number of coarse-grained components, with some sort of presentation layer (web UI, desktop UI, API, standalone app, etc.) built on top. A "component" in this sense is a combination of the business and data access logic related to a specific thing (e.g., domain concept, bounded context, etc.). As [**I've described before**](http://www.codingthearchitecture.com/2014/06/01/an_architecturally_evident_coding_style.html), I give these components a public interface and package-protected implementation details, which includes the data access code. If that new feature set C needs to access data related to A and B, it is forced to go through the public interface of components A and B. No direct access to the data access layer is allowed, and you can enforce this if you use Java's access modifiers properly. Again, "architectural layering" is a secondary organization mechanism. For this to work, you have to [**stop using the public keyword by default**](http://www.codingthearchitecture.com/2014/10/01/modularity_and_testability.html). This structure raises some interesting questions about testing, not least **کَم درجہ کی چیز**  about how we mock-out the data access code to create quick-running **"unit tests"**. |

|  |  |
| --- | --- |
|  | **Packaging By Component** To bring this discussion back to code, the organization of the codebase can really help or hinder here. Organizing a codebase by layer makes it easy to see the overall structure of the software but there are trade-offs. For example, you need to delve inside multiple layers (e.g. packages, namespaces, etc.) in order to make a change to a feature or user story. Also, many codebases end up looking eerily **پُراسرار طور پر** similar given the fairly standard approach to layering within enterprise systems. Uncle Bob Martin says that if you're looking at a codebase, it should scream something about the business domain. Organizing your code by feature rather than by layer gives you this, but again there are trade-offs. A variation I've been experimenting with is organizing code explicitly by **component**. The following screenshot shows an example of this in the codebase for my [techtribes.je](http://techtribes.je/) website (a content aggregator and portal for Jersey's digital sector). This screenshot only shows the core components; there's a separate **Spring MVC** project and the controllers use the components illustrated here.  This is similar to packaging by feature, but it's more akin **خُونی رشتہ دار , طرح** to the "micro services" that Mark Needham talks about in [his blog post](http://www.markhneedham.com/blog/2012/02/20/coding-packaging-by-vertical-slice/). Each **sub-package** of **je.techtribes.component** houses a separate **component**, complete with its own internal layering and Spring configuration. As far as possible, all of the internals are package scoped. You could potentially pull each component out and put it in it's own project or source code repository to be versioned separately. This approach will likely seem familiar to you if you're building something that has a very explicit loosely coupled architecture such as a distributed messaging system made up of loosely coupled components. I'm fairly confident that most people are still building something more monolithic in nature though, despite thinking about their system in terms of components. I've certainly **یقیناً , بلا شبہ** packaged \*parts\* of monolithic codebases using a similar approach in the past but it's tended to be fairly **ad hoc خاص مقصد کے لیے.** Let's be honest, organizing code into packages isn't something that gets a lot of brain-time, particularly given the refactoring tools that we have at our disposal. Organizing code by component lets you explicitly reflect the concept of "a component" from the architecture into the codebase. If your software architecture diagram screams something about your business domain (and it should), this will be reflected in your codebase too. |

|  |  |
| --- | --- |
| **Architecturally aligned Testing**  The short answer is don't bother, unless you really need to. [I've spoken about](https://www.youtube.com/watch?v=ehH3UGdSwPo) and [written about](http://www.codingthearchitecture.com/2014/10/01/modularity_and_testability.html) this before, but architecture and testing are related. Instead of the typical testing triangle (lots of "unit" tests, fewer slower running "integration" tests and even fewer slower UI tests), consider this.  I'm trying to make a conscious effort to not use the term "unit testing" because everybody has a different view of how big a "unit" is. Instead, I've adopted a strategy where some classes can and should be tested in isolation. This includes things like domain classes, utility classes, web controllers (with mocked components), etc. Then there are some things that are easiest to test as components, through the public interface. If I have a component that stores data in a MySQL database, I want to test everything from the public interface right back to the MySQL database. These are typically called "integration tests", but again, this term means different things to different people. Of course, treating the component as a black box is easier if I have control over everything it touches. If you have a component that is sending asynchronous messages or using an external, third-party service, you'll probably still need to consider adding dependency injection points (e.g. ports (**interface**) and adapters) to adequately test the component, but this is the exception not the rule. All of this still applies if you are building a **micro-services style** of architecture. You'll probably have some low-level class tests, hopefully a bunch of service tests where you're testing your **micro-services** though their public interface, and some system tests that run scenarios end-to-end. Oh, and you can still write all of this in a test-first, TDD style if that's how you work.  I'm using this strategy for some systems that I'm building and it seems to work really well. I have a relatively simple, clean and (to be honest) boring codebase with understandable dependencies, minimal [test-induced design damage](http://david.heinemeierhansson.com/2014/test-induced-design-damage.html) and a manageable quantity of test code. This strategy also bridges the [model-code gap](http://www.codingthearchitecture.com/2014/05/29/software_architecture_vs_code.html), where the resulting code actually reflects the architectural intent. In other words, we often draw "components" on a whiteboard when having architecture discussions, but those components are hard to find in the resulting codebase. Packaging code by layer is a major reason why this mismatch between the diagram and the code exists. Those of you who are familiar with my [C4 model](https://leanpub.com/software-architecture-for-developers/read#c4) will probably have noticed the use of the terms "class" and "component". This is no coincidence. Architecture and testing are more related than perhaps we've admitted in the past. | **Architecturally aligned Testing (**& a reshaped pyramid**)** |

**Challenging the traditional layered approach**

This deserves a separate blog post, but something I also mentioned during the talk was that teams should challenge the traditional layered architecture and the way that we structure our codebase. One way to achieve a nice mapping between architecture and code is to ensure that your code **reflects** the **abstract concepts** shown on your **architecture diagrams**, which can be achieved by [writing components rather than classes in layers](http://www.codingthearchitecture.com/2013/04/08/mapping_software_architecture_to_code.html). Another side-effect of changing the organization of the code is less [test-induced design damage](http://david.heinemeierhansson.com/2014/test-induced-design-damage.html). The key question to ask here is whether **layers** are architecturally significant building blocks or merely an implementation detail, which should be wrapped up inside of (e.g.) components. As I said, this needs a separate blog post.

|  |
| --- |
| There are a number of reasons why auto-generating such diagrams is tricky but, once we start coding, much of the semantics associated with "containers" (runtime environments, process boundaries, etc.) and "components" becomes lost of the sea of classes that make up the typical codebase. Many developers break their systems up into a number of projects within their IDEs to represent **reusable libraries and deployable units** but external tools often don't have access to this information if they are solely working from a **bunch of JAR files or DLLs (for example)**. In essence, the information related to the abstract structural elements isn't adequately represented within a codebase. If you take a look at most codebases, I'm fairly sure that you could come up with a set of rules as to what defines a component but perhaps it would be easier to simply make these concepts explicit. Some techniques already exist to do this (e.g. the [Architecture Description Language](https://en.wikipedia.org/wiki/Architecture_description_language)) but I've never seen them used in the corporate world. |
| |  |  |  | | --- | --- | --- | | Answer to above paragraph "Frank, I think there is a misconception lurking **چُھپ کر بیٹھنا** **, فضول میں وقت گزارنا** in the dark. If your **team** is small enough that everybody can work on the same **code base**, the **structuring** can simply happen by putting stuff in the 'right' package. No extra interface, no extra separation. If and only if your application grows one would consider splitting it into separate **projects/maven modules** or whatever your tool names those thingies. A clean package structure will give you a head start on that." | In **Java** to achieve this **structure** in reality you require **maven build tool** | The major **caveat اِنتباہ** to all of this is that designing a software system based around components isn't "the only way". It's a nice approach to think about software systems that are more monolithic in nature and it's a great stepping stone to designing loosely coupled architectures. But it isn't a silver bullet. Regardless of how you design software, I do hope this post has got you thinking about the mapping between software architecture and how it's reflected in the code. | |

**-----------------------------------------------------------------------------------------------------------------------------Continue----------------------------------------------------------------------------------------------------------------------------------------------------------**

|  |  |
| --- | --- |
| I am an advocate for the “***Package by component*”** approach and, picking up on Simon Brown diagram about ***Package by component***, I would shamelessly change it to the following: | These sections of code are cross-cutting to the layers previously described, they are the [***components***](https://herbertograca.com/2017/07/05/software-architecture-premises/) of our application. Examples of components can be **~~Authentication~~, ~~Authorization~~**, **Billing**, **User**, **Review or Account**, but they are always related to the **domain**. **Bounded contexts** like **Authorization and/or Authentication** should be seen as external **tools** for which we create an **adapter** and hide behind some kind of **port**. |

|  |  |
| --- | --- |
| **Decoupling the components**  Just like the **fine-grained code units** (classes, interfaces, traits, mixins, …), also the **coarsely grained code-units** (**components**) benefit from low coupling and high cohesion.    To decouple classes we make use of **Dependency Injection**, by injecting dependencies into a class as opposed to instantiating them inside the class, and **Dependency Inversion**, by making the class depend on **abstractions** (interfaces and/or abstract classes) instead of concrete classes. This means that the depending class has no knowledge about the concrete class that it is going to use, it has no reference to the fully qualified class name of the classes that it depends on.    In the same way, having completely decoupled components means that a component has **no** direct knowledge of any another component. In other words, it has no reference to any fine-grained code unit from another component, not even **interfaces!** This means that Dependency Injection and Dependency Inversion are not enough to decouple components, we will need some sort of architectural constructs. We might need **events**, a **shared kernel**, **eventual consistency**, and even a **discovery service**!    **Triggering مُتحَرَک کَرنا , حَرکَت میں لانا logic in other components**  When one of our components (**component B**) needs to do something whenever something else happens in another component (**component A**), we cannot simply make a direct call from component A to a class/method in component B because A would then be coupled to B.    However we can make A use an **event dispatcher** to **dispatch** an **application event** that will be delivered to any component **listening** to it, including **B**, and the **event listener** in B will trigger the desired action. This means that component A will depend on an **event dispatcher**, but it will be decoupled from B.  ------------------**Observe Diagram carefully**---------------------------  **Domain Layer & Domain Model are two Separate things** |  |

Nevertheless, پھر بھی , اس کے باوجود if the **event** itself “lives” in **A** this means that **B** knows about the existence of **A**, it is coupled to **A**. To remove this dependency, we can create a library with a set of application core functionality that will be shared among all **components**, the [**Shared Kernel**](http://ddd.fed.wiki.org/view/welcome-visitors/view/domain-driven-design/view/shared-kernel). This means that the components will both depend on the **Shared Kernel** but they will be **decoupled** from each other. The **Shared Kernel** will contain functionality like **application and domain events**, but it can also contain **Specification objects**, and whatever makes sense to share, keeping in mind that it should be as minimal as possible because any changes to the **Shared Kernel** will affect **all components of the application**. Furthermore, if we have a **polyglot** system, let’s say a **micro-services ecosystem** where they are written in different languages, the **Shared Kernel** needs to be **language agnostic** so that it can be understood by all **components**, whatever the language they have been written in. For example, instead of the **Shared Kernel** containing an Event class, it will contain the **event description** (i.e.. name, properties, maybe even methods although these would be more useful in a Specification object) in an agnostic language like JSON, so that all components/micro-services can interpret it and maybe even auto-generate their own concrete implementations. Read more about this in my follow-up post: [More than concentric layers](https://herbertograca.com/2018/07/07/more-than-concentric-layers/).

|  |  |
| --- | --- |
|  | This approach works both in **monolithic applications and distributed applications** like micro-services ecosystems. However, when the **events** can only be delivered **asynchronously**, for contexts where triggering logic in other components needs to be done immediately this approach will not suffice! Component **A** will need to make a direct **HTTP** call to component **B**. In this case, to have the components **decoupled**, we will need a **discovery service** to which **A** will ask where it should send the request to trigger the desired action, or alternatively make the request to the **discovery service** who can **proxy** it to the relevant service and eventually return a response back to the requester. This approach will couple the components to the discovery service but will keep them decoupled from each other.  **-------------------------Continue from Heading--->(Getting data from other components) --------------------------**  **More than concentric layers**  In [my previous post in this series](https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together/), I published an infographic that reflects the mental map I use to figure out the relationships between the code units types.    However, there was something that I always felt it was not very well reflected there, but I just didn’t know how to do it any better: the **shared kernel.**    Furthermore, I figured out a few more things, and I’m gonna write about it all in this post!  Looking at the infographic I published in my last post of this series, we see the **shared kernel** in the very center of the diagram, looking like it is within the **Domain layer** and above the conic sections that represent **bounded contexts**.  Despite its location, I did not mean to **imply اِشارتاً کہنا** that the **shared kernel** depends on the remainder of the code nor that the shared kernel is another layer within the Domain layer. |
| **What is the shared kernel?!**  The shared kernel as defined by **Eric Evans**, the *father* of **DDD**, is the code that the development team decides to share between several bounded contexts:  *[…] some subset of the domain model that the two teams agree to share. Of course this includes, along with this subset of the model, the subset of code or of the database design associated with that part of the model. This explicitly shared stuff has special status, and shouldn’t be changed without consultation with the other team.*  [*Shared Kernel*](http://ddd.fed.wiki.org/view/shared-kernel)*, DDD wiki by Ward Cunningham*  So basically, it can be any type of code: domain layer code, application layer code, libraries, … whatever. | **Bounded Context**  Multiple models are in play on any large project. Yet when code based on distinct models is combined, software becomes buggy, unreliable, and difficult to understand. Communication among team members becomes confused. It is often unclear in what context a model should not be applied.  **Therefore:**  Explicitly define the context within which a model applies. Explicitly set boundaries in terms of team organization, usage within specific parts of the application, and physical manifestations such as code bases and database schemas. Keep the model strictly consistent within these bounds, but don't be distracted or confused by issues outside. |

|  |  |
| --- | --- |
| In the **context** of the **mental map** I use, however, I think of it as a subset of that, as specific types of code. In my mental map, the shared kernel contains the domain and application layers code that is shared among bounded contexts in order for communication between bounded contexts to be possible.  This means, for example, the **events** that are triggered in one or more bounded contexts and **listened** to within other bounded contexts. Together with those **events** we need to share all data types that are used by those events, for example: entities IDs, value objects, enums, etc. Complex objects like entities should not be used by events directly because they can be problematic to **serialize/un-serialize** into/from a **queue**, so the shared code shouldn’t propagate much.  Of course that if we have in our hands a polyglot system, composed of micro-services developed in different languages, this shared kernel needs to be descriptive, in json, xml, yaml, or whatever, so that all micro-services can understand it.  As a result, this shared kernel is completely decoupled from the remainder of the codebase, from the components. This is great, as it means that the components, although coupled to the shared kernel, are decoupled between each other. The shared code is explicitly identified and easily extractable to a separate library.  This is also quite handy if and when we decide to extract one of the bounded contexts into a micro-service separated from the monolith, we know exactly what is shared and we can simply extract the shared kernel into a library to be installed in both the monolith and the micro-service. | **2003 Domain Driven Design**    **Observe carefully "CRM" is Generic subdomain (buy it "Microsft-365 dynamics") & many vendors** |

So to recap, in my mental map, the application core depends on the shared kernel which contains code, from the domain and application layers, that is shared between bounded contexts.

**When the language is insufficient…**

So, we have the application code with all its concentric layers, and the application core depends on the shared kernel, which sits below all that code.

We can also say that all that code depends on the programming language(s) being used, but that is such an obvious fact that we tend to completely ignore it.

I am bringing it up, however, because of the question “what do we do when the language constructs are not enough?!”. Well, obviously we create those language constructs ourselves and thus compensate for the language flaws. But the important follow up questions I have are “How do we transmit the rationale behind that code existence? Where do we place it? How do we make explicit when and how it should be used?”.

What I’ve seen, and done myself, is a package called Utils or Commons where that code is placed. But this ends up being a bucket where we dump all the code that we don’t know where to put! All kind of different code, with different purposes and usability (wrapped in an adapter, used directly, …) ends up being dumped there, the package has no conceptual meaning, no consistency, no cohesion, no explicitness, plenty of ambiguity.

I want to abolish the Utils and Commons packages!

All packages must have conceptual cohesion! When and how a package is to be used must be explicit! There must be no ambiguity!

So, for example, if our application has some special way to interact with the CLI, instead of placing it in a namespace ‘Acme/Util/SpecialCli’ we can put it into ‘Acme/App/Infrastructure/Cli/SpecialCli’ which tells us that this package is about the CLI, it is part of the infrastructure of the application ‘App’ of company ‘Acme’. Being part of the App infrastructure also tells us that there is a port, in the application core, to which this package complies to.

Alternatively, if we see this package as something that the language itself should/could have, we can place it in a namespace that reflects that, for example ‘Acme/PhpExtension/SpecialCli’. This tells us that this package should be seen as part of the language itself, and therefore its code should be used directly in the codebase as any language construct. Of course, however, if another company depends on this package it might be wise for them not to depend directly on it, it is safer to create a port/adapter so they can swap it for something else, but if we own the package we can decide to treat it as part of the language as the risk of needing to swap it for another alternative is not that much. It’s always about the tradeoffs.

Another example of something I could consider part of the language would be UUIDs in PHP. I can imagine it is not part of the language because there is a new version every once in a while and it would be a maintainability burden, but otherwise it would be a concept generic, widespread and consistent enough to be part of the language.

So, why not create an implementation of UUIDs and use it as if it was part of PHP itself, just like we use a DateTime object?! As long as we control the implementation, I can see no down side to it.

What about Doctrine Query Language (DQL) ? (Doctrine is a port of Hibernate into PHP) Can we treat DQL as if it was SQL, Elasticsearch QL, or Mongo QL?

|  |  |
| --- | --- |
| **Conclusion**  So in conclusion, I see these 4 major types of code at the macro level, and I think making them explicit in the code organization can be key for us to stop ending up with a big ball of mud.    What, for me, is an unquestionable truth is that**the architecture will always be there, the only question is: are we in control of it or not?!**  So let’s **explicitly organize the code so it helps communicate the architecture**, follow my mental map fully or partially, follow your own mental map, or follow someone else’s mental map, but please let’s use some consistent reasoning to organize the code and make the project explicitly communicate its architecture using its structure, its code organization. |  |

This approach works both in **monolithic applications and distributed applications** like micro-services ecosystems. However, when the **events** can only be delivered **asynchronously**, for contexts where triggering logic in other components needs to be done immediately this approach will not suffice! Component **A** will need to make a direct **HTTP** call to component **B**. In this case, to have the components **decoupled**, we will need a **discovery service** to which **A** will ask where it should send the request to trigger the desired action, or alternatively make the request to the **discovery service** who can **proxy** it to the relevant service and eventually return a response back to the requester. This approach will couple the components to the discovery service but will keep them decoupled from each other.

**Getting data from other components**

The way I see it, a component is not allowed to change data that it does not “own”, but it is fine for it to query and use any data.

**Data storage shared between components**

When a component needs to use **data** that belongs to another component, let’s say a billing component needs to use the client name which belongs to the **accounts component**, the billing component will contain a query object that will query the **data storage** for that **data**. This simply means that the **billing component** can know about any dataset, but it must use the data that it does not “own” as read-only, by the means of queries.

**Data storage segregated per component**

In this case, the same pattern applies, but we have **more complexity** at the **data storage level**. Having components with their own data storage means each data storage contains:

|  |
| --- |
| * + A **set of data** that it owns and is the only one allowed to change, making it the single source of truth;   + A **set of data** that is a copy of other components data, which it cannot change on its own, but is needed for the component functionality, and it needs to be updated whenever it changes in the owner component. |

Each component will create a local copy of the data it needs from other components, to be used when needed. When the data changes in the component that owns it, that owner component will trigger a **domain event** carrying the **data** changes. The components holding a copy of that data will be **listening** to that domain event and will update their local copy accordingly.

**Flow of control**

As I said above, the flow of control goes, of course, from the user into the Application Core, over to the infrastructure tools, back to the Application Core and finally back to the user. But how exactly do classes fit together? Which ones depend on which ones? How do we compose them?

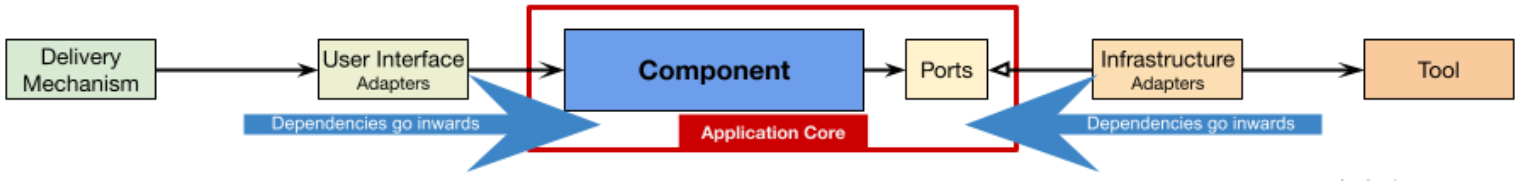
Following Uncle Bob, in his article about Clean Architecture, I will try to explain the flow of control with UMLish diagrams…

|  |  |
| --- | --- |
|  | **Without a Command/Query Bus**  In the case we do not use a command bus, the **Controllers** will **depend** either on an **Application Service** or on a **Query object**.    In the **diagram** on left side we use an **interface** for the Application Service, although we might argue that it is not really needed since the Application Service is part of our application code and we will not want to swap it for another implementation, although we might refactor it entirely.    The Query object will contain an optimized query that will simply return some raw data to be shown to the **user**. That data will be returned in a **DTO** which will be injected into a **ViewModel**. This **ViewModel** may have some view logic in it, and it will be used to populate a **View**.    The **Application Service**, on the other hand, will contain the **use case logic**, the logic we will trigger when we want to do something in the system, as opposed to simply view some data. The **Application Services** depend on **Repositories** which will return the **Entity(ies)** that contain the logic which needs to be triggered. It might also depend on a **Domain Service** to coordinate a domain process in several entities, but that is hardly ever the case.  After unfolding the **use case**, the **Application Service** might want to notify the whole **system** that, that **use case** has happened, in which case it will also depend on an **event dispatcher** to trigger the **event**.  It is interesting to note that we place **interfaces** both on the **persistence engine** and on the **repositories**. Although it might seem redundant, they serve different **purposes:** |

|  |
| --- |
| * + The **persistence interface** is an **abstraction layer over the ORM** so we can swap the **ORM** being used with no changes to the **Application Core**.   + The **repository interface** is an **abstraction on the persistence engine** itself. Let’s say we want to switch from **MySQL** to **MongoDB**. The **persistence interface** can be the same, and, if we want to continue using the same ORM, even the **persistence adapter** will stay the same. However, the **query language** is completely different, so we can create new repositories which use the same persistence mechanism, implement the same repository interfaces but builds the queries using the **MongoDB query language** instead of **SQL**. |

|  |  |
| --- | --- |
|  | **With a Command/Query Bus**  In the case that our application uses a **Command/Query Bus**, the diagram stays pretty much the same, with the exception that the **controller** now depends on the **Bus** and on a **command or a Query**. It will instantiate the Command or the Query, and pass it along to the Bus who will find the appropriate handler to receive and handle the command.    In the diagram on left side, the **Command Handler** then uses an **Application Service**. However, that is not always needed, in fact in most of the cases the **handler** will contain all the logic of the **use case**. We only need to extract logic from the **handler** into a separated **Application Service** if we need to **reuse** that same logic in another **handler**.    You might have noticed that there is no **dependency** between the **Bus** and the **Command**, the **Query** nor the **Handlers**. This is because they should, in fact, be unaware of each other in order to provide for good **decoupling**. The way the Bus will know what Handler should handle what Command, or Query, should be set up with mere configuration.    As you can see, in both cases all the arrows, the dependencies, that cross the border of the application core, they point inwards. As explained before, this a fundamental rule of Ports & Adapters Architecture, Onion Architecture and Clean Architecture. |

As you can see, in both cases all the arrows, the dependencies, that cross the border of the application core, they point inwards. As explained before, this a fundamental rule of Ports & Adapters Architecture, Onion Architecture and Clean Architecture.



**Conclusion**

The goal, as always, is to have a codebase that is loosely coupled and high cohesive, so that changes are easy, fast and safe to make.

*Plans are worthless, but planning is everything.*

*Eisenhower*

This infographic is a concept map. Knowing and understanding all of these concepts will help us plan for a healthy architecture, a healthy application.

Nevertheless:

*The map is not the territory.*

*Alfred Korzybski*

Meaning that **these are just guidelines! The application is the territory, the reality, the concrete use case where we need to apply our knowledge, and that is what will define what the actual architecture will look like!**

**We need to understand all these patterns, but we also always need to think and understand exactly what our application needs, how far should we go for the sake of decoupling and cohesiveness.** This decision can depend on plenty of factors, starting with the project functional requirements, but can also include factors like the time-frame to build the application, the lifespan of the application, the experience of the development team, and so on.

This is it, this is how I make sense of it all. This is how I rationalize it in my head.

However, how do we make all this explicit in the code base? That’s the subject of one of my next posts: how to reflect the architecture and domain, in the code.

Last but not least, thanks to my colleague [Francesco Mastrogiacomo](https://www.linkedin.com/in/francescomastrogiacomo/), for helping me make my infographic look nice.

**Reflecting architecture and domain in code**

**When creating an application, the easy part is to build something that works. To build something that has performance despite handling a massive load of data, that is a bit more difficult. But the greatest challenge is to build an application that is actually maintainable for many years (10, 20, 100 years).**

Most companies where I worked have a history of rebuilding their applications every 3 to 5 years, some even 2 years. This has extremely high costs, it has a major impact on how successful the application is, and therefore how successful the company is, besides being extremely frustrating for developers to work with a messy code base, and making them want to leave the company. A serious company, with a long-term vision, cannot afford any of it, not the financial loss, not the time loss, not the reputation loss, not the client loss, not the talent loss.

Reflecting the architecture and domain in the codebase is fundamental to the maintainability of an application, and therefore crucial in preventing all those nasty problems.

**Explicit Architecture** is how I rationalize a set of principles and practices advocated by developers far more experienced than me and how I organize a code base to make it reflect and communicate the architecture and domain of the project.

In my [previous post](https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together/), I talked about how I put all those ideas together and presented some infographics and *UMLish* diagrams to try to create some kind of a concept map of how I think of it.

However, how do we actually put it to practice in our codebase?!

In this post, I will talk about how I reflect the architecture and the domain of a project in the code and will propose a generic structure that I think that can help us plan for maintainability.

|  |  |  |
| --- | --- | --- |
| **My two mental maps**  In the last two posts of this series I explained the two mental maps I use to help me think about code and organize a codebase, at least in my head.  The first one is composed by a **series of concentric layers**, which in the end are **sliced** to make up the **domain wise modules** of the application, the **components**. In this diagram, the dependency direction goes inwards, meaning that outer layers know about inner layers, but not the other way around.   |  |  | | --- | --- | |  | The **second** one is a set of **horizontal layers**, where the previous diagram sits on top, followed by the code shared by the components (shared kernel), followed by our own extensions to the languages, and finally the actual programming languages in the bottom. Here, the dependencies direction goes downwards. | |

**Architecturally evident coding style**

Using an architecturally evident coding style means that our coding style (coding standards, class, methods and variables naming conventions, code structure, …) somehow communicates the domain and the architecture to who is reading the code. There are two main ideas on how to achieve an architecturally evident coding style.

*“[…] an architecturally evident coding style that lets you drop hints to code readers so that they can correctly infer the design.”*

[***George Fairbanks***](https://resources.sei.cmu.edu/asset_files/Presentation/2013_017_001_48651.pdf)

The first one is about using the code artefacts (classes, variables, modules, …) names to convey both domain and architectural meaning. So, if we have a class that is a repository dealing with invoice entities, we should name it something like **`Invoice-Repository`**, which will tell us that it deals with the Invoice domain concept and its architectural role is that of a repository. This helps us know and understand where it should be located, plus how and when to use it. Nevertheless, I think we don’t need to do it with every code artefact in our codebase, for example I feel that post-fixing an entity with ‘Entity’ is redundant and just adds noise.

*“[…] the code should reflect the architecture. In other words, if I look at the code, I should be able to clearly identify each of the components […]”*

[***Simon Brown***](http://www.codingthearchitecture.com/2014/06/01/an_architecturally_evident_coding_style.html)

The second one is about making sub-domains explicit as top level artefacts of our codebase, as domain wise modules, as components.

So, the first one should be clear and I don’t think it requires any further explanation. However, the second one is more tricky, so let's dive into that.

**Making architecture explicit**

We’ve seen, in my first diagram, that at the highest zoom level we have 3 different types of code:

|  |
| --- |
| * + **The user interface**, containing the code adapting a delivery mechanism to a use case;   + **The application core**, containing the use cases and the domain logic;   + **The infrastructure**, containing the code that adapts the **tools**/**libraries** to the application core needs. |

|  |  |
| --- | --- |
|  | So at the root of our source folder we can reflect these types of code by creating 3 folders, one for each type of code. These three folders will represent three **namespaces** and later on we can even create a test to assert that both the user interface and the infrastructure know about the core, but not the other way around, in other words, we can test that the dependencies direction goes inwards. |

|  |  |
| --- | --- |
|  | **The user interface**  Within a web enterprise application, it is common to have several APIs, for example a REST API for clients, another one for web-hooks used by 3rd party applications, maybe a legacy SOAP API that still needs to be maintained, or maybe a GraphQL API for a new mobile app…    It is also common for such applications to have several CLI commands used by Cron jobs or on demand maintenance operations.    And of course, it will have the website itself, used by regular users, but maybe also another website used by the administrators of the application.    These are all different views on the same application, they are all different user interfaces of the application.    So our application can actually have several user interfaces, even if some of them are used only by non-human users (other 3rd party applications). Let’s reflect that by means of folders/namespaces to separate and isolate the several user interfaces.  We have mainly 3 types of user interfaces, APIs, CLI and websites. So let's start by making that difference explicit inside the ***UserInterface*** root namespace by creating one folder for each of them. |

|  |  |
| --- | --- |
|  | Next, we go deeper and inside the namespace for each type and, if needed, we create a namespace for each UI (maybe for the CLI we don’t need to do it).  **The infrastructure**  In a similar way as in the User Interface, our application uses several tools (libraries and 3rd party applications), for example an ORM, a message queue, or an SMS provider.  Furthermore, for each of these tools we might need to have several implementations. For example, consider the case where a company expands to another country and for pricing reasons it's better to use a different SMS provider in each country: we will need different adapter implementations using the same port so they can be used interchangeably. Another case is when we are refactoring the database schema, or even switching the DB engine, and need (or decide) to also switch to another ORM: then we will have 2 ORM adapters hooked into our application. |

|  |  |
| --- | --- |
|  | So within the Infrastructure namespace we start by creating a namespace for each tool type (ORM, MessageQueue, SmsClient), and inside each of those we create a namespace for each of the adapters of the vendors we use (Doctrine, Propel, MessageBird, Twilio, …).  **The Core**  Within the Core, at the highest level of zoom, we have three types of code, the **Components**, the **Shared Kernel** and the **Ports**. So, we create folders/namespaces for all of them.  **Components**  In the Components namespace we create a namespace for each component, and inside each of those we create a namespace for the Application layer and a namespace for the Domain layer. Inside the Application and Domain namespaces we start by just dumping all classes on there and as the number of classes grow, we start grouping them as needed (I find it overzealous **حد سے زیادہ جوشیلا** to create a folder to put just one class in it, so I rather do it as the need for it arises). |

|  |  |
| --- | --- |
|  | At this point we need to decide if we should group them by subject (invoice, transaction, …) or by technical role (repository, service, value object, …), but I feel that, whatever the choice, it doesn’t really have much impact because we are at the leafs of the organization tree so, if needed, it's easy to do changes to that last bit of structure without much impact to the rest of the codebase.  **Ports**  The Ports namespace will contain a namespace for each tool the Core uses, just like we did for the Infrastructure, where we will place the code that the core will use in order to use the underlying tool. |

|  |  |
| --- | --- |
|  | This code will also be used by the adapters, whose role is to translate between the port and the actual tool. In its simplest form, a **port** is just an **Interface** but in many cases it also needs any of the following:   1. **value objects,** 2. **DTOs,** 3. **services,** 4. **builders,** 5. **query objects or even** 6. **repositories**.   **Shared Kernel**  In the Shared Kernel we will place the code that is shared among Components. After experimenting with different inner structures for the Shared Kernel, I can’t decide on a structure that will fit all scenarios. I feel that for some code it makes sense to separate it per component as we did in Core\Component (i.e.. Entity IDs clearly belong to one component) but other cases not so much (i.e.. Events might be triggered and listened to by several components, so they belong to none). Maybe a mix is the better fit. |

|  |  |
| --- | --- |
|  | **User-land language extensions**  Last, but not least, we have our own extensions to the language. As explained in the previous post on this series, this is code that could be part of the language but, for some reason, it’s not. In the case of PHP we can think, for example, of a DateTime class based on the one provided by PHP but with some extra methods. Another example could be a UUID class, which although not provided by PHP, it is by nature very aseptic, domain agnostic, and therefore could be used by any project independently of the Domain.    This code is used as if it would be provided by the language itself, so it needs to be fully under our control. This doesn’t mean, however, that we can’t use 3rd party libraries. We can, and should use them when it makes sense, but they must be wrapped by our own implementation (so that we can easily switch the underlying 3rd party library) which is the code that will be used directly on the application codebase. Eventually, it can be a project on its own, in its own CVS repository, and used in several projects. |

**Enforcing the architecture**

All these ideas and the way we decide to put them to practice, are a lot to take in, and they are not easy to master. Even if we do master all this, in the end we are only humans, so we will make mistakes and our colleagues will make mistakes, it’s just the way it goes.

Just like we make mistakes in code and have a test suite to prevent those mistakes from reaching production, we must do the same with the codebase structure.

To do this, in the PHP world we have a little tool called [Deptrac](https://github.com/sensiolabs-de/deptrac) (but I bet similar tools exist for other languages as well), created by Sensiolabs. We configure it using a **yaml** file, where we define the layers we have and the allowed dependencies between them. Then we run it through the command line, which means we can easily run it in a CI, just like we run a test suite in the CI.

We can even have it create a diagram of the dependencies, which will visually show us the dependencies, including the ones breaking the configured rule set:

|  |  |
| --- | --- |
|  | **Conclusion**  An application is composed of a **domain** and a **technical structure, the architecture**. Those are the real differentials in an application, not the used tools, libraries, or delivery mechanisms. If we want an application to be maintainable for a long time, both of these need to be explicit in the codebase, so that developers can know about it, understand it, comply to it and further evolve it as needed.  This explicitness will help us understand the boundaries as we code, which will in turn help us keep the application design modular, with high cohesion and low coupling.  Again, most of these ideas and practices I have been talking about in my previous posts, come from developers far better and more experienced than me. I have discussed them at length with plenty of my colleagues in different companies, I have experimented with them in codebases of enterprise applications, and they have been working very well for the projects I’ve been involved with.  Nevertheless, I believe there are no *silver bullets*, no *one boot fits all*, no *Holy Grail*.  These ideas and the structure I refer in this post should be seen as a generic template that can be used in most enterprise applications, but if necessary it should be adapted with no regrets. We always need to evaluate the context (the project, the team, the business, …) and do the best we can, but I believe and hope that this template is a good starting point or, at the very least, food for thought. |