# Clean Architecture – Make Your Architecture Scream

February 18, 2018 By Michael Outlaw — 28 Comments

Like 2 Tweet

00:00 00:00

Podcast: Play in new window | Download

Subscribe: Apple Podcasts | Google Podcasts | Spotify | Stitcher | TuneIn | RSS

Michael can't tell higher from lower, Allen puts his views where he wants them, and Joe snaps it to a Slim Jim as we discuss how to make our architectures scream while discussing Robert C. Martin's Clean Architecture.

To read these notes on something other than your podcast player, you can find these show notes in all their big screen glory at https://www.codingblocks.net/episode75.

## Sponsors

- FreshBooks.com/Coding – Use code "CODING BLOCKS" in the "How Did You Hear About Us?" section

## Survey Says ...

With Valentine's Day in mind, we ask: Have you ever made a mix tape?

**Have you ever made a mix tape?**

○ Does a playlist count? Oh, it doesn't? Then no, I haven't.

○ Man, I got my technique down and everything. I know exactly when to play Bon Jovi's five words ... yes I have

vote

## News

- Thank you to everyone that left us a review:
  - iTunes: Tillman32, Satisfied photo printer, Cyclomatic Double Complexity, portugueezy
  - Stitcher: Styx, EdwardDunn, Rosengren
- Clarification: The Output window tip for SQL Server Management Studio is specific to SSMS 2017.
- Joe will be speaking at the Orlando Code Camp March 17th. Visit http://www.orlandocodecamp.com/ to learn more.

- Become the master of your 3d domain! New master's course from Unity3d College (Unity3d College 2018 Master Course) available February 26th. We'll be running a contest giveaway, so join our mailing list to participate.
- Help us out by visiting our sponsors and affiliate resources.

## Make Your Architecture Scream

### Policy and Level

- A computer program is a statement of policy – known inputs and expected outputs
- Most systems have many statements of separate policies
  - Business rules, formatting, ETL, etc
- Policies that change for the same reasons should be grouped into the same components and vice versa
- Goal is to create acyclic dependency graphs where items with similar policies are at the same level, and dependencies are at the edges
  - Direction of dependencies is based on the level of components they connect
    - Level is the distance from the inputs and outputs
    - The further the policy is from the inputs and outputs, the higher the level
- Data flows and source code dependencies do not always point in the same direction
  - Dependencies should be decoupled from data flow but coupled to the level
- The importance of this is the fact that the higher level components are now reusable with different input and output sources
- Policies that change for the same reason or at the same time are bounded by the SRP or CCP
  - Higher level components typically change less frequently than the lower level components
  - Lower level components (those that change frequently) should be plugins to the higher level components

### Business Rules

- *"Business rules are rules or procedures that make or save the business money"* (whether or not they were implemented on a computer)
- Critical business rules – rules critical to the business itself, with or without a computer, for example, calculating interest on a loan.
- Critical business data – critical data that would exist even if there wasn't an automated system.
- Critical business rules and data are tightly bound and therefore a good space for an object, also called…

### Entity

- An object that contains critical business rules and critical business data
- These should be separated from every other concern in the application
  - No dependencies on databases, 3rd party dependencies, user interfaces, etc.
- These objects are pure business.

### Use Cases

- There are additional business rules that are not "critical" – they define how the automated system should work, but would have no impact on a manual business operation
- Use case – description of how an automated system is to be used
  - These indicate how and when a critical business entity should be invoked
  - These also indicate the inputs and outputs but not where they come from (database, UI, etc.)

    - The how of data gets in and out is irrelevant to the use case
- Entities have no knowledge of how use cases use them
  - Follows the Dependency Inversion Principle – higher level components know nothing of the lower level components – the direction is inverted
- Use cases are specific to a single application
- Entities are generalizations

## Request and Response Models

- Use cases accept simple request objects for input and return simple response objects for outputs
  - They SHOULD NOT depend on any frameworks or other dependencies – simple objects
  - Seems like a good idea to return a reference to an entity object – do **NOT** do this
    - Entity objects are higher level components that will change for different reasons so stick to the simple request / response objects
    - Coupling them together violates the Common Closer and Single Responsibility Principles

## So, Business Rules …

- *"Business rules are the reason a software system exists"*
- Should remain pristine
- Are independent and reusable

## Screaming Architecture

- Upon initial inspection, does your application structure scream Spring, or does it scream Healthcare software?
- Architecture should scream the use cases of the application, not the frameworks or dependencies in the application
- *"If your architecture is based on frameworks, then it cannot be based on your use cases"*
- A house's architecture is focused on usability, not whether the house is built of bricks or stucco
  - Again, going back to deferring decisions like that until further down the road
- The "web" is a delivery mechanism – it's your IO layer – much like a mobile app, desktop application or any other

## The Theme and Purpose

- The architecture is all about structures that support the use cases of the application
- Architecture is not a framework nor is it supplied by a framework.
- Good architecture allows you to defer and delay decisions as well as make it easy to change your mind about those decisions.

## Frameworks are Tools, NOT Ways of Life

- Look at frameworks with skepticism – you don't want to adopt the be all end all position
  - They should NOT dictate your application architecture
- How should you use it?
- How should you protect yourself from it?

## Testable Architectures

- If you've done your job right, then unit testing should be easy to do as everything was decoupled properly
  - Entity objects will be plain old objects with no external dependencies

- Use case objects will coordinate the use of entity objects, again with no infrastructure dependencies

## Does your Architecture Scream?

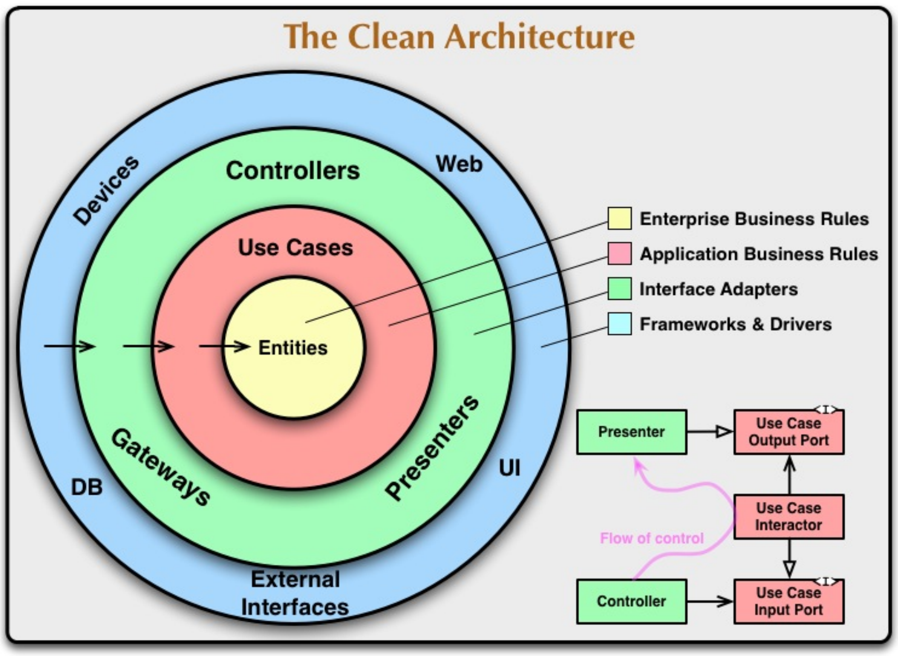
- Your architecture should quickly identify the purpose of your system.

They: *"We see some things that look like models – where are the views and controllers?"*

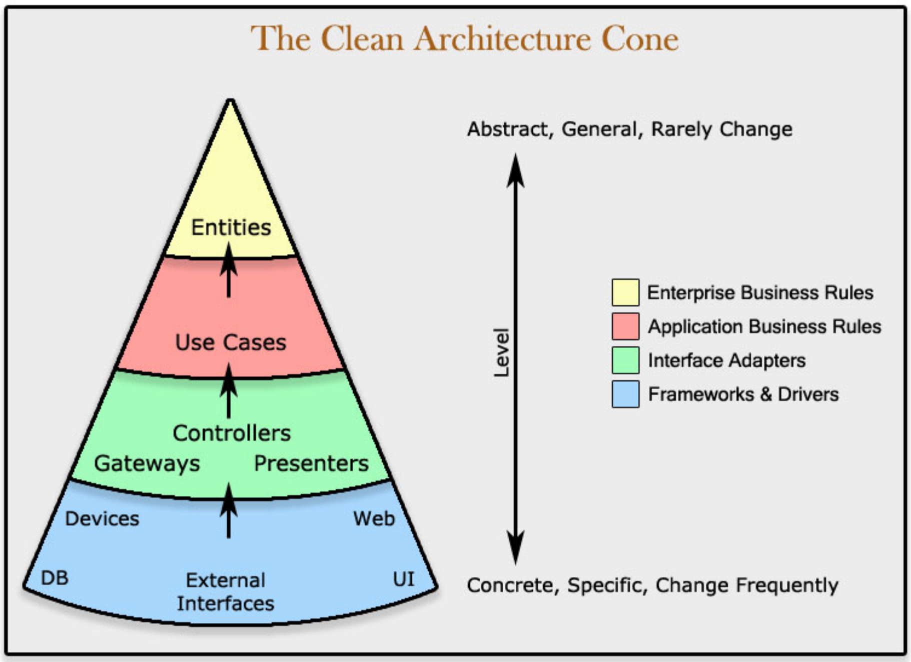
You: *"Oh, those are details that needn't concern us at the moment. We'll decide about them later."*

## The Clean Architecture

Uncle Bob's version of the Clean Architecture diagram:



Our re-imagining of it as a cone:


The Clean Architecture Cone
Entities
Use Cases
Controllers
Gateways
Presenters
Devices
Web
DB
External Interfaces
UI
Abstract, General, Rarely Change
Level
Concrete, Specific, Change Frequently
Enterprise Business Rules
Application Business Rules
Interface Adapters
Frameworks & Drivers

- It's really about the Separation of Concerns
  - Dividing software into layers
- Should be independent of frameworks
- They should be testable
- They should be independent of a UI
- They should be independent of a database
- They should be independent of interfaces to 3rd party dependencies
- The Clean Architecture Diagram
  - Innermost: "Enterprise / Critical Business Rules" – Entities
  - Next out: "Application business rules" – Use Cases
  - Next out: "Interface adapters" – Gateways, Controllers, Presenters
  - Outer: "Frameworks and drivers" – Devices, Web, UI, External Interfaces, DB
- Moving inward, the level of abstraction and policy increase
- The innermost circle is the most general/highest level
- Inner circles are policies
- Outer circles are mechanisms
- Inner circles cannot depend on outer circles
- Outer circles cannot influence inner circles

## Entities

- Entities should be usable by many applications (critical business rules) and should not be impacted by anything other than a change to the critical business rule itself
- They encapsulate the most general/high-level rules.

## Use Cases

- Use cases are application specific business rules
  - Changes should not impact the Entities
  - Changes should not be impacted by infrastructure such as a database
- The use cases orchestrate the flow of data in/out of the Entities and direct the Entities to use their Critical Business Rules to achieve the use case

## Interface Adapters

- Converts data from data layers to use case or entity layers
  - Presenters, views and controllers all belong here
- No code further in (use cases, entities) should have any knowledge of the db

## Frameworks and Drivers

- These are the glue that hook the various layers up
- The infrastructure details live here
- You're not writing much of this code, i.e. we use SQL Server, but we don't write it.

## Crossing Boundaries

- Flow of control went from the controller, through the application use case, then to the presenter
- Source code dependencies point in towards the use cases
- Dependency Inversion Principle
  - Use case needs to call a presenter – doing so would violate the dependency rule – inner circles cannot call (or know about) outer circles….
    - The use case would need to call an interface
      - The implementation of that interface would be provided by the interface adapter layer – this is how the dependency is inverted

- This same type of inversion of control is used all throughout the architecture to invert the flow of control

### Data Crossing Boundaries

- Typically data crossing the boundaries consist of simple data structures
- DO NOT PASS ENTITY OBJECTS OR DATA ROWS!
  - This would violate the dependency rules
- Data is passed in the format that is most convenient to the inner circle / layer
- These are isolated, simple data structures
  - Meaning our DTOs needed to cross the boundaries should belong in the inner circle, or at least their definition (interface, abstract class)

### Conclusion

- Conforming to the rules is not difficult (but requires work) and will set you up to be able to plug and play pieces in the future

## Resources We Like

- Clean Architecture by Robert C. Martin (Amazon)
- Domain-Driven Design Fundamentals by Julie Lerman and Steve Smith (available at Pluralsight)

## Tip of the Week

- Embedding images in email is easy, but make sure you've got the right mime types (jpeg, not jpg – no period!)
- Target a specific csproj in your VS solution in a Team City build configuration like *MyProjectName:Build*. And if your project name contains spaces, target it like *My%%20Project%%20Name:Build*. Thanks to our Slack community for help figuring out the special space encoding: Russ (kritner), Sean (sean), Robert (robert), Martin (azeteg)
- #1. Use a logging framework. #2. Abstract said framework to reduce your dependency on it.
  - Common Logging Abstraction for .NET – https://github.com/net-commons/common-logging
  - SLF4J – https://www.slf4j.org/
  - Serilog – https://serilog.net/
- Read your Kindle books online – https://read.amazon.com/
- Use built-in functionality, such as System.Net.Mail.MailAddress, to validate that a string is an email address.
  - Don't try to write your own! It gets crazy – http://emailregex.com/

**Share the joy**

*Filed Under: Podcasts Tagged With: architecture, best practices, clean architecture, Clean Code, design, uncle bob*

### About Michael Outlaw

Michael Outlaw is a father, husband, and software developer based out of Atlanta, GA. A fan of gadgets and all things technology, when he isn't found behind the clickety clack of his favorite mechanical keyboard he's usually found on a bicycle or tennis court. A father of two boys, Michael often spends his free time teaching his boys how to lose with dignity as they destroy him in game franchises like Overwatch, Halo, and Call of Duty. When not sharing his love of Git, C#, and unit testing, he can be found spending his "study time" studying the mysteries of machine learning.

LinkedIn: linkedin.com/in/michaeloutlaw
Twitter: @iamwaltuo

无法连接到 reCAPTCHA 服务。请检查您的互联网连接，然后重新加载网页以获取 reCAPTCHA 验证。