CSC207 Course Notes (Clean Architecture)

1 What is Design and Architecture?

With software design, the low-level details and the high-level structure are all part of the same whole. They form a continuous fabric that defines the shape of the system. You can't have one without the other, no clear dividing line separates them. There is simply a continuum of decisions from the highest to the lowest levels.

1.1 The Goal

The goal of software architecture is to minimize the human resources required to build and maintain the required system. The measure of design quality is simply the measure of the effort required to meet the needs of the customer. If that effort is low, and stays low throughout the lifetime of the system, the design in good. If that effort grows with each new release, the design is bad.

1.2 Conclusion

In every case, the best option is for the development organization to recognize and avoid its own overconfidence and to start taking the quality of its software architecture seriously. To take software architecture seriously, you need to know what good software architecture is. To build a system with a design and an architecture that minimize effort and maximize productivity, you need to know which attributes of system architecture lead to that end.

2 A Tale of Two Values

Every software system provides two different values to the stakeholders: behaviour and structure. Software developers are responsible for ensuring that both those values remain high.

2.1 Behaviour

The first value of software is its behavior. Programmers are hired to make machines behave in a way that makes or saves money for the stakeholders. We do this by helping the stakeholders develop a functional specification, or requirements document. Then we write the code that causes the stakeholder's machines to satisfy those requirements. When the machine violates those requirements, programmers get their debuggers out and fix the problem.

2.2 Architecture

Software must be easy to change. When the stakeholders change their minds about a feature, that change should be simple and easy to make. The difficulty in making such a change should be proportional only to the scope of the change, and not to the shape of the change.

It is the difference between the scope and shape that often drives the growth in software development costs. It is the reason that costs grow out of proportion to the size of the requested changes. Its is the reason that the first year of development is much cheaper than the second.

From the stakeholders' point of view, they are providing as stream of changes of roughly similar scope. From the developers' point of view, the stakeholders are giving them requests that are harder to fit then the last, because the shape of the system does not match the shape of the request.

The more the architecture prefers one shape over another, the more likely new features will be harder to fit into that structure. Therefore architectures that are shape agnostic are practical.

2.3 The Greater Value

For developers, it is more important for the software system to be easy to change than for the software system to work.

- If you have a program that works perfectly but it is impossible to change, then it won't work when the requirements change, and you won't be able to make it work. Therefore the program will become useless.
- If you have a program that does not work but is easy to change, then you can make it work, and keep it working as requirements change. Therefore the program will remain continually useful.

There are systems that are practically impossible to change, because the cost of change exceeds the benefit of change. Many systems reach that point in some of their features or configurations.

For business managers, it is more important that the software system to work than the software system to be easy to change. Although business managers want to be able to make changes, they may note that the current functionality is more important than any later flexibility. In contrast, if the business managers ask for a change, and the estimated costs for that change are unaffordably high, the business managers will likely be furious that the system was allowed to get to a point where the change was impractical.

2.4 Eisenhower's Matrix

The first value of software–behaviour–is urgent but not always particularly important. The second value of software–architecture–is important but never particularly urgent. Some things are both urgent and important, while others are not urgent and not important. These four couplets can be arranged into priorities:

1. Urgent and important
2. Not urgent and important
3. Urgent and not important
4. Not urgent and not important

Business managers and developers often fail to separate those features that are urgent but not important from those features that truly are urgent and important. This failure then leads to ignoring the important architecture of the system in favour of unimportant features of the system.

2.5 Fight for the Architecture

If architecture comes last, then the system will become ever more costly to develop, and eventually change will become practically impossible for part or all of the system.

3 Paradigm Overview

3.1 Structured Programming

Structured programming imposes discipline on direct transfer of control.

3.2 Object-oriented Programming

Object-orientated programming imposes discipline on direct transfer of control.

3.3 Functional Programming

Functional programming imposes discipline upon assignment.

3.4 Food for Thought

Each of the paradigms removes capabilities from the programmer. None of them adds new capabilities. Each imposes some kind of extra discipline that is negative in its intent. The paradigm tells us what not to do.

3.5 Conclusion

The three paradigms align with the three big concerns of architecture: function, separation of components, and data management.

4 Structured Programming

4.1 Functional Decomposition

Structured programming allows modules to be recursively decomposed into provable units, which in turn means that modules can be functionally decomposed. That is, you can take a large-scale problem statement and decompose it into high-level functions, ad infinitum. Moreover, each of those decomposed functions can be represented using the restricted control structures of structured programming.

4.2 Tests

Structured programming forces us to recursively decompose a program into a set of small provable functions. We can then use tests to try to prove these small provable functions incorrect. If such tests fail to prove incorrectness, then we deem the functions to be correct enough for our purpose.

4.3 Conclusion

At every level, from the smallest function to the largest component, software is like a science and, therefore, is driven by falsifiablility. Software architects strive to define modules, components, and services that are easily falsifiable (testable). To do so, they employ restrictive disciplines similar to structured programming, albeit at a much higher level.

5 Object-Oriented Programming

The nature of object-orientated design (OO) is explained by: encapsulation, inheritance, and polymorphism. The implication is that OO is the proper admixture of these three things, or at least that an OO language must support three things.

5.1 Encapsulation

Encapsulation is cited as part of the definition of OO since OO languages provide easy and effective encapsulation of data and function. As a result, a line can be drawn around a cohesive set of data and functions. Outside of that line, the data is hidden and only some of the functions are known. We see this concept in action as private data members and the public member functions of a class. The idea is not unique to OO. The way encapsulation is partially repaired is by introducing the `public`, `private`, and `protected` keywords into the language. Many OO languages have little or no enforced encapsulation.

5.2 Inheritance

Inheritance is the redeclaration of a group of variables and functions within an enclosing scope.

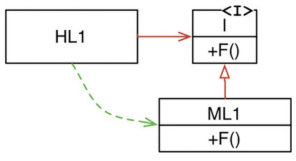
5.3 Polymorphism

The UNIX operating system requires that every IO device driver provide five standard functions: `open`, `close`, `read`, `write`, and `seek`. The signatures of those functions must be identitcal for every IO driver.

Polymorphism is an application of pointers to functions. The problem with explicitly using pointers to functions to create polymorphic behavior is that pointers to functions are dangerous. Such use is driven by a set of manual conventions. You have to remember to follow the convention to initialize those pointers. You have to remember to follow the convention to call your functions through those pointers. If any programmer fails to remember these conventions, the resulting bug can be hard to track down and eliminate. OO imposes discipline on indirect transfer of control.

OO allows plug in architecture to be used anywhere, for anything.

5.4 Dependency Inversion



In the figure above, module `HL1` calls the `F()` function in module `ML1`. This is a source code contrivance since it calls the function through an interface. At runtime, the interface doesn't exist, `HL1` simply calls `F()` within `ML1`.

However, the source code dependency (the inheritance relationship) between `HL1` and the interface `I` points in the opposite direction compared to the flow of control. This is called dependency inversion.

Given that OO languages provide safe and convenient polymorphism means that any source code dependency, no matter where it is, can be inverted.

Software architects working in systems written in OO languages have absolute control over the direction of all source code dependencies in the system. They are not constrained to align those dependencies with the flow of control. No matter which module does the calling and which module is called, the software architect can point the source code dependency in either direction.

When the source code in a component changes, only that component needs to be redeployed. This is independent deployability.

If the modules in a system can be deployed independently, then thy can be developed independently by different teams. This is independent developability.

5.5 Conclusion

To software architects, OO is the ability, through the use of polymorphism, to gain absolute control over every source code dependency in the system. It allows the architect to create a plugin architecture, in which modules that contain high-level policies are independent of modules that contain low-level details. The low-level details are relegated to plugin modules that can be deployed and developed independently from the modules that contain high-level policies.

6 Functional Programming

Variables in functional languages do not vary.

6.1 Immutability and Architecture

Architects must be concerned with the mutability of variables as all race conditions, deadlock conditions, and concurrent update problems are due to mutable variables. You cannot have a race condition or a concurrent update problem if no variable is ever updated. You cannot have deadlocks without mutable locks. All the problems that we face in concurrent applications-all the problems we face in applications that require multiple threads, and multiple processors-cannot happen if there are no mutable variables.

Architects must make sure that the systems they design are robust in the presence of multiple threads and processes.

Immutability can be practicable, if certain compromises are made.

6.2 Segregation of Mutability

One of the most common compromises in regard to immutability is to segregate the application, or the services within the application, into mutable and immutable components. The immutable components perform their tasks in a purely functional way, without using any mutable variables. The immutable components communicate with one or more other components that are not purely functional, and allow for the states of variables to be mutated.

Since mutating state exposes those components to all the problems of concurrency, it is common practice to use some kind of transactional memory to protect the mutable variables from concurrent updates and race conditions. Transactional memory protects those variables with a transaction- or retry-based scheme.

Well-structured applications are segregated into those components that do not mutate variables and those that do. This kind of segregation is supported by the use of appropriate disciplines to protect those mutated variables.

Architects should push as much processing as possible into the immutable components, and drive as much code as possible out of those components that must allow mutation.

6.3 Event Sourcing

Event sourcing is a strategy wherein we store the transactions, but not the state. When state is required, we simply apply all the transactions from the beginning of time.

Since neither updates nor deletions occur in the data store, there cannot be any concurrent update issues.

If we have enough storage and enough processor power, we can make our applications entirely immutable and entirely functional.

6.4 Conclusion

To summarize:

- Structured programming is discipline imposed upon direct transfer of control.
- Object-orientated programming is discipline imposed upon indirect transfer of control.
- Functional programming is discipline imposed upon variable assignment.

Each paradigm restricts some aspect of how we write code, none add to our power or capabilities.

Software is composed of sequence, selection, iteration, and indirection.

7 SRP: The Single Responsibility Principle

The Single Responsibility Principle (SRP) states the best software system is heavily influenced by the social structure of the organization that uses it so that each software module has one, and only one, reason to change. A module should be responsible to one, and only one, group of people who require a change. A module is a source code. However, for languages and development environments that don't use source files to contain their code, a module is a cohesive set of functions and data structures. Cohesive implies SRP. Cohesion is the force that binds together the code responsible to group of people who require a change.

7.1 Conclusion

The Single Responsibility Principle is about functions and classes- but reappears in a different form at two more levels. At the level of components, it becomes the Common Closure Principle. At the architectural level, it becomes the Axis of Change responsible for the creation of Architectural Boundaries.

8 OCP: The Open-Closed Principle

The Open-Closed Principle (OCP) states that for software systems to be easy to change, they must be designed to allow the behavior of those systems to be changed by adding new code, rather than changing existing code. A software artifact should be open for extension but closed for modification. The behaviour of a software artifact should be extendable, without having to modify that artifact.

8.1 A Thought Experiment

Architects separate functionality based on how, why, and when it changes, and then organize that separated functionality into a hierarchy of components. Higher-level components in that hierarchy are protected from the changes made to lower-level components. This is how the OCP works at the architectural level.

8.2 Information Hiding

Transitive dependencies are a violation of the general principle that software entities should not depend on things they don't directly use.

8.3 Conclusion

The goal of the OCP is to make the system easy to extend without incurring a high impact of change. The goal is accomplished by partitioning the system into components, and arranging those components into a dependency hierarchy that protects higher-level components from changes in lower-level components.

9 LSP: The Liskov Substitution Principle

The Liskov Substitution Principle (LSP) states that if for each object `o1` of type `S` there is an object of type `T` such that for all programs `P` defined in terms of `T`, the behaviour of `P` is unchanged when `o1` is substituted for `o2` then `S` is a subtype of `T`.

9.1 Conclusion

The LSP can, and should, be extended to the level of architecture. A simple violation of substitutability, can cause a system's architecture to be polluted with a significant amount of extra mechanisms.

10 ISP: The Interface Segregation Principle

The Interface Segregation Principle (ISP) advises software designers to avoid depending on things that they don't use.

10.1 ISP and Language

Statically typed languages force programmers to create declarations that used must `import`, or `use`, or otherwise `include`. It is these `included` declarations in source code that create the source code dependencies that force recompilation and redeployment.

In dynamically typed languages, such declarations don't exist in source code and are instead inferred at runtime. Thus there are no source code dependencies to force recompilation and redeployment. This is the primary reason that dynamically typed languages create systems that are more flexible and less tightly coupled than statically typed languages.

ISP is a language issue, rather than an architecture issue.

10.2 ISP and Architecture

It is harmful to depend on modules that contain more than you need.

10.3 Conclusion

Depending on something that carries baggage that we don't need can cause us troubles that we didn't expect.

11 DIP: The Dependency Inversion Principle

The Dependency Inversion Principle (DIP) tells us that the most flexible systems are those in which source code dependencies refer only to abstractions, not to concretions.

In a statically typed language, this means that the `use`, `import`, and `include` statements should refer only to source modules containing interfaces, abstract classes, or some kind of abstract declaration. Nothing concrete should be depended on.

In a dynamically typed language, source code dependencies should not refer to any module in which the functions being called are implemented.

We tend to ignore the stable background of operating system and platform facilities when it comes to DIP. We tolerate those concrete dependencies because we know we can rely on them not to change.

It is the volatile concrete elements of our system that we want to avoid depending on. Those are the modules that we are actively developing and that are undergoing frequent change.

11.1 Stable Abstractions

Every change to an abstract interface corresponds to a change to its concrete implementations. Conversely, changes to concrete implementations do not always, or even usually, require changes to the interfaces that they implement. Therefore interfaces are less volatile than implementations.

Good software designers and architects work hard to reduce the volatility of interfaces by trying to find ways to add functionality to implementations without making changes to the interfaces. This is Software Design 101.

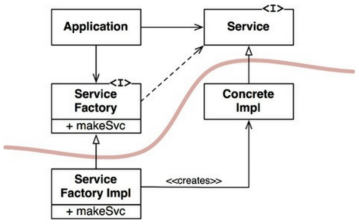
Stable software architectures are those that avoid depending on volatile concretions, and that favor the use of stable abstract interfaces. This implication is summarized by a set of very specific coding practices:

- Don't refer to volatile concrete classes. Refer to abstract interfaces instead. This rule applies to all languages, whether statically or dynamically typed. It also puts severe constraints on the creation of objects and generally enforces the use of Abstract Factories.
- Don't derive from volatile concrete classes. In statically typed languages, inheritance is the strongest, and most rigid, of all the source code relationships; consequently, great care should be used. In dynamically typed languages, inheritance is less of a problem, but it is still a dependency-and caution should be used.
- Don't override concrete functions. Concrete functions often require source code dependencies. When we override those functions, we do not eliminate those dependencies-indeed, we inherit them. To manage those dependencies, we should make the function abstract and create multiple implementations.
- Never mention the name of anything concrete and volatile.

11.2 Factories

To comply with these rules, the creation of volatile concrete objects requires special handling. In all languages, the creation of an object requires a source code dependency on the concrete definition of the object.

In most object-oriented languages, an Abstract Factory is used to manage this dependency. In the diagram



above, the `Application` uses the `ConcreteImp1` through the `Service` interface. However, the `Application` must create instances of the `ConcreteImp1`. To achieve this without creating a source code dependency on the `ConcreteImp1`, the `Application` calls the `makeSvc` method of the `ServiceFactory` interface. This method is implemented by the `ServiceFactoryImp1` class, which derives from `ServiceFactory`. That implementation instantiates the `ConcreteImp1` and returns it as a `Service`.

The curved line in the figure above is an architectural boundary. It seperated the abstract from the concrete. All source code dependencies cross that curved line pointing in the same direction, toward the abstract side.

The curved line divided the system into two components: one abstract and the other concrete. The abstract component contains all the high-level business rules of the application. The concrete component contains all the implementation details that those business rules manipulate.

The flow of control crosses the curved line in the opposite direction of the source code dependencies. The source code dependencies are inverted against the flow of control- which is why the principle is referred as Dependency Inversion.

11.3 Concrete Components

The concrete components in the figure above contains a single dependency, so it violates the DIP. DIP violations cannot be entirely removed, but they can be gathered into a small number of concrete components and kept separate from the rest of the system.

Most systems contains at least one such concrete component-called `main` as it contains the `main` function. In the case illustrated by the figure above, the `main` would instantiate `ServiceFactoryImp1` and place that instance in a global variable of type `ServiceFactory`. The `Application` would then access the factory through that global variable.

11.4 Conclusion

The code that implements high-level policy should not depend on the code that implements low-level details. Rather, details should depend on policies.

12 Components

Components are the units of deployment. They are the smallest entities that can be deployed as part of a system.

Components can be linked together into a single executable. Or they can be aggregated together into a single archive, such a `.war` file. Or they can be independently deployed as separate dynamically loaded plug-ins, such as `.jar` or `.dll` or `.exe` files. Regardless of how they eventually deployed, well-designed components always retain the ability to be independently deployable and, therefore, independently developable.

12.1 Conclusion

Dynamically linked files, which can be plugged together at runtime, are the software components of our architecture.

13 Component Cohesion

13.1 The Reuse/Release Equivalence Principle

From a software design and architecture point of view, the Reuse/Release Equivalence Principle (REP) means that the classes and modules that are formed into a component must belong to a cohesive group. The component cannot contain a random collection of classes and modules; instead, there must be some overarching theme or purpose that those modules all share.

13.2 The Common Closure Principle

The Common Closure Principle (CCP) says to gather into components those classes that change for the same reasons and at the same time and separate into different components those classes that change at different times and for different reasons. In other words, the CCP says that a component should not have multiple reasons to change.

For most applications, maintainability is more important than reusability. If the code in an application must change, one would rather that all of the changes occur in one component, rather than being disturbed across many components. If changes are confined to a single component, then we need to redeploy only the one changed component. Other components that don't depend on the changed component do not need to be recalculated or redeployed.

The CCP says to gather together in one place all the classes that are likely to change for the same reasons, If two classes are so tightly bound, either physically or conceptually, that they always change together, then they belong in the same component. This minimizes the workload related to releasing, revalidating, and redeploying the software.

We define our classes such that they are closed to the most common kids of changes that we expect or have experienced.

The CCP says that by gathering together into the same component those classes that are closed to the same types of changes, when a change in requirements comes along, that change has a good chance of being restricted to a minimal number of components.

13.2.1 Similarity with SRP

The CCP is the component form of the SRP. The SRP tells us to separate methods into different class, if they change for different reasons. The CCP tells us to separate classes into different components, if they change for different reasons. The CCP and SRP can be summarized by the following: Gather together those things that change at the same times and for the same reasons and separate those things that change at different times or for different reasons.

13.3 The Common Reuse Principle

The Common Reuse Principle (CRP) states that classes and modules that tend to be reused together belong in the same component. The principle helps us to decide which classes and modules should be placed into a component.

Reusable classes collaborate with other classes that are part of the reusable abstraction. The CRP states that these classes belong together in the same component. In such a component, classes should have lots of dependencies on each other.

A simple example is a container class and its associated iterators. These classes are reused together because they are tightly couples to each other. Thus they must be in the same component.

The CRP tells us which class not to keep together in a component. When one component uses another, a dependency is created between the components. Even if the using component uses only one class within the used, it still doesn't weaken the dependency. The using component still depends on the used component.

Because of that dependency, every time the used component is changed, the using component will likely need corresponding changes. Even if no changes are necessary to the using component, it will likely still need to be recompiled, revalidated, and redeployed. This is true even if the using component doesn't care about the change made in the used component.

When we depend on a component, we want to make sure we depend on every class in that component. In other words, we want to make sure that the classes that we put into a component are inseparable-that it is impossible to depend on some and not on the others. Otherwise, we will redeploying more components than is necessary, and wasting significant effort.

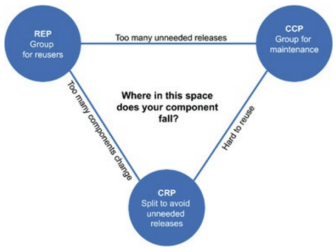
The CRP tells us more about which classes shouldn't be together than about which classes should be together. The CRP says that classes that are not tightly bound to each other should not be in the same component.

13.3.1 Relation to ISP

The CRP is the generic version of the ISP. The ISP advises us not depend on classes that have methods we don't use. The CRP advises us not to depend on components that have classes we don't use.

13.4 The Tension Diagram for Component Cohesion

The REP and CCP are inclusive principles: Both tend to make components larger. The CRP is an exclusive principle, driving components to be smaller. It is the tension between these principles that good architects seek to resolve. The figure above is a tension diagram that shows how the three principles of cohesion interact



with each other. The edges of the diagram describe the cost of abandoning the principle on the opposite vertex.

An architect who focuses on just the REP and CRP will find that too many components are impacted

when simple changes are made. In contrast, an architect who focuses too strongly on the CCP and REP will cause too many unneeded releases to be generated.

Generally, projects tend to start on the right hand side of the triangle, where the only sacrifice is reuse. As the project matures, and other projects begin to draw from it, the project will slide over to the left. This means that component structure of a project can vary with time and maturity. It has more to do with the way that project is developed and used, than with what the project actually does.

13.5 Conclusion

In choosing the classes to group together into components, we must consider the opposing forces involved in reusability and develop-ability. Balancing these forces with the needs of the application is nontrivial. Moreover, the balance is almost always dynamic. That is, the partitioning that is appropriate right now might not be appropriate later. As a consequence, the composition of the components will likely shift and evolve with time as the focus of the project changes from develop-ability to reusability.

14 Component Coupling

14.1 The Acyclic Dependencies Principle

14.1.1 The Weekly Build

The weekly build advises developers ignore each other for the first four days of the week to work on private copies of the code, and not to worry about integrating their work on a collective basis. Then, on the last day, the developers integrate all their changes and build the system.

However, as the duty cycle of development versus integration decreases, the efficiency of the team decreases. To maintain efficiency, the build schedule continually lengthens- but lengthening the build schedule increases project risks. Integration and testing becomes difficult, and the team loses the benefit of rapid feedback.

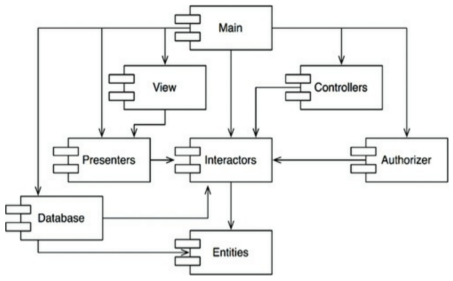
14.1.2 Eliminating Dependency Cycles

The solution to the problems associated with weekly build is to partition the development environment into reasonable components. The components become units of work that can be the responsibility of a single developer, or a team of developers. When developers get a components working, they release it for use by the other developers. They give it a release number and move it into a directory for other teams to use. They then continue to modify their component in their own private areas. Everyone else uses the released version.

As new releases of a component are made available, other teams can decide whether they will immediately adopt the new release. If they decide not to, they simply continue using the old release. Once they decide that they are ready, they begin to use the new release.

Thus no team is dependent on each other. Changes made to one component do not need to have an immediate affect on other teams. Each team can decide for itself when to adapt its own components. Moreover, integration happens in small increments.

To make it work successfully, groups must manage the dependency the dependent structure of the components. The figure above shows a structure of components assembled into an application. Note that the



structure is a directed graph. The components are the nodes, and the dependency relationships are the directed edges. Furthermore, regardless of which component we begin at, it is impossible to follow the dependency relationships and wind up back at that component. This structure has no cycles. It is a directed acyclic graph (DAG).

When the team responsible for `Presenters` makes a new release of their component, we find that `View` and `Main` will both be affected by following the dependency arrows backward. The developers currently working on those components must decide when to integrate their work with the new release of `Presenters`.

When `Main` is released, it has no effect on any other components in the system so the impact of releasing `Main` is small.

When the developers working on the `Presenters` component run a test of that component, they just need to build their version of `Presenters` with the versions of the `Interactors` and `Entities` components that they are currently using. None of the other components in the system need be involved.

When it is time to release the whole system, the process proceeds from the bottom up. First the `Entities` component is compiled, tested, and released. Then the same is done for `Database` and `Interactors`. These components are followed by `Presenters`, `View`, `Controllers`, and then `Authorizer`. `Main` goes last.

14.1.3 The Effect of a Cycle in the Component Dependency Graph

Cycles make it very difficult to isolate components. Unit testing and releasing become very difficult and error prone. In addition, build issues grow geometrically with the number of modules.

Moreover, When there are cycles in the dependency graph, it can be very difficult to work out the order in which components are built. Often, there is no correct order.

14.1.4 Breaking the Cycle

It is always possible to break a cycle of components and reinstate the dependency graph as a DAG. There are two primary mechanisms for doing so:

1. Apply the Dependency Inversion Principle (DIP).
2. Create a new component that the components in the cycle depend on. Move the class(es) that they both depend on into that new component.

14.1.5 The "Jitters"

As the application grow , the component dependency structure jitters and grows. Thus the dependency structure must always be monitored for cycles. When cycles occur, they must be broken somehow. Sometimes this will mean creating new components, making the dependency structure grow.

14.2 Top-Down Design

The component structure cannot be designed from the the top down. Component dependent diagrams are a map to the buildability and maintainability of the application. The component dependency structure grows and evolves with the logical with the logical design of the system.

14.3 The Stable Dependencies Principle

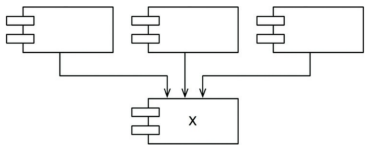
Designs cannot be completely static. Some volatility is necessary if the design is to be maintained. By conforming to the Common Closure Principle (CCP), we create components that are sensitive to certain

kinds of changes but immune to others. Some of these components are designed to be volatile. We expect them to change.

Any component that we expect to be volatile should not be depended on by a component that is difficult to change. Otherwise, the volatile component will also be difficult to change.

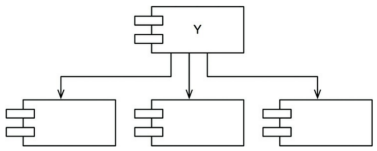
By conforming to the Stable Dependencies Principle (SDP), we ensure that modules that are intended to be easy to change are not dependent on by modules that are harder to change.

14.3.1 Stability



In the figure above, `x` is a stable component. Three components depend on `x`, so it has three reasons not to change. `x` is responsible to three components. Conversely, `x` depends on nothing, so it has no external influence to make it change. `x` is independent.

14.3.2 Stability



In the figure above, `y` is a very unstable component. No other component depends on `y`, so we say that it is irresponsible. `y` also gas three components that it depends on, so changes may come from three external sources. `y` is dependent.

14.3.3 Stability Metrics

One way to measure stability is to count the number of dependencies that enter and leave that component. These counts will allow us to calculate the positional stability of the component.

- Fan-in: Incoming dependencies. This metric identifies the number of classes outside this component that depend on classes within the component.
- Fan-out: Outgoing dependencies. This metric identifies the number of classes inside this component that depend on classes outside the component.
- I: Instability: I = Fat-out $\div$ (Fan-in + Fan-out). This metric has the range $[0, 1]$. I = 0 indicates a maximally stable component. I = 1 indicates a maximally unstable component.

The Fan-in and Fan-out metrics are calculated by counting the number of classes outside the component in question that have dependencies with the classes inside the component in question.

When the I metric is equal to 1, it means that no other component depends on this component (Fan-in = 0), and this component depends on other components (Fan-out ¿ 0). This situation is as unstable as a component can get; it is irresponsible and dependent. Its lack of dependents gives the component no reason not to change, and the components that it depends on mat give it ample reason to change.

When the I metric is equal to 0, it means that component is depended on by other components (Fan-in ¿ 0), but does not itself depend on any other components (Fan-out = 0). Such a component is responsible and independent. It is as stable as it can get. Its dependents make it hard to change the component, and it has no dependencies that might force it to change.

The SDP says that the I metric of a component should be larger than the I metrics of the components that it depends on. That is, I metrics should decrease in the direction of dependency.

14.3.4 Not all Components should be Stable

If all the components in a system were maximally stable, the system would be unchangeable. This is not a desireable situation. We want to design out component structure so that some components are unstable and some are stable.

In an ideal configuration diagram, the changeable components are on top and depend on the stable component at the bottom. Putting the unstable components at the top of the diagram is a useful convention because any arrow that points up is violating the SDP and ADP.

14.3.4.1 Abstract Components

Abstract components are very stable and, therefore, are ideal targets for less stable components to depend on.

14.4 The Stable Abstractions Principle

A component should be as abstract as it is stable.

14.4.1 Where do we put High-Level Policy?

Some software in the system should not change very often. This software represents high-level architecture and policy decisions. Business and architectural designs should not be volatile. Thus the software that encapsulates the high-level policies of the system should be places into stable components (I = 0). Unstable components (I = 1) should contain only the software that is volatile-software that we want to be able to quickly and easily change.

14.4.2 Introducing the Stable Abstractions Principle

The Stable Abstraction Principle (SAP) sets up a relationship between stability and abstractness. The SAP says that a stable component should also be abstract so that its stability does not prevent it from being extended and that an unstable component should be concrete since it its instability allows the concrete code within it to be easily changed.

Thus, if a component is to be stable, it should consist of interfaces and abstract classes so that it can be extended. Stable components that are extensible are flexible and do not overly constrain the architecture.

The SAP and the SDP combined amount to the DIP for components since the SDP says that dependencies should run in the direction of stability, and the SAP says that stability implies abstraction. Thus dependencies run in the direction of abstraction.

The DIP is a principle that deals with classes. Either a class is abstract or it is not. The combination of the SDP and the SAP deals with components, and allows that a component can be partially abstract or partially stable.

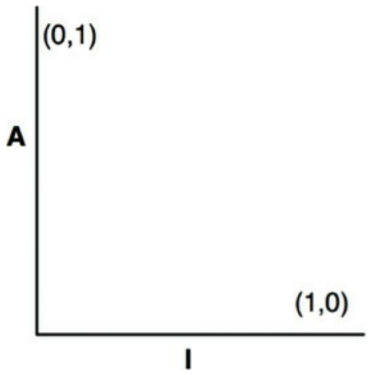
14.4.3 Measuring Abstraction

The A metric is a measure of the abstractness of a component. Its value is the ratio of interfaces and abstract classes in a component to the total number of classes in the component.

- Nc: The number of classes in the component.
- Na: The number of abstract classes and interfaces in the component.
- A: Abstractness. A = Na ÷ Nc.

The A metric ranges form 0 to 1. An A value of 0 implies that the component has no abstract classes at all. A value of 1 implies that the component contains nothing but abstract classes.

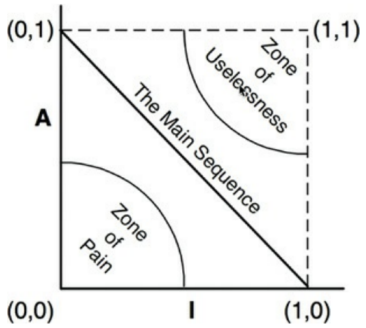
14.4.4 The Main Sequence



In the graph above, A is on the vertical axis and I is on the horizontal axis. Components that are maximally stable and abstract are at the upper left of the graph at (0, 1). The components that are maximally unstable and concrete are at the lower right of the graph at (1, 0).

Not all components fall into one of these two positions, because components often have degrees of abstraction and stability.

Since we cannot enforce a rule that all components sit at either (0, 1) or (1, 0), we must assume that there is a locus of points on the A/I graph that defines reasonable positions for components. We infer what the locus is by finding the areas where components should not be-in other words, by determining the zones of exclusion (see figure below).



14.4.4.1 The Zone of Pain

A component in the area of (0, 0) is highly stable and concrete component. Such a component is not desireable because it is rigid. It cannot be extended because it is not abstract, and it is very difficult to change because of its stability. Thus we do not normally expect to see well-designed component sitting near (0, 0). The area around (0, 0) is a zone of exclusion called the Zone of Pain.

Nonvolatile components are harmless sin the (0, 0) zone since they are not likely to be changed. For that reason, it is only volatile software components that are problematic in the Zone of Pain. The more volatile a component in the Zone of Pain, the more painful it is. Volatility can be considered as the third axis of the graph. Thus, the graph above only shows the most painful plane, where volatility = 1.

14.4.4.2 The Zone of Uselessness

A component near (1, 1) is undesirable because it is maximally abstract, yet has no dependents. Such components are useless. Thus this area is called the Zone of Uselessness.

The software entities that inhabit this region are a kind of detritus. They are often leftover abstract classes that no one ever implemented.

A component that has a position deep position deep within the Zone of Uselessness must contain a significant fraction of such entities. The presence of such useless entities is undesirable.

14.4.5 Avoiding the Zones of Exclusion

Most volatile components should be kept as far from both zones of exclusion as possible. The locus of points that are maximally distant drom each zone is the line that connects (1, 0) and (0, 1). We call this line the Main Sequence.

A sequence that sits on the Main Sequence is not too abstract for its stability, nor is it too unstable for its abstractness. It is neither useless nor particularly painful. It is depended on to the extent that it is abstract, and it depends on to the the extent that it is concrete.

The most desireable position for a component is at one of the two endpoints if the Main Sequence. Good architects strive to position the majority of their components at those endpoints. However, some small fraction of the components in a large system are neither perfectly abstract nor perfectly stable. Those components have the best characteristics if they are on, or close, to the Main Sequence.

14.4.6 Distance from the Main Sequence

If it is desireable for components to be on, or close, to the Main Sequence, then a metric measures how far away a component is from this ideal.

- D: Distance. $D = |A + I - 1|$. The range of this metric is $[0, 1]$, A value of 0 indicates that the component is directly on the Main Sequence. A value of 1 indicates that the component is as far away possible from the Main Sequence.

Given this metric, a design can be analyzed for its overall conformance to the Main Sequence. The D metric for each component can be calculated. Any component that has a D value that is not near zero can be reexamined and restructured.

Statistical analysis of a design is also possible. We can calculate the mean and variance of all the D metrics for the components within a design. We expect a conforming design to have a mean and variance that are close to zero. The variance can be used to establish control limits so as to identify components that are exceptional in comparison to all others.

Another way to use the metrics is to plot the D metric of a component over time.

14.5 Conclusion

Certain dependencies are good and others are bad.

15 What Is Architecture?

The architecture of a software system is the shape given to that system by those who build it. The form of that shape is in the division of that system into components, the arrangement of those components, and the ways in which those components communicate with each other.

The purpose of that shape is to facilitate development, deployment, operation, and maintenance of the software system contained within in. The strategy behind that facilitation is to leave as many options as possible, for as long as possible.

The primary purpose of architecture is to support the life cycle of the system. Good architecture makes the system easy to understand, easy to develop, easy to maintain, and easy to deploy. The ultimate goal is to minimize the lifetime cost of the system and to maximize programmer productivity.

15.1 Development

A software system that is hard to develop is not likely to have a long and healthy lifetime. So the architecture of a system should make that system easy to develop, for the team(s) who develop it.

15.2 Deployment

To be effective, a software system should be deployable. The higher the cost of deployment, the less useful the system is. A goal of a software architecture is to make a system that can be easily deployed with a single action.

15.3 Operation

The architecture of a system makes the operation of the system readily apparent to the developers. Architecture should reveal operation. The architecture of the system should elevate the use cases, the features, and the required behaviors of the system to first-class entities that are visible landmarks for the developers. This simplifies the understanding of the system and greatly aids in development and maintenance.

15.4 Maintenance

Maintenance is the most costly aspect of a software system. The primary cost of maintenance is in spelunking and risk. Spelunking is the cost of digging through the existing software, trying to determine the best place and the best strategy to add a new feature or to repair a defect. While making such changes, the likelihood of creating inadvertent defects is always there, adding to the cost of risk.

A carefully thought-through architecture vastly mitigates these costs. By separating the system into components, and isolating those components through stable interfaces, it is possible to illuminate the pathways for future features and greatly reduce the risk of inadvertent breakage.

16 Policy and Level

Software systems are statements of policy. Indeed, at its core, that's all a computer program actually is. A computer program is a detailed description of the policy by which inputs are transformed into outputs.

In most nontrivial systems, that policy can be broken down into many different smaller statements of policy. Some of those statements will describe how particular business rules are to be calculated. Others will describe how certain reports are to be formatted. Still others will describe how input data are to be validated.

Part of the art of developing a software architecture is carefully separating those policies from one another, and regrouping them based on the ways that they change. Policies that change for the same reasons, and at the same times, are at the same level and belong together in the same component. Policies that change for different reasons, or at different times, are at different levels and should be separated into different components.

The art of architecture often involves forming the regrouped components into a directed acyclic graph. The nodes of the graph are the components that contain policies at the same level. The directed edges are the dependencies between those components. They connect components that are at different levels.

In a good architecture, the direction of those dependencies is based on the level of the components that they connect. In every case, low-level components are designed so that they depend on high-level components.

16.1 Level

A strict definition of "level" is "the distance from the inputs and outputs." The farther a policy is from both the inputs and the outputs of the system, the higher its level. The policies that manage input and output are the lowest-level policies in the system.

Note that the data flows and the source code dependencies do not always point in the same direction. Source code dependencies should be decoupled from data flow and coupled to level.

Recall that policies are grouped into components based on the way that they change. Policies that change for the same reasons and at the same times are grouped together by the SRP and CCP. Higher-level policies—those that are farthest from the inputs and outputs—tend to change less frequently, and for more important reasons, than lower-level policies. Lower-level policies—those that are closest to the inputs and outputs tend to change frequently, and with more urgency, but for less important reasons.

Another way to look at this issue is to note that lower-level components should be plugins to the higher-level components. The component diagram below shows this arrangement. The `Encryption` component knows nothing of the *IODevices* component; the *IODevices* component depends on the `Encryption` component.

17 Business Rules

Business rules are rules or procedures that make or save the business money. In more detail, these rules would make or save the business money irrespective of whether they were implemented on a computer. They would make or save money even if they were executed manually.

These rules known as Critical Business Rules, are critical to the business itself, and would even exist if there were no system to automate them. Critical Business Rules usually require some data to work with.

This data is known as Critical Business Data, and is the data that would exist even if the system were not automated.

The critical rules and critical data are inextricably bound and are known collectively as an Entity object.

17.1 Entities

An Entity is an object within our computer system that embodies a small set of critical business rules operating on Critical Business Data. The Entity object either contains the Critical Business Data or has very easy access to that data. The interface of the Entity consists of the functions that implement the Critical Business Rules that operate on that data.

When we create this kind of class, we are gathering together the software that implements a concept that is critical to the business, and separating it from every other concern in the automated system we are building. This class stands alone as a representative of the business. It is unsullied with concerns about databases, user interfaces, or third-party frameworks. It could serve the business in any system, irrespective of how that system was presented, or how the data was stored, or how the computers in that system were arranged. The Entity is pure business and nothing else.

17.2 Use Cases

Not all business rules are as pure as Entities. Some business rules make or save money for the business by defining and constraining the way that an automated system operates. These rules would not be used in a manual environment, because they make sense only as part of an automated system.

A use case is a description of the way that an automated system is used. It specifies the input to be provided by the user, the output to be returned to the user, and the processing steps involved in producing that output. A use case describes application-specific business rules as opposed to the Critical Business Rules within the Entities.

Use cases contain the rules that specify how and when the Critical Business Rules within the Entities are invoked. Use cases control the dance of the Entities.

Use cases do not describe the user interface other than to informally specify the data coming in from that interface, and the data going back out through that interface. From the use case, it is impossible to tell whether the application is delivered on the web, or on a thick client, or on a console, or is a pure service.

This is very important. Use cases do not describe how the system appears to the user. Instead, they describe the application-specific rules that govern the interaction between the users and the Entities. How

the data gets in and out of the system is irrelevant to the use cases.

A use case is an object. It has one or more functions that implement the application specific business rules. It also has data elements that include the input data, the output data, and the references to the appropriate Entities with which it interacts.

Entities have no knowledge of the use cases that control them. This is another example of the direction of the dependencies following the Dependency Inversion Principle. High-level concepts, such as Entities, know nothing of lower-level concepts, such as use cases. Instead, the lower-level use cases know about the higher-level Entities.

Entities are high level and use cases are low level because use cases are specific to a single application and, therefore, are closer to the inputs and outputs of that system. Entities are generalizations that can be used in many different applications, so they are farther from the inputs and outputs of the system. Use cases depend on Entities; Entities do not depend on use cases.

17.3 Request and Response Models

Use cases expect input data, and they produce output data. However, a well-formed use case object should have no inkling about the way that data is communicated to the user, or to any other component.

The use case class accepts simple request data structures for its input, and returns simple response data structures as its output. These data structures are not dependent on anything.

This lack of dependencies is critical. If the request and response models are not independent, then the use cases that depend on them will be indirectly bound to whatever dependencies the models carry with them.

17.4 Conclusion

Business rules are the reason a software system exists. They carry the code that makes, or saves, money. The business rules should remain pristine, unsullied by baser concerns such as the user interface or database used. Ideally, the code that represents the business rules should be the heart of the system, with lesser concerns being plugged in to them. The business rules should be the most independent and reusable code in the system.

18 Screaming Architecture

18.1 The Theme of an Architecture

Software architectures are structures that support the uses cases of the system. Architectures are not (or should not be) about frameworks. Architectures should not be supplied by frameworks. Frameworks are tools to be used, not architectures to be conformed to.

18.2 The Purpose of an Architecture

Good architectures are centered on use cases so that architects can safely describe the structures that support those use cases without committing to frameworks, tools, and environments.

A good software architecture allows decisions about frameworks, databases, web servers, and other environmental issues and tools to be deferred and delayed. Frameworks are options to be left open.

18.3 But what about the Web?

The web is a delivery mechanism an IO device—and application architectures should treat it as such. The fact that your application is delivered over the web is a detail and should not dominate your system structure. Indeed, the decision of how the application will be delivered over the web is one that should be deferred. A system architecture should be as ignorant as possible about how it will be delivered. It should be able to be delivered as a console app, or a web app, or a thick client app, or even a web service app, without undue complication or change to the fundamental architecture.

18.4 Testable Architectures

If a system architecture is all about the use cases, and if frameworks are kept at arm's length, then we should be able to unit-test all those use cases without any of the frameworks in place. The Entity objects should be plain old objects that have no dependencies on frameworks or databases or other complications. The use case objects should coordinate your Entity objects. Finally, all of them together should be testable in position, without any of the complications of frameworks.

18.5 Conclusion

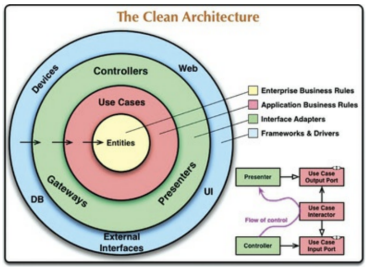
Architectures should tell readers about the system, not about the frameworks used in the system.

19 The Clean Architecture

Each 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 us to use such frameworks as tools, rather than forcing us to conform to the 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. Your business rules are not bound to the database.
- Independent of any external agency. Business rules don't know anything at all about the interfaces to the outside world.

The diagram below illustrates this into a single actionable idea.



19.1 The Dependency Rule

The concentric circles in the figure above 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.

19.1.1 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.

If there is no enterprise and we are just writing a single application, then these entities are the business objects of the application. They encapsulate the most general and high-level rules. They are the least likely to change when something external changes. No operational change to any particular application should affect the entity layer.

19.1.2 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.

Changes in this layer should not affect the entities. The layer should also not 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.

Changes to the operation of the application will affect the use cases and, therefore, the software in this layer. If the details of a use case change, then some code in this layer will certainly be affected.

19.1.3 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.

Similarly, data is converted, in this layer, from the form most convenient for entities and use cases, to the form most convenient for whatever persistence framework is being used (i.e., the database). No code inward of this circle should know anything at all about the database.

Also in this layer is any other adapter necessary to convert data from some external form, such as an external service, to the internal form used by the use cases and entities.

19.1.4 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.

19.1.5 Only Four Circles?

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.

19.1.6 Crossing Boundaries

The flow of control begins in the controller, moves through the use case, and then winds up executing in the presenter. The source code dependencies point inward toward the use cases.

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.

19.1.7 Which Data crosses the Boundaries

Typically the data that crosses the boundaries consists of simple data structures. The data pass across a boundary is always in the form that is most convenient for the inner circle.

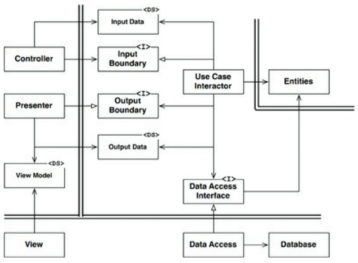
19.2 A Typical Scenario

The diagram below shows a typical scenario for a web-based 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 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 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 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.

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



19.3 Conclusion

Conforming to these simple rules is not difficult, and it will save time going forward. By separating the software into layers and conforming to the Dependency Rule, we create a system that is intrinsically testable, with all the benefits that implies. 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.

20 Presenters and Humble Objects

20.1 The Humble Object Pattern

The Humble Object Pattern is a design pattern that was originally identified as a way to help unit testers to separate behaviors that are hard to test from behaviors that are easy to test. The idea is very simple: Split the behaviors into two modules or classes. One of those modules is humble; it contains all the hard-to-test behaviors stripped down to their barest essence. The other module contains all the testable behaviors that were stripped out of the humble object.

20.2 Presenters and Views

The View is the humble object that is hard to test. The code in this object is kept as simple as possible. It moves data into the GUI but does not process that data.

The Presenter is the testable object. Its job is to accept data from the application and format it for presentation so that the View can simply move it to the screen.

Every button on the screen will have a name. That name will be a string in the View Model, placed there by the presenter. If those buttons should be grayed out, the Presenter will set an appropriate boolean flag in the View model. Every menu item name is a string in the View model, loaded by the Presenter. The names for every radio button, check box, and text field are loaded, by the Presenter, into appropriate strings and booleans in the View model. Tables of numbers that should be displayed on the screen are loaded, by the Presenter, into tables of properly formatted strings in the View model.

Anything and everything that appears on the screen, and that the application has some kind of control over, is represented in the View Model as a string, or a boolean, or an enum. Nothing is left for the View to do other than to load the data from the View Model into the screen. Thus the View is humble.

20.3 Database Gateways

Between the use case interactors and the database are the database gateways. These gateways are polymorphic interfaces that contain methods for every create, read, update, or delete operation that can be performed by the application on the database.

Recall that we do not allow SQL in the use cases layer; instead, we use gateway interfaces that have appropriate methods. Those gateways are implemented by classes in the database layer. That implementation is the humble object. It simply uses SQL, or whatever the interface to the database is, to access the data required by each of the methods. The interactors, in contrast, are not humble because they encapsulate application-specific business rules. Although they are not humble, those interactors are testable, because the gateways can be replaced with appropriate stubs and test-doubles.

20.4 Data Mappers

ORMs also known as "data mappers" form another kind of Humble Object boundary between the gateway interfaces and the database.

20.5 Service Listeners

The application will load data into simple data structures and then pass those structures across the boundary to modules that properly format the data and send it to external services. On the input side, the service listeners will receive data from the service interface and format it into a simple data structure that can be used by the application. That data structure is then passed across the service boundary.

20.6 Conclusion

At each architectural boundary, the Humble Object pattern will be present. The communication across that boundary will almost always involve some kind of simple data structure, and the boundary will frequently divide something that is hard to test from something that is easy to test. The use of this pattern at architectural boundaries vastly increases the testability of the entire system.

21 Partial Boundaries

21.1 Skip the Last Step

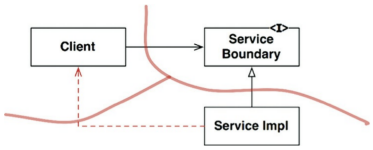
One way to construct a partial boundary is to do all the work necessary to create independently compilable and deployable components, and then simply keep them together in the same component. The reciprocal interfaces are there, the input/output data structures are there, and everything is all set up—but we compile and deploy all of them as a single component.

This kind of partial boundary requires the same amount of code and preparatory design work as a full boundary. However, it does not require the administration of multiple components.

21.2 One-Dimensional Boundaries

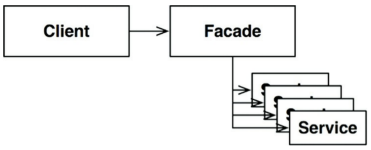
The full-fledged architectural boundary uses reciprocal boundary interfaces to maintain isolation in both directions. Maintaining separation in both directions is expensive both in initial setup and in ongoing maintenance.

A simpler structure that serves to hold the place for later extension to a full-fledged boundary is shown below. It exemplifies the traditional Strategy pattern. A `ServiceBoundary` interface is used by clients and implemented by `ServiceImpl` classes. The necessary dependency inversion is in place in an attempt to isolate the Client from the `ServiceImpl`. It should also be clear that the separation can degrade pretty rapidly, as shown by the nasty dotted arrow in the diagram. Without reciprocal interfaces, nothing prevents this kind of `backchannel` other than the diligence and discipline of the developers and architects.



21.3 Facades

An even simpler boundary is the Facade pattern, illustrated below. In this case, even the dependency inversion is sacrificed. The boundary is simply defined by the Facade class, which lists all the services as methods, and deploys the service calls to classes that the client is not supposed to access. Note, however, that the `Client` has a transitive dependency on all those service classes. In static languages, a change to the source code in one of the Service classes will force the `Client` to recompile. Also, you can imagine how easy backchannels are to create with this structure.



21.4 Conclusion

Each approach has its own set of costs and benefits. Each is appropriate, in certain contexts, as a placeholder for an eventual full-fledged boundary. Each can also be degraded if that boundary never materializes.

It is one of the functions of an architect to decide where an architectural boundary might one day exist, and whether to fully or partially implement that boundary.

22 The Test Boundary

22.1 Tests as System Components

From an architectural point of view, all tests are the same. Tests, by their very nature, follow the Dependency Rule, they are very detailed and concrete; and they always depend inward toward the code being tested. In fact, we can think of the tests as the outermost circle in the architecture. Nothing within the system depends on the tests, and the tests always depend inward on the components of the system.

Tests are also independently deployable. In fact, most of the time they are deployed in test systems, rather than in production systems. So, even in systems where independent deployment is not otherwise necessary, the tests will still be independently deployed.

Tests are most isolated system component. They are not necessary for system operation. No user depends on them. Their role is to support development, not operation. And yet, they are no less a system component than any other. In fact, in many ways they represent the model that all other system components should follow.

22.2 Design for Testability

The extreme isolation of the tests, combined with the fact that they are not usually deployed, often causes developers to think that tests fall outside of the design of the system. This is a catastrophic point of view. Tests that are not well integrated into the design of the system tend to be fragile, and they make the system rigid and difficult to change.

The issue, of course, is coupling. Tests that are strongly coupled to the system must change along with the system. Even the most trivial change to a system component can cause many coupled tests to break or require changes.

This situation can become acute. Changes to common system components can cause many tests to break. This is known as the Fragile Tests Problem. Fragile tests often have the perverse effect of making the system rigid. When developers realize that simple changes to the system can cause massive test failures, they may resist making those changes.

The solution is to design for testability. The first rule of software design—whether for testability or for any other reason—is always the same: Don't depend on volatile things. GUIs are volatile. Test suites that operate the system through the GUI must be fragile. Therefore design the system, and the tests, so that business rules can be tested without using the GUI.

22.3 The Testing API

The way to accomplish this goal is to create a specific API that the tests can use to verify all the business rules. This API should have superpowers that allow the tests to avoid security constraints, bypass expensive resources (such as databases), and force the system into particular testable states. This API will be a superset of the suite of interactors and interface adapters that are used by the user interface.

The purpose of the testing API is to decouple the tests from the application. This decoupling encompasses more than just detaching the tests from the UI: The goal is to decouple the structure of the tests from

the structure of the application.

22.3.1 Structural Coupling

Structural coupling is one of the strongest, and most insidious, forms of test coupling. Imagine a test suite that has a test class for every production class, and a set of test methods for every production method. Such a test suite is deeply coupled to the structure of the application.

When one of those production methods or classes changes, a large number of tests must change as well. Consequently, the tests are fragile, and they make the production code rigid.

The role of the testing API is to hide the structure of the application from the tests. This allows the production code to be refactored and evolved in ways that don't affect the tests. It also allows the tests to be refactored and evolved in ways that don't affect the production code.

This separation of evolution is necessary because as time passes, the tests tend to become increasingly more concrete and specific. In contrast, the production code tends to become increasingly more abstract and general. Strong structural coupling prevents— or at least impedes—this necessary evolution, and prevents the production code from being as general, and flexible, as it could be.

22.3.2 Security

The superpowers of the testing API could be dangerous if they were deployed in production systems. If this is a concern, then the testing API, and the dangerous parts of its implementation, should be kept in a separate, independently deployable component

22.4 Conclusion

Tests are not outside the system; rather, they are parts of the system that must be well designed if they are to provide the desired benefits of stability and regression. Tests that are not designed as part of the system tend to be fragile and difficult to maintain

23 The Web is a Detail

23.1 The Upshot

The GUI is a detail. The web is a GUI. So the web is a detail. As an architect, you want to put details like that behind boundaries that keep them separate from your core business logic.

However, the argument can be made that a GUI, like the web, is so unique and rich that it is absurd to pursue a device-independent architecture.

Another boundary between the UI and the application can be abstracted. The business logic can be be though of as a suite of use cases, each of which performs some function on behalf of a user. Each use case can be described based on the input data, the processing performed, and the output data.

At some point in the dance between the UI and the application, the input data can be said to be complete, allowing the use case to be executed. Upon completion, the resultant data can be fed back into the dance between the UI and the application.

The complete input data and the resultant output data can be placed into data structures and used at the input values and output values for a process that executed the use case. With this approach, we can consider each use case operating the IO device of the UI in a device-independent manner.

24 Frameworks are Details

24.1 The Risks

- The architecture of the framework is often not very clean. Frameworks tend to violate the Dependency Rule.
- The framework may help you with some early features of your application. However, as your project matures, it may outgrow the facilities of the framework.
- The framework may evolve in a direction that you don't find helpful.
- A new and better framework may come along that you wish you could switch to.

24.2 The Solution

Don't let frameworks into your core code. Instead, integrate them into components that plug in to your core code, following the Dependency Rule.

25 The Missing Chapter

25.1 Package by Layer

The design approach "package by layer" is the traditional horizontal layered architecture, where we separate our code based based on what it does from a technical perspective.

In this layered architecture, we have one layer for the web code, one layer for our business logic, and one layer for persistence. In other words, code is sliced horizontally into layers, which are used as a way to group similar types of things. In a strict layered architecture, layers should depend only on the next adjacent lower layer.

25.2 Package by Feature

The design approach "package by feature" is a vertical slicing, based on related features, domain concepts, or aggregate roots.

With this approach, we have the same interfaces and classes as the "package by layer" approach, but they are all placed into a single package rather than being split among separate packages. This is a very simple refactoring from the "package by layer" style, but the top-level organization of the code now screams something about the business domain.

25.3 Ports and Adapters

Approaches such as "ports and adapters", the "hexagonal architecture", "boundaries, controllers, entities", and so on aim to create architectures where business/domain-focused code is independent and separate from the technical implementation details such as frameworks and databases. To summarize, we often see such code bases being composed on an "inside" (domain) and an "outside" (infrastructure).

The "inside" region contains all of the domain concepts, whereas the "outside" region contains the interactions with the outside world (e.g., UIs, databases, third-party integrations). The major rule here is that the "outside" depends on the "inside" - never the other way around.

25.4 Package by Component

In a strict layered architecture, the dependency arrows should always point downward, with layers depending only on the next adjacent lower layer. This comes back to creating a nice, clean, acyclic dependency graph, which is achieved by introducing some rules about how elements on a code base should depend on each other. The big problem here is that we can cheat by introducing some undesirable dependencies, yet still create a nice, acyclic dependency graph.

The "package by component" option, is a hybrid approach with the goal of bundling all of the responsibilities related to a single coarse-grained component into a single package. In the same way that ports and adapters treat the web as just another delivery mechanism, "package by component" keeps the user interface separate from coarse-grained components.

25.5 Other Decoupling Modes

Another option is to decouple dependencies at the source code level, by splitting code across different source code trees. If we take the ports and adapters example, we could have three source code trees:

- The course code for the business and domain (i.e. everything that is independent of technology and framework choices).
- The source code for the web.
- The source code for the data persistence.

The latter two source code trees have a compile-time dependency on the business and domain code, which itself doesn't know anything about the web or the data persistence code.

A simpler approach that others follow for their ports and adapters code is to have just two source code trees:

- Domain code (the "inside")
- Infrastructure code (the "outside")

There is a compile-time dependency from the infrastructure to the domain.