**THE CLEAN ARCHITECTURE**

Architecture systems have the same objective, which is the separation of concerns. They all achieve this separation by dividing the software into layers. Each has at least one layer for business rules, and another layer for user and system interfaces.

Architecture produces systems that have the following characteristics:

• Independent of frameworks. The architecture does not depend on the existence of some library of feature-laden software. This allows you to use such frameworks as tools, rather than forcing you to cram your system into their limited constraints.

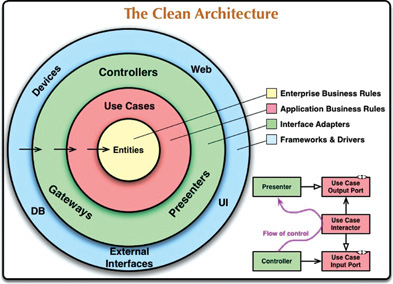
• Testable. The business rules can be tested without the UI, database, web server, or any other external element.

• Independent of the UI. The UI can change easily, without changing the rest of the system. A web UI could be replaced with a console UI, for example, without changing the business rules.

• Independent of the database. You can swap out Oracle or SQL Server for Mongo, BigTable, CouchDB, or something else. Your business rules are not bound to the database.

• Independent of any external agency. In fact, your business rules don’t know anything at all about the interfaces to the outside world.

Example of Clean Architecture:



The concentric circles represent different areas of software. In general, the further in you go, the higher level the software becomes. The outer circles are mechanisms. The inner circles are policies.

The overriding rule that makes this architecture work is the Dependency Rule: \* *Source code dependencies must point only inward, toward higher-level policies.*

Nothing in an inner circle can know anything at all about something in an outer circle. In particular, the name of something declared in an outer circle must not be mentioned by the code in an inner circle. That includes functions, classes, variables, or any other named software entity. By the same token, data formats declared in an outer circle should not be used by an inner circle, especially if those formats are generated by a framework in an outer circle. We don’t want anything in an outer circle to impact the inner circles.

**Entities**

Entities encapsulate enterprise-wide Critical Business Rules. An **entity** can be an **object with methods**, or it can be a **set of data structures and functions**. It doesn’t matter so long as the entities can be used by many different applications in the enterprise. They are the least likely to change when something external changes. For example, you would not expect these objects to be affected by a change to page navigation or security. No operational change to any particular application should affect the entity layer.

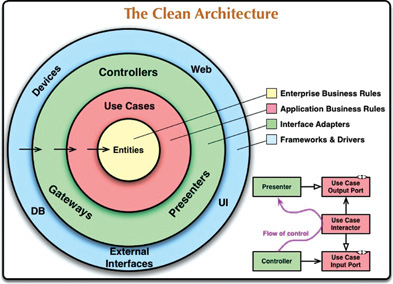
**Use Cases**

The software in the use cases layer **contains application-specific business rules**. It encapsulates and implements all of the use cases of the system. These use cases orchestrate the flow of data to and from the entities, and direct those entities to use their Critical Business Rules to achieve the goals of the use case. We do not expect changes in this layer to affect the entities. We also do not expect this layer to be affected by changes to externalities such as the database, the UI, or any of the common frameworks. The use cases layer is isolated from such concerns.

**Interface Adapters**

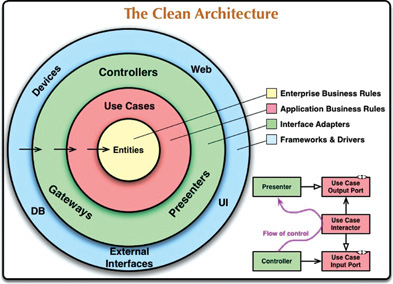
The software in the interface adapters layer is a set of adapters that convert data from the format most convenient for the use cases and entities, to the format most convenient for some external agency such as the database or the web. It is this layer, for example, that will wholly contain the MVC architecture of a GUI. The presenters, views, and controllers all belong in the interface adapters layer. The models are likely just data structures that are passed from the controllers to the use cases, and then back from the use cases to the presenters and views. No code inward of this circle should know anything at all about the database. If the database is a SQL database, then all SQL should be restricted to this layer—and in particular to the parts of this layer that have to do with the database.

**Frameworks and Drivers**

The outermost layer of the model is generally composed of frameworks and tools such as the database and the web framework. Generally, you don’t write much code in this layer, other than glue code that communicates to the next circle inward. The frameworks and drivers layer is where all the details go. The web is a detail. The database is a detail. We keep these things on the outside where they can do little harm.

\* The circles are intended to be schematic: You may find that you need more than just these four. There’s no rule that says you must always have just these four. However, the Dependency Rule always applies. Source code dependencies always point inward. As you move inward, the level of abstraction and policy increases. The outermost circle consists of low-level concrete details. As you move inward, the software grows more abstract and encapsulates higher-level policies. The innermost circle is the most general and highest level.

**Crossing Boundaries**

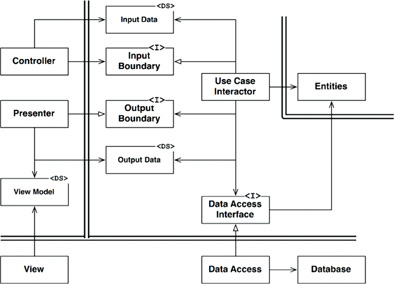
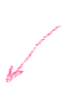
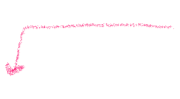
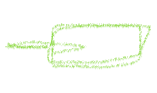
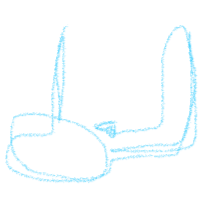
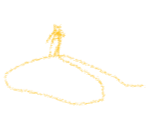
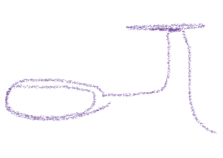
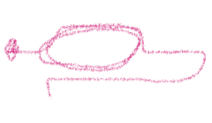
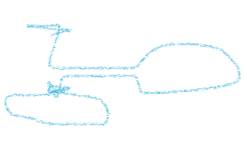
****This shows the controllers and presenters communicating with the use cases in the next layer. Note the flow of control: It begins in the controller, moves through the use case, and then winds up executing in the presenter. Note also the source code dependencies: Each points inward toward the use cases.

We usually resolve this apparent contradiction by using the Dependency Inversion Principle. In a language like Java, for example, we would arrange interfaces and inheritance relationships such that the source code dependencies oppose the flow of control at just the right points across theboundary.

For example, suppose the use case needs to call the presenter. This call must not be direct because that would violate the Dependency Rule: **No name in an outer circle can be mentioned by an inner circle**. So we have the use case call an interface (in image as “use case output port”) in the inner circle, and have the presenter in the outer circle implement it. The same technique is used to cross all the boundaries in the architectures. We take advantage of dynamic polymorphism to create source code dependencies that oppose the flow of control so that we can conform to the Dependency Rule, no matter which direction the flow of control travels.

Which Data crosses the boundaries?

Typically the data that crosses the boundaries consists of simple data structures. You can use basic structs or simple data transfer objects if you like. Or the data can simply be arguments in function calls. Or you can pack it into a hashmap, or construct it into an object. **The important thing is that isolated, simple data structures are passed across the boundaries. We don’t want to cheat and pass Entity objects or database rows.** We don’t want the data structures to have any kind of dependency that violates the Dependency Rule.



Note the directions of the dependencies. All dependencies cross the boundary lines pointing inward, following the Dependency Rule.

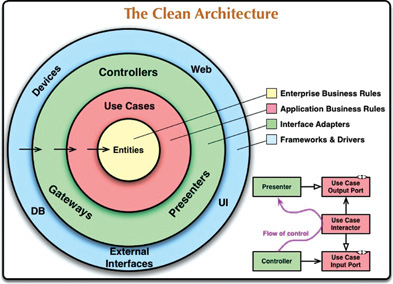
The diagram shows a typical scenario for a web-based Java system using a database. The web server gathers input data from the user and hands it to the Controller on the upper left. The Controller packages that data into a plain old Java object and passes this object through the InputBoundary to the UseCaseInteractor. The UseCaseInteractor interprets that data and uses it to control the dance of the Entities. It also uses the DataAccessInterface to bring the data used by those Entities into memory from the Database. Upon completion, the UseCaseInteractor gathers data from the Entities and constructs the OutputData as another plain old Java object. The OutputData is then passed through the OutputBoundary interface to the Presenter. The job of the Presenter is to repackage the OutputData into viewable form as the ViewModel, which is yet another plain old Java object. The ViewModel contains mostly Strings and flags that the View uses to display the data. Whereas the OutputData may contain Date objects, the Presenter will load the ViewModel with corresponding Strings already formatted properly for the user.

The same is true of Currency objects or any other business-related data. Button and MenuItem names are placed in the ViewModel, as are flags that tell the View whether those Buttons and MenuItems should be gray. This leaves the View with almost nothing to do other than to move the data from the ViewModel into the HTML page.

**Conclusion**

When any of the external parts of the system become obsolete, such as the database, or the web framework, you can replace those obsolete elements with a minimum of fuss.

\* The book and the teacher explain high level a little different



Dependency

Low Level

High Level

Teacher

Low Level

High Level

Dependency

|  |
| --- |
| UI |
| Presenters |
| Services |
| Model Objects |

**Quiz:**

**1 According to the clean architecture, which of the following statements is true?**

Entities have dependencies on databases.

Use cases have dependencies on entities.

Entities have dependencies on use cases.

Use cases have dependencies on Controllers.