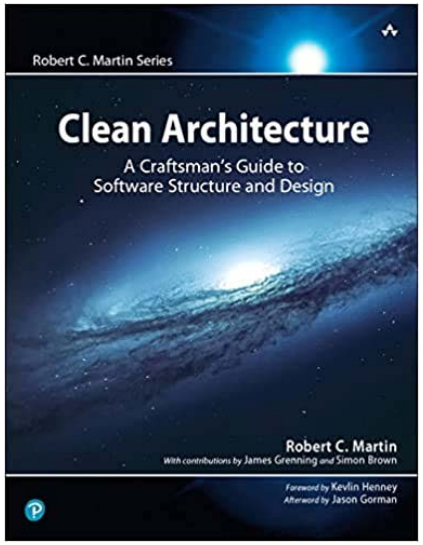
Clean Architecture

“Clean Architecture: A Craftsman's Guide to Software Structure and Design,” Robert C. Martin, Pearson, 2017



Shin-Jie Lee (李信杰)
Associate Professor
Computer and Network Center
Department of CSIE
National Cheng Kung University

Outline

- What is design and architecture?
- Design principles
- Component principles
- The Clean Architecture

WHAT IS DESIGN AND ARCHITECTURE?

- The word “architecture” is often used in the context of something at a high level that is divorced from the lower-level details, whereas “design” more often seems to imply structures and decisions at a lower level.
  - But this usage is nonsensical when you look at what a real architect does.

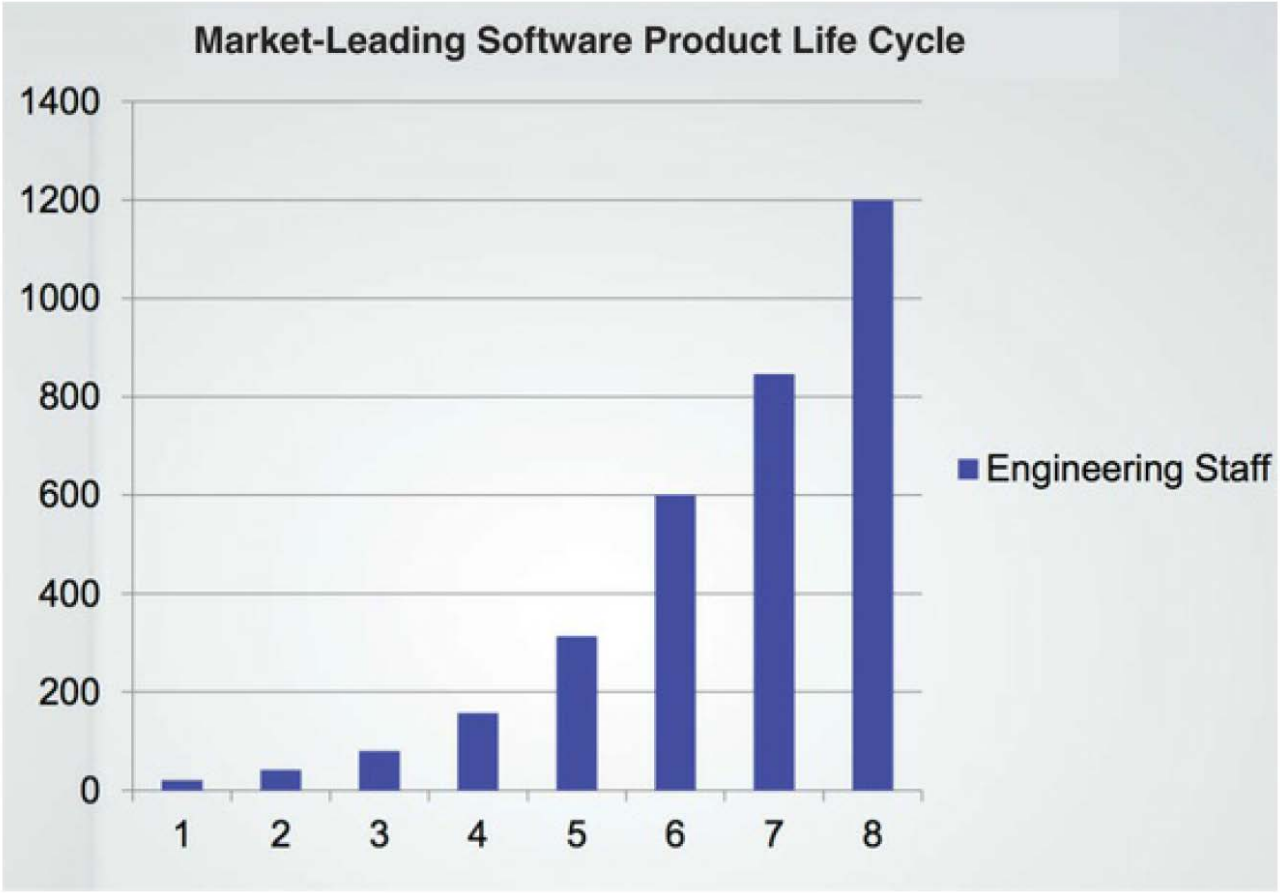
- The low-level details and the high-level structure are all part of the same whole. They form a continuous fabric that defines the shape of the system.
  - You can’t have one without the other; indeed, **no clear dividing line separates them. There is simply a continuum of decisions from the highest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

The goal of software architecture

- ***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 is good.
  - If that effort grows with each new release, the design is bad.

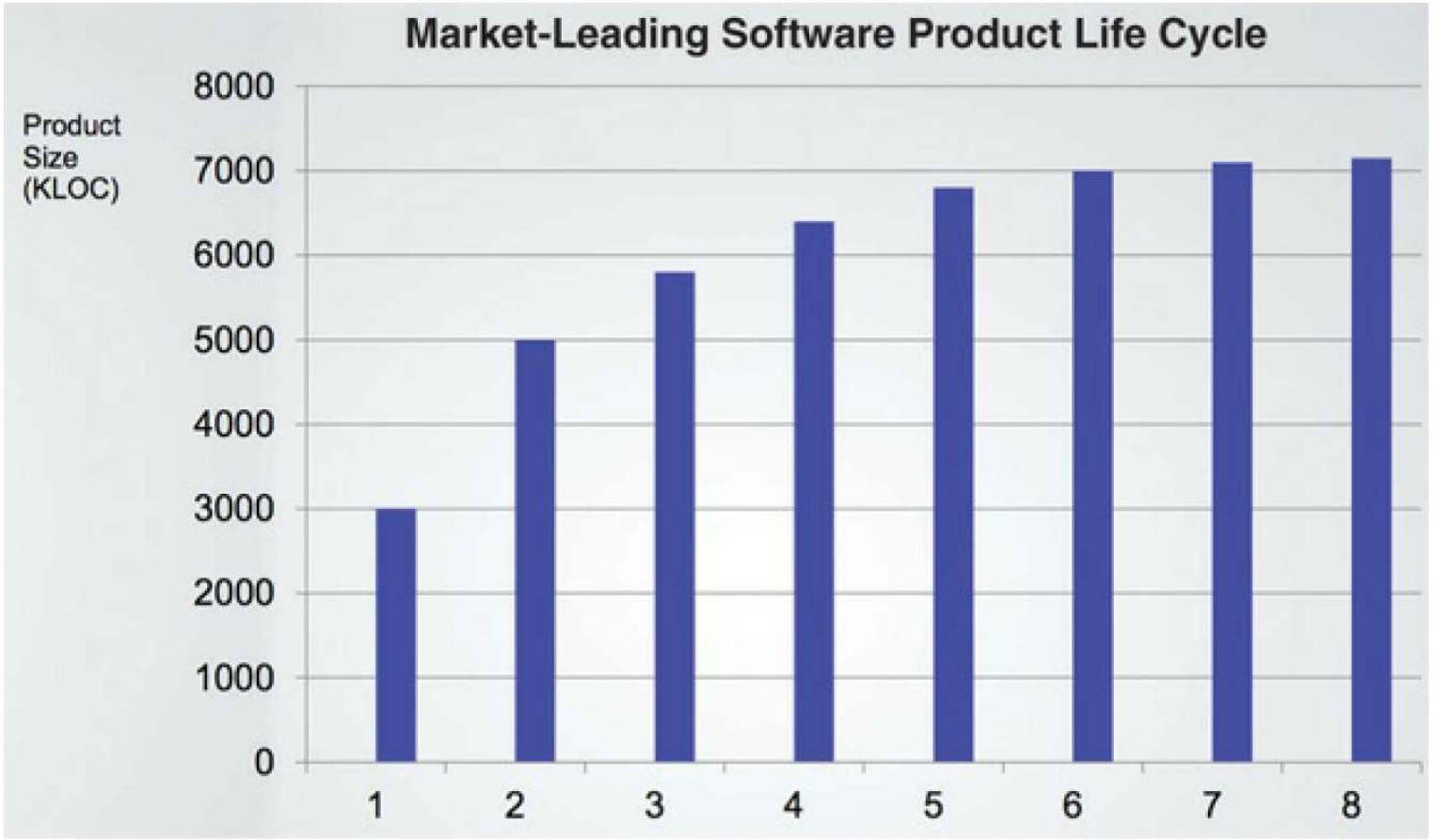
CASE STUDY$_1$

- Growth like that shown must be an indication of significant success!



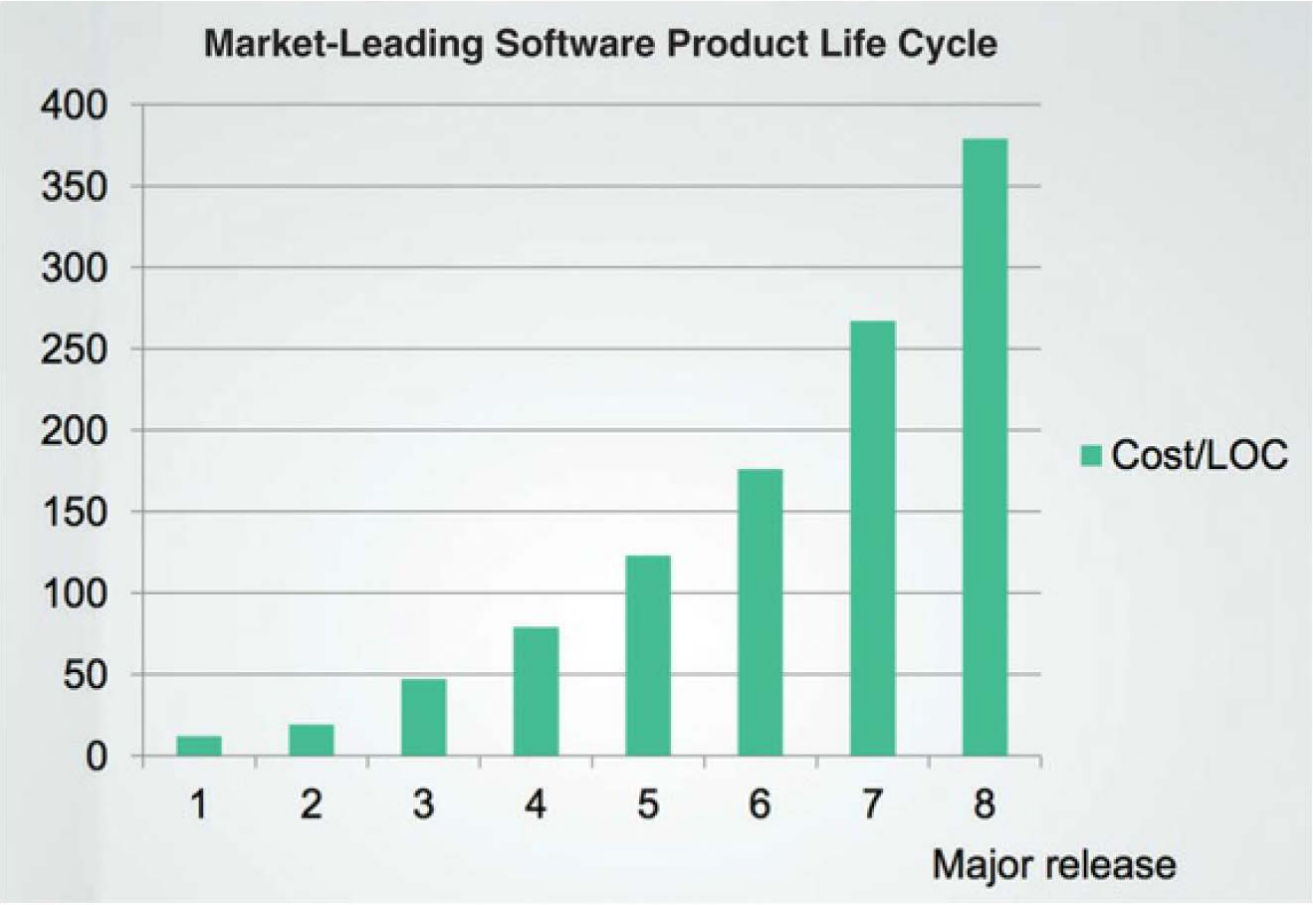
CASE STUDY 2

- Clearly something is going wrong here. Even though every release is supported by an ever-increasing number of developers, the growth of the code looks like it is approaching an asymptote.



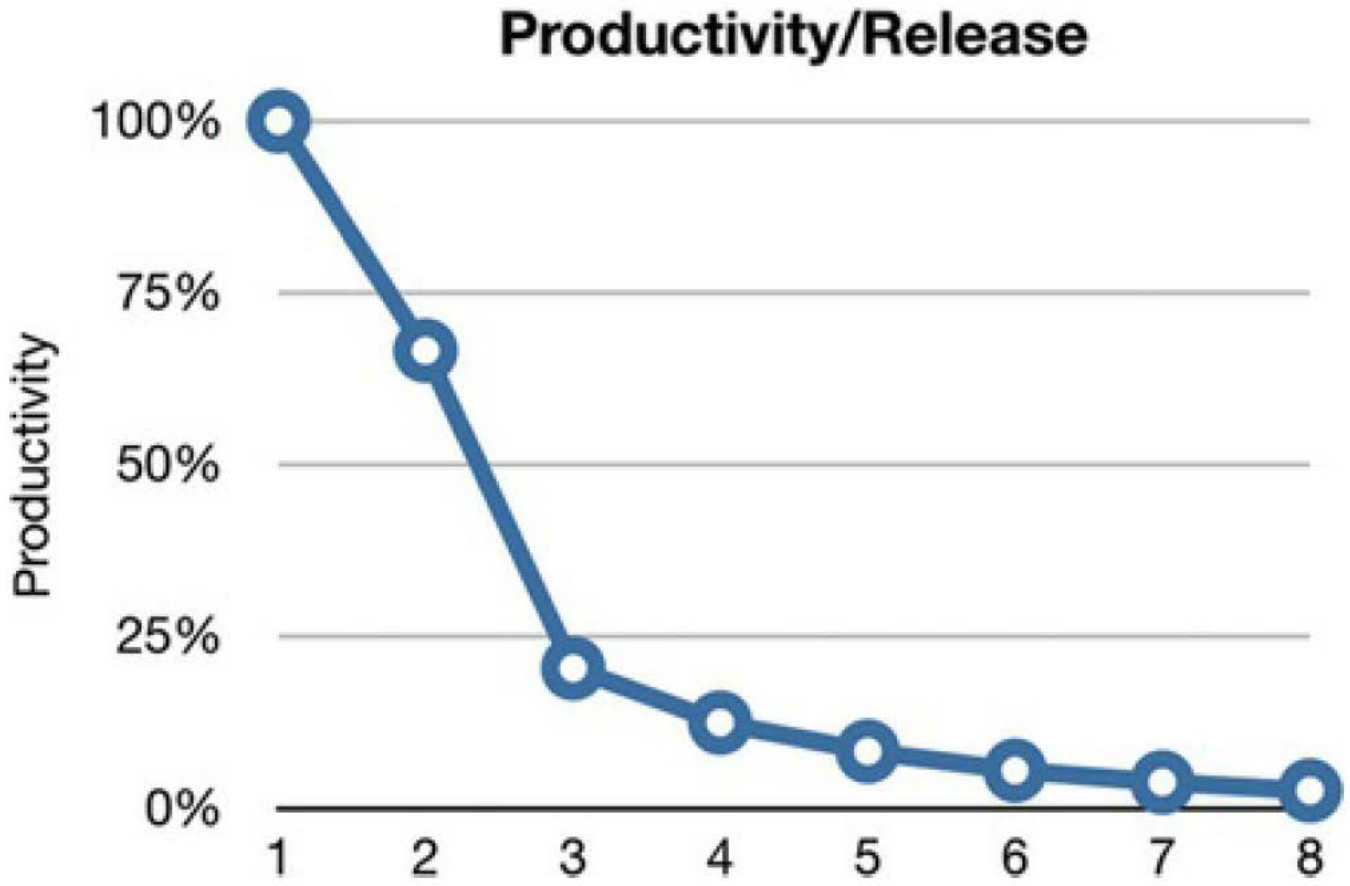
CASE STUDY$_3$

- The code 40 times more expensive to produce in release 8 as opposed to release 1



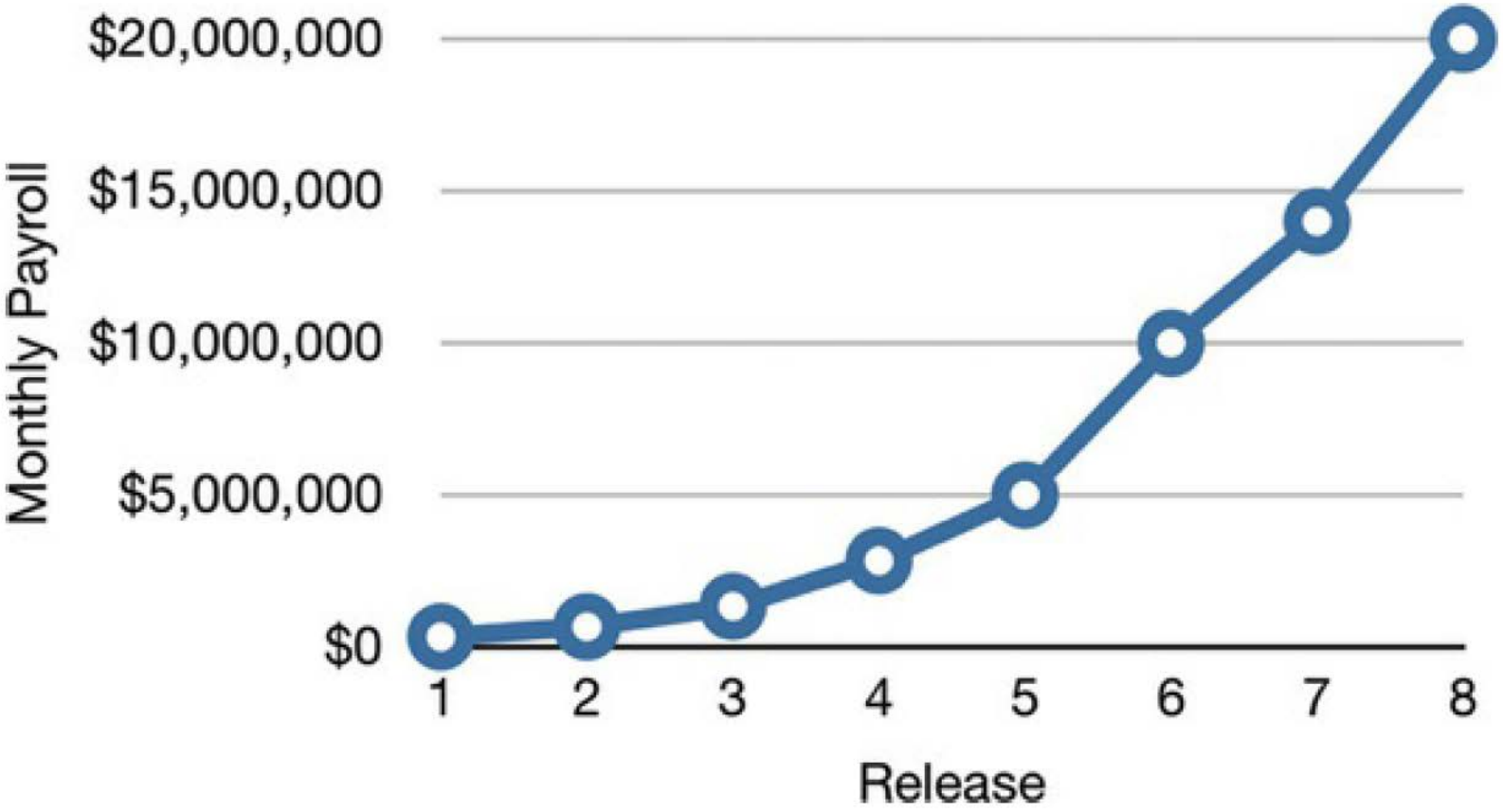
CASE STUDY 4

- From the developers' point of view, this is tremendously frustrating, because everyone is working hard. Nobody has decreased their effort.



CASE STUDY$_5$

- That initial few hundred thousand dollars per month bought a lot of functionality—but the final $20 million bought almost nothing!



WHAT WENT WRONG?

- Developers who accept this lie exhibit the hare's overconfidence in their ability to switch modes from making messes to cleaning up messes sometime in the future, but they also make a simple error of fact.
  - The fact is that making messes is always slower than **staying clean**, no matter which time scale you are using.

- *"The only way to go fast, is to go well."*

A TALE OF TWO VALUES

- Every software system provides two different values to the stakeholders: behavior and structure.
- Unfortunately, software developers often focus on the lesser of the two values, leaving the software system eventually valueless.

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.
- Many programmers believe that is the entirety of their job. They believe their job is to make the machine implement the requirements and to fix any bugs.
- They are sadly mistaken.

ARCHITECTURE

- "*software*"—a compound word composed of "soft" and "ware."
  - "*ware*" means "*product*";
  - "*soft*" means "*intended to be a way to easily change the behavior of machines*"
- 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.
  - Each new request is harder to fit than the last, because the shape of the system does not match the shape of the request.
  - Software developers often feel as if they are forced to jam square pegs into round holes.
- The more this architecture prefers one shape over another, the more likely new features will be harder and harder to fit into that structure.

THE GREATER VALUE$_1$ - Function or architecture?

- If you ask the business managers, they'll often say that it's more important for the software system to work. Developers, in turn, often go along with this attitude.
- But it's the wrong attitude
  - If you give me a program that works perfectly but is impossible to change, then it won't work when the requirements change, and I won't be able to make it work. Therefore the program will become useless.
  - If you give me a program that does not work but is easy to change, then I can make it work, and keep it working as requirements change. Therefore the program will remain continually useful.

THE GREATER VALUE$_2$ - Function or architecture?

- If you ask the business managers if they want to be able to make changes, they'll say that of course they do,
  - but may then qualify their answer by noting that the current functionality is more important than any later flexibility.
- In contrast, if the business managers ask you for a change, and your estimated costs for that change are unaffordably high,
  - the business managers will likely be furious that you allowed the system to get to the point where the change was impractical.

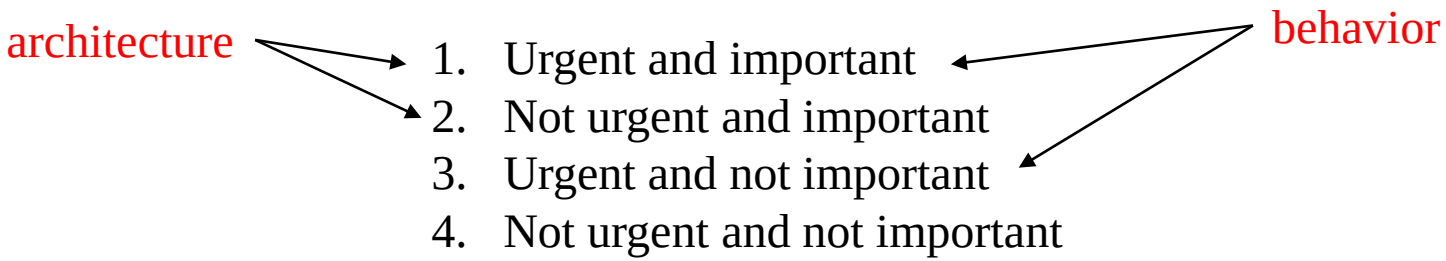
EISENHOWER'S MATRIX[1]

- Those things that are urgent are rarely of great importance, and those things that are important are seldom of great urgency.

IMPORTANT URGENT	IMPORTANT NOT URGENT
UNIMPORTANT URGENT	UNIMPORTANT NOT URGENT

EISENHOWER'S MATRIX$_2$

- We can arrange these four couplets into priorities:



- The mistake that business managers and developers often make is to elevate items in position 3 to position 1.
- The dilemma for software developers is that business managers are not equipped to evaluate the importance of architecture. That's what software developers were hired to do.
- Therefore it is the responsibility of the software development team to assert the importance of architecture over the urgency of features.

FIGHT FOR THE ARCHITECTURE

- Remember, as a software developer, you are a stakeholder. You have a stake in the software that you need to safeguard.
- That's part of your role, and part of your duty. And it's a big part of why you were hired.

- 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.
- If that is allowed to happen, it means the software development team did not fight hard enough for what they knew was necessary.

DESIGN PRINCIPLES

How to arrange the bricks into walls and rooms

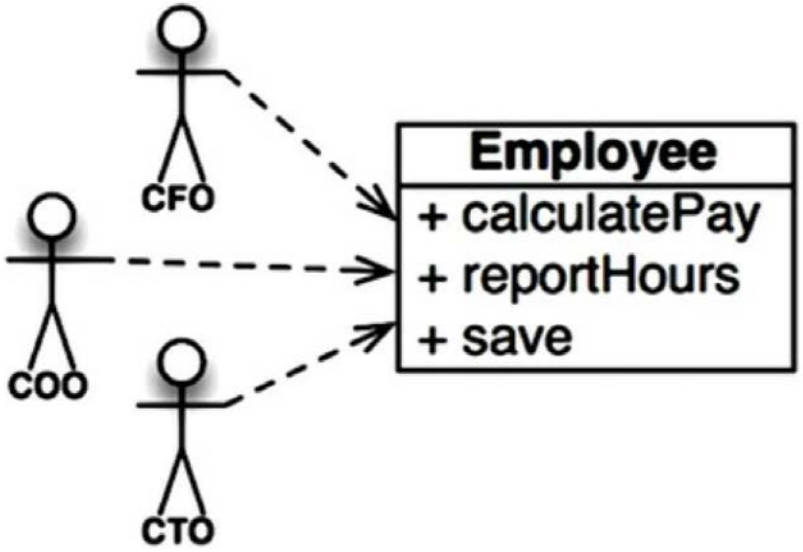
DESIGN PRINCIPLES - SOLID

- The goal of the principles is the creation of mid-level software structures that:
  - Tolerate change
  - Are easy to understand, and
  - Are the basis of components that can be used in many software systems.

- **SRP:** The Single Responsibility Principle
- **OCP:** The Open-Closed Principle
- **LSP:** The Liskov Substitution Principle
- **ISP:** The Interface Segregation Principle
- **DIP:** The Dependency Inversion Principle

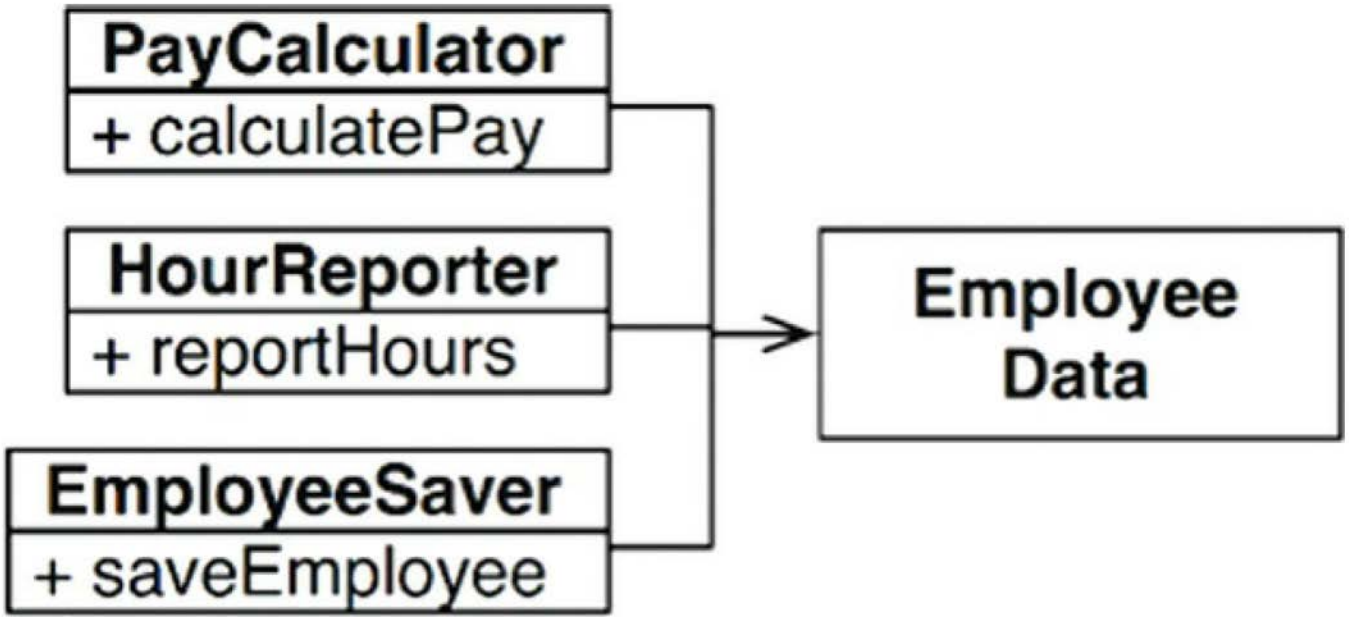
SRP: THE SINGLE RESPONSIBILITY PRINCIPLE$_1$

- Historically, the SRP has been described this way:
  - *A module should have one, and only one, reason to change.*
- We can rephrase the principle to say this:
  - *A module should be responsible to one, and only one, actor*.
- This class violates the SRP because those three methods are responsible to three very different actors.



SRP: THE SINGLE RESPONSIBILITY PRINCIPLE$_2$

- Perhaps the most obvious way to solve the problem is to separate the data from the functions.
- Each class holds only the source code necessary for its particular function.



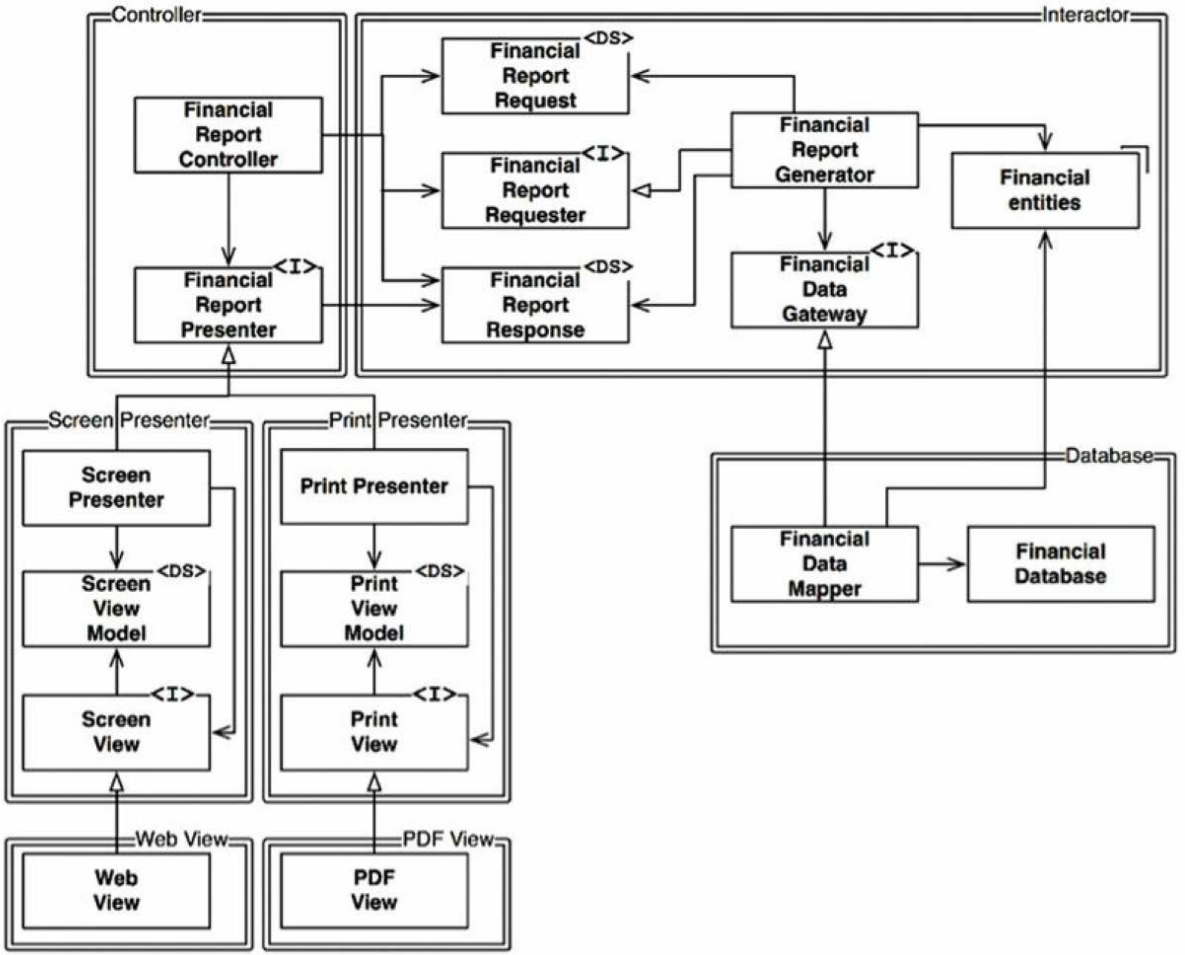
OCP: THE OPEN-CLOSED PRINCIPLE$_1$

- ***A software artifact should be open for extension but closed for modification***
- Imagine, for a moment, that we have a system that displays a financial summary on a web page. The data on the page is scrollable, and negative numbers are rendered in red.
- Now imagine that the stakeholders ask that this same information be turned into a report to be printed on a black-and-white printer. The report should be properly paginated, with appropriate page headers, page footers, and column labels. Negative numbers should be surrounded by parentheses.

Figure 8.1 Applying the SRP

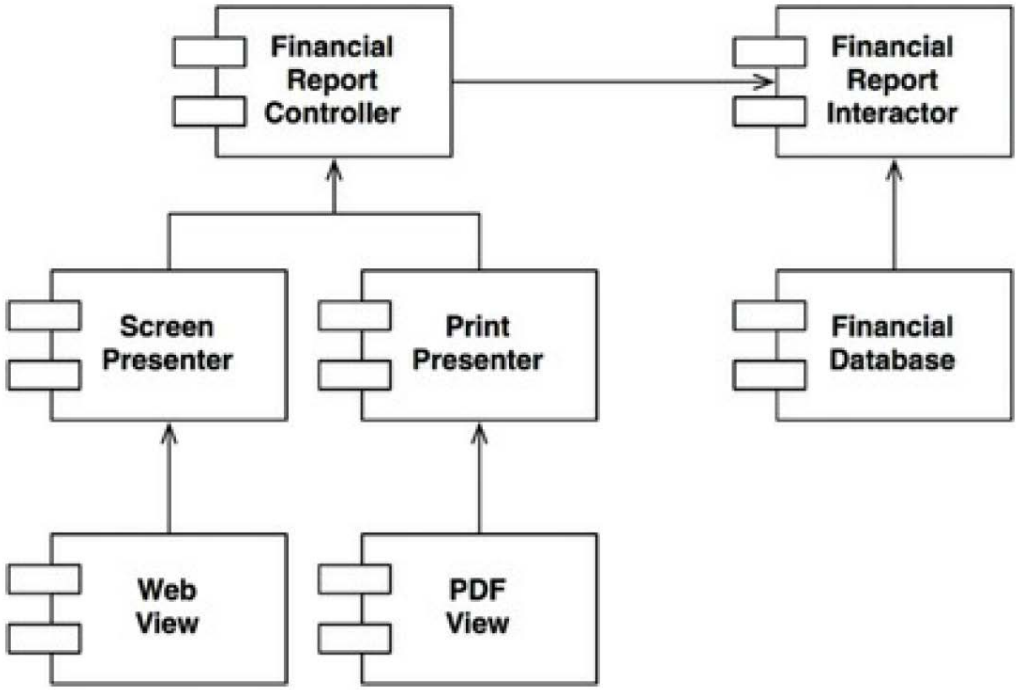
OCP: THE OPEN-CLOSED PRINCIPLE$_2$

- Partitioning the processes into classes, and separating those classes into components



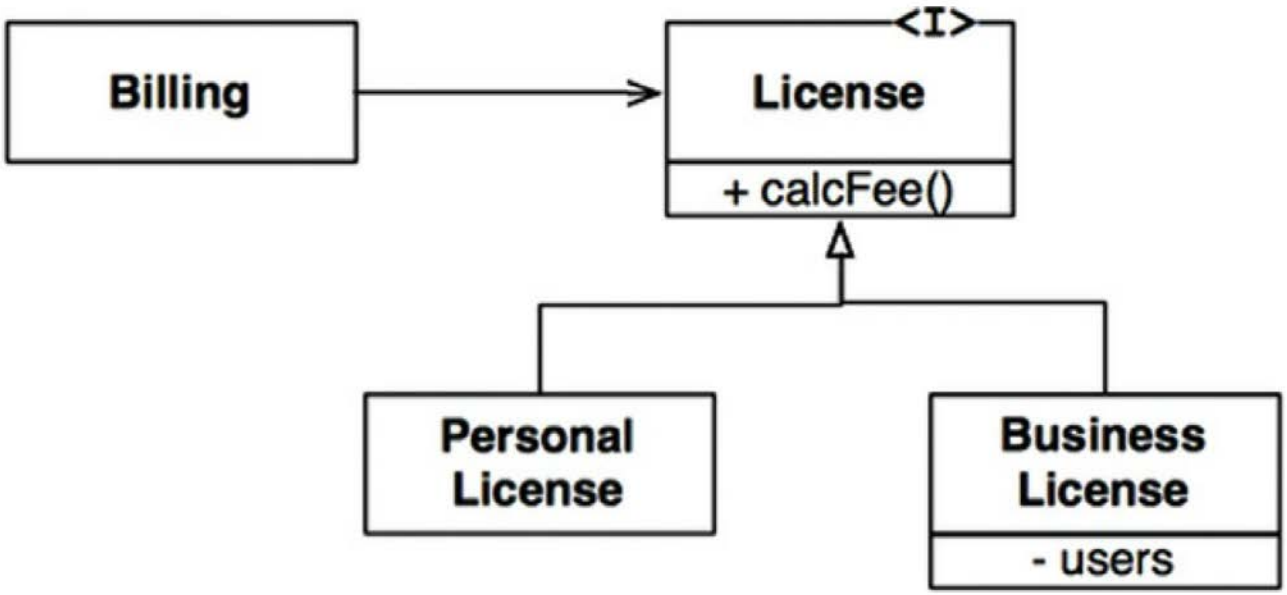
OCP: THE OPEN-CLOSED PRINCIPLE$_3$

- All component relationships are unidirectional.
  - *If component A should be protected from changes in component B, then component B should depend on component A.*
- This is how the OCP works at the architectural level.
  - Higher-level components in that hierarchy are protected from the changes made to lower-level components.



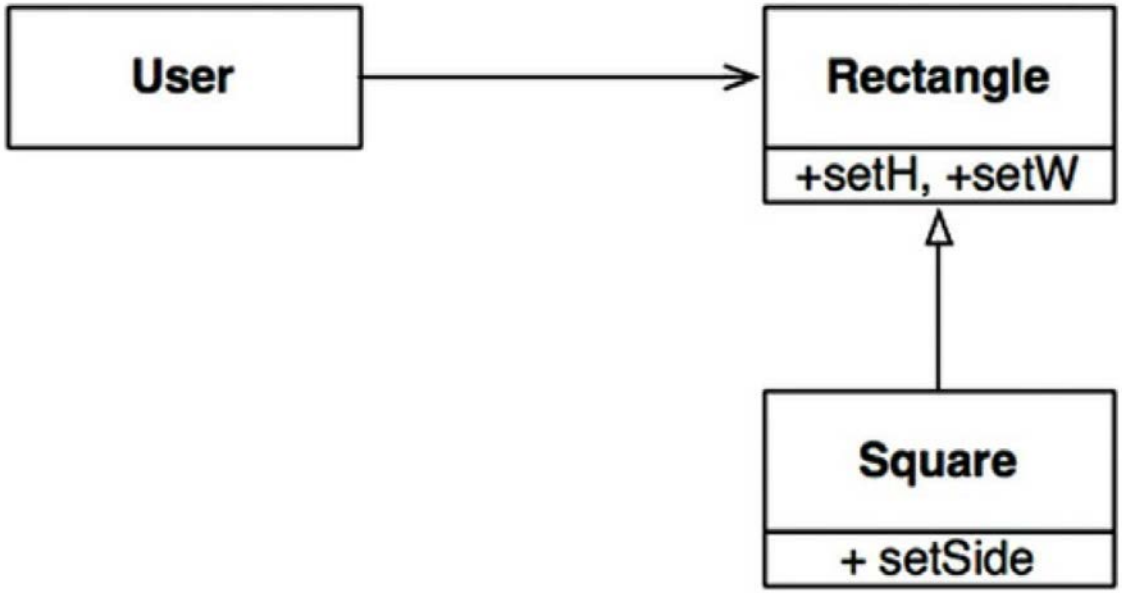
LSP: THE LISKOV SUBSTITUTION PRINCIPLE$_1$

- In 1988, Barbara Liskov wrote the following as a way of defining subtypes.
  - *If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T,*
  - *the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T.*



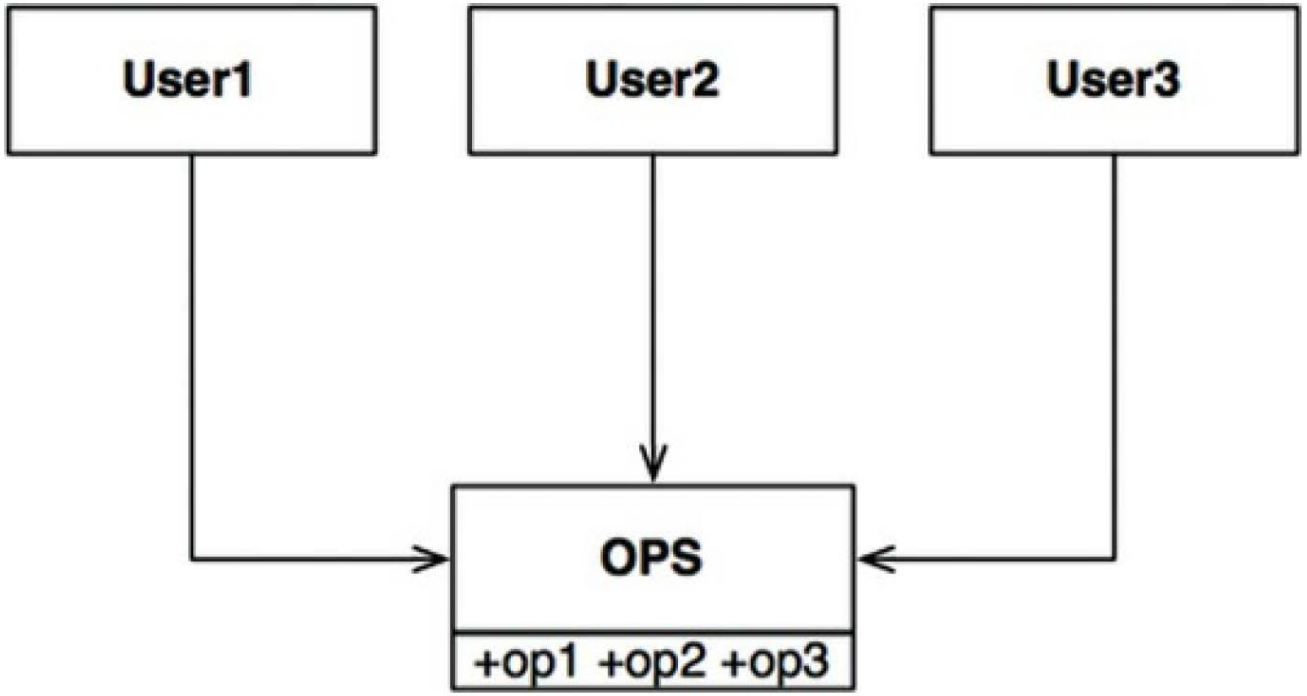
LSP: THE LISKOV SUBSTITUTION PRINCIPLE$_2$

- The canonical example of a violation of the LSP is the square/rectangle problem
  - The only way to defend against this kind of LSP violation is to add mechanisms to the `User` (such as an `if` statement) that detects whether the `Rectangle` is, in fact, a `Square`



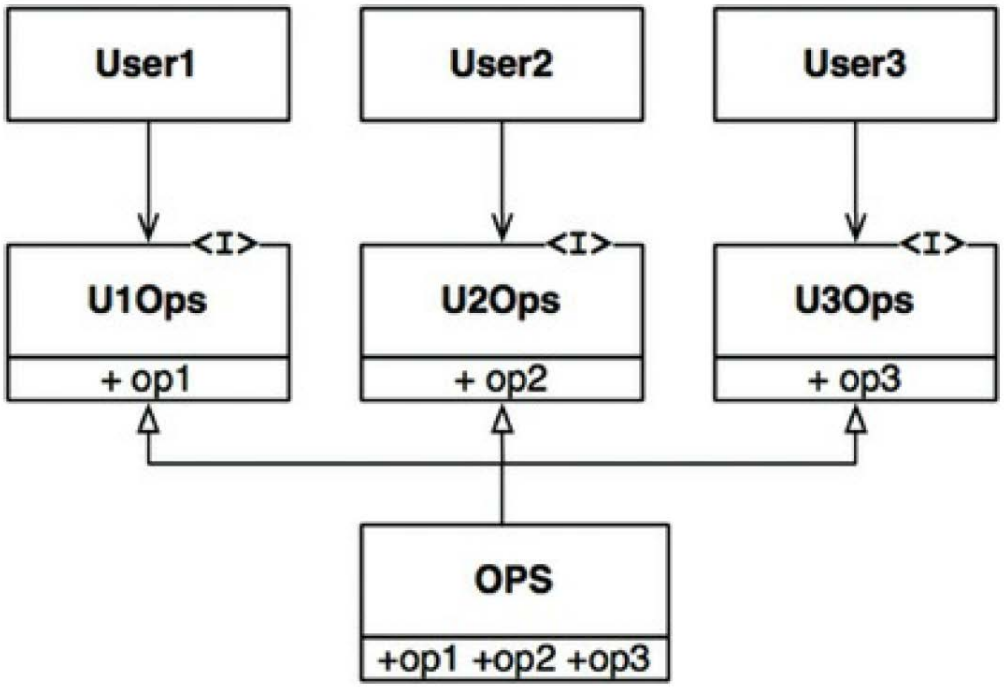
ISP: THE INTERFACE SEGREGATION PRINCIPLE$_1$

- Let's assume that `User1` uses only `op1`, `User2` uses only `op2`, and `User3` uses only `op3`.
  - A change to the source code of `op2` in `OPS` will force `User1` to be recompiled and redeployed, even though nothing that it cared about has actually changed.



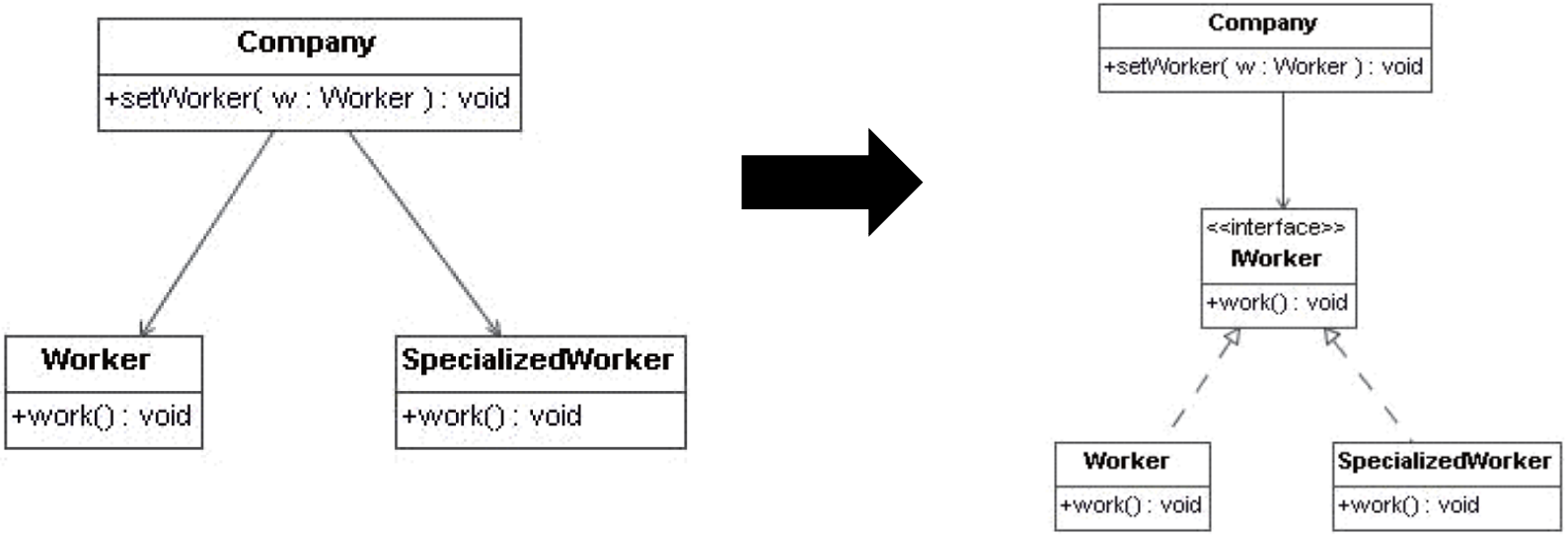
ISP: THE INTERFACE SEGREGATION PRINCIPLE$_2$

- This problem can be resolved by segregating the operations into interfaces
- Thus a change to `OPS` that `User1` does not care about will not cause `User1` to be recompiled and redeployed.



DIP: THE DEPENDENCY INVERSION PRINCIPLE$_1$

- The most flexible systems are those in which source code dependencies refer only to abstractions, not to concretions.



DIP: THE DEPENDENCY INVERSION PRINCIPLE$_2$

- Clearly, treating this idea as a rule is unrealistic. For example, the `String` class in Java is concrete.
- By comparison, the `String` class is very stable. Changes to that class are very rare and tightly controlled.
- 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.

DIP: THE DEPENDENCY INVERSION PRINCIPLE$_3$ - STABLE ABSTRACTIONS

- Good software designers and architects work hard to reduce the volatility of interfaces. They try to find ways to add functionality to implementations without making changes to the interfaces.

- **Don’t refer to volatile concrete classes.**
- **Don’t derive from volatile concrete classes**
- **Don’t override concrete functions**
- **Never mention the name of anything concrete and volatile**

COMPONENT PRINCIPLES

How to arrange the rooms into buildings

COMPONENTS

- Components are the units of deployment.
- In Java, they are jar files. In Ruby, they are gem files. In .Net, they are DLLs. In compiled languages, they are aggregations of binary files.
- Components can be linked together into a single executable. Or they can be aggregated together into a single archive, such as a `.war` file. Or they can be independently deployed as separate dynamically loaded plugins, such as `.jar` or `.dll` or `.exe` files.
- Well-designed components always retain the ability to be independently deployable and, therefore, independently developable.

COMPONENT COHESION

THE REUSE/RELEASE EQUIVALENCE PRINCIPLE (REP)

- People who want to reuse software components cannot, and will not, do so unless those components are tracked through a release process and are given release numbers.
  - This is not simply because, without release numbers, there would be no way to ensure that all the reused components are compatible with each other
  - Rather, it also reflects the fact that software developers need to know when new releases are coming, and which changes those new releases will bring.
- Classes and modules that are grouped together into a component should be releasable together.
- The fact that they share the same version number and the same release tracking, and are included under the same release documentation, should make sense both to the author and to the users.

THE COMMON CLOSURE PRINCIPLE (CCP)$_1$

- ***Gather into components those classes that change for the same reasons and at the same times.***
- ***Separate into different components those classes that change at different times and for different reasons.***

- This is the Single Responsibility Principle restated for components.

THE COMMON CLOSURE PRINCIPLE (CCP) 2

- The Open Closed Principle (OCP) states that classes should be closed for modification but open for extension.
- Because 100% closure is not attainable, closure must be strategic.
- The CCP amplifies this lesson 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.

THE COMMON REUSE PRINCIPLE (CRP)

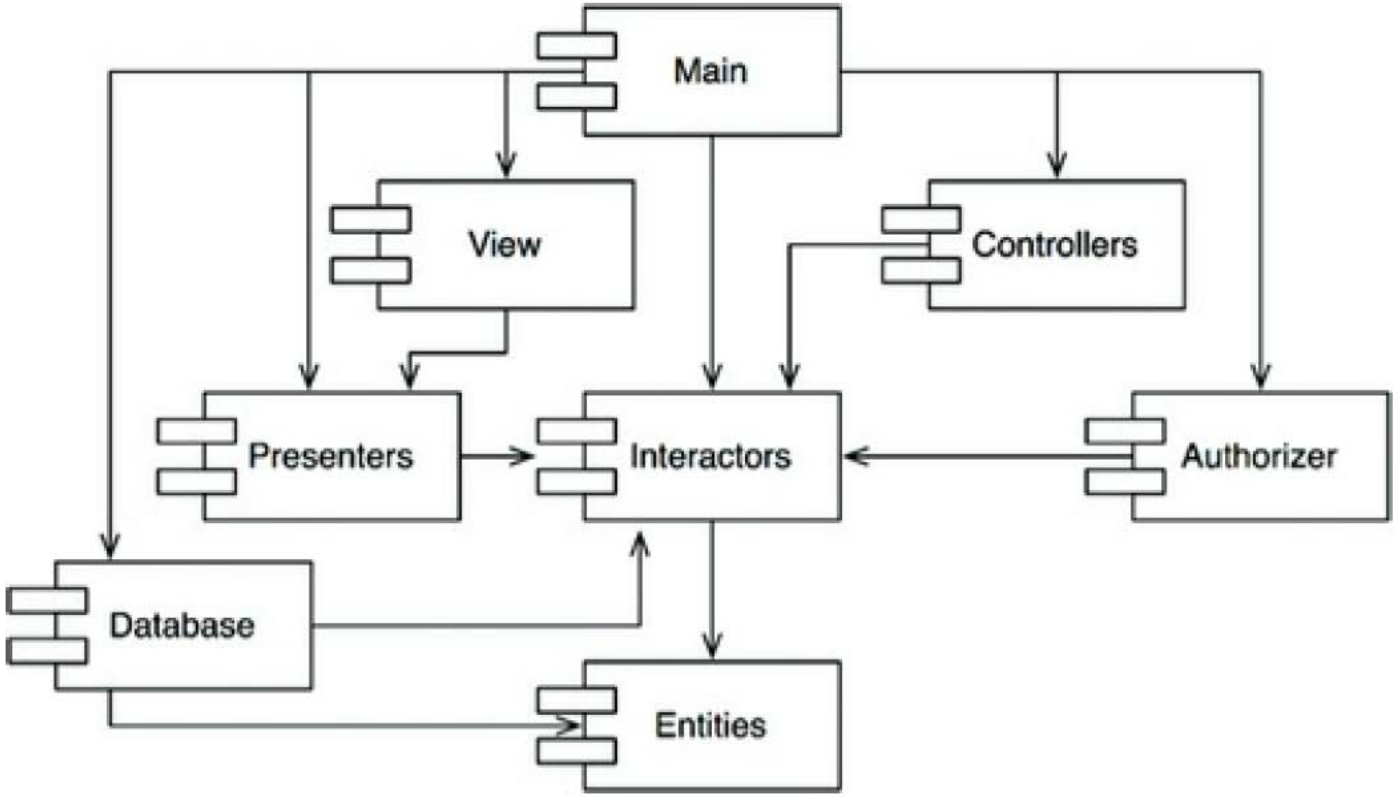
- Classes and modules that tend to be reused together belong in the same component.
- In a component we would expect to see classes that have lots of dependencies on each other.
- Classes that are not tightly bound to each other should not be in the same component.

- ***Don’t depend on things you don’t need***

COMPONENT COUPLING

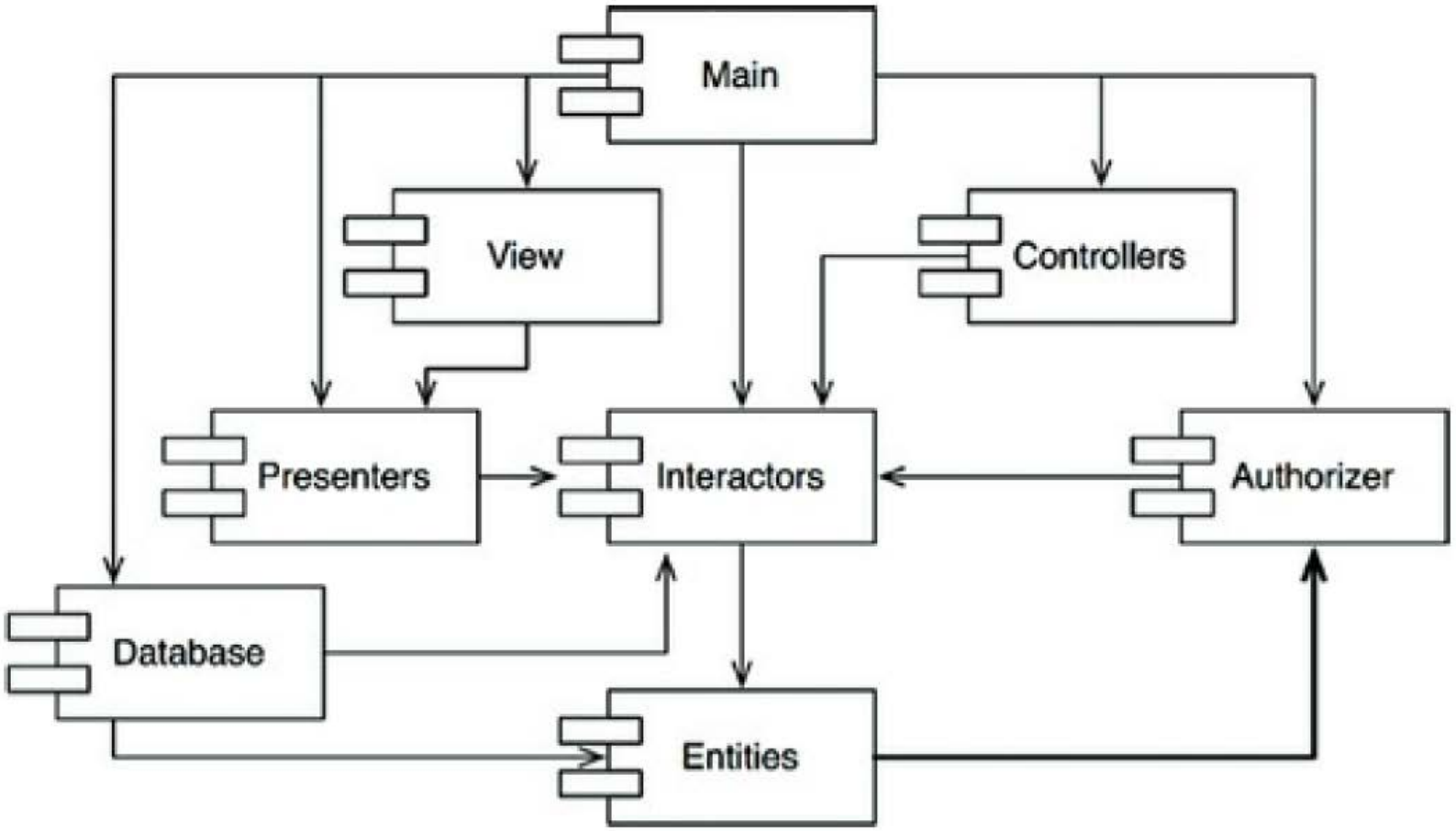
THE ACYCLIC DEPENDENCIES PRINCIPLE[1]

- ***Allow no cycles in the component dependency graph***
- When the developers working on the `Presenters` component would like to run a test of that component, they just need to build with the versions of the `Interactors` and `Entities` components.



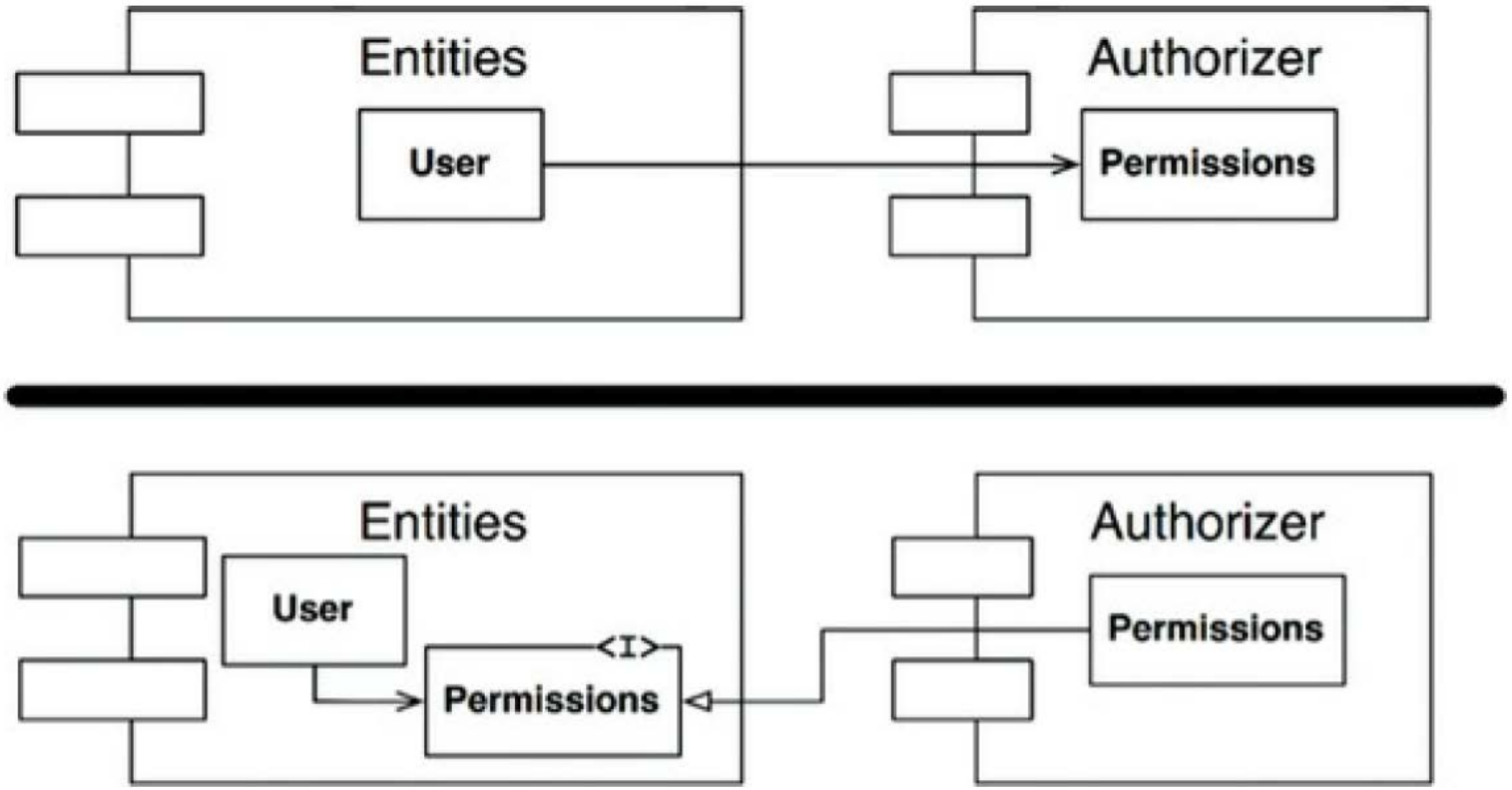
THE ACYCLIC DEPENDENCIES PRINCIPLE$_2$

- A cycle in the component dependency graph



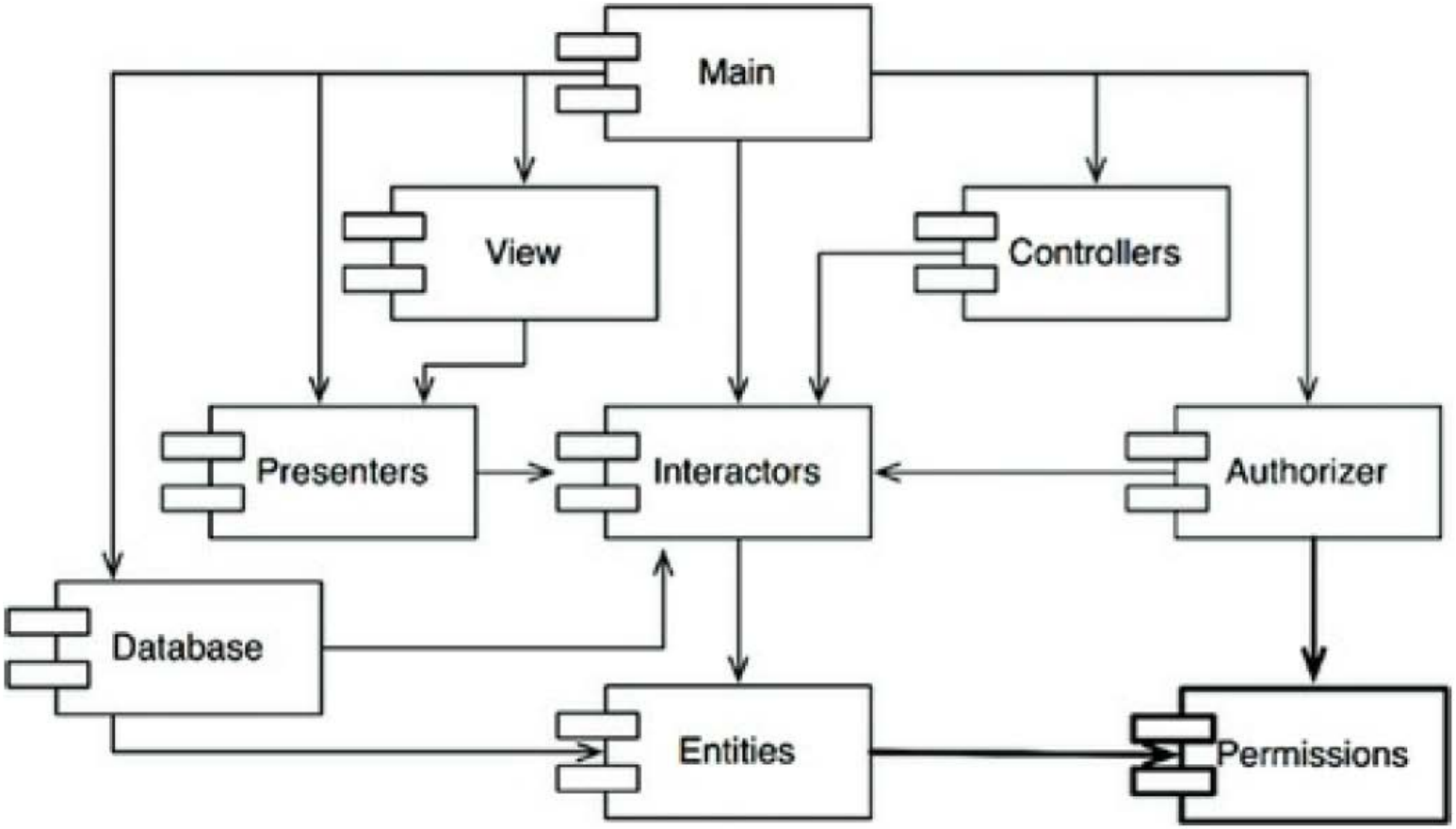
THE ACYCLIC DEPENDENCIES PRINCIPLE$_3$

- Inverting the dependency between `Entities` and `Authorizer`



THE ACYCLIC DEPENDENCIES PRINCIPLE$_4$

- The new component that both `Entities` and `Authorizer` depend on



TOP-DOWN DESIGN

- ***The component structure cannot be designed from the top down***

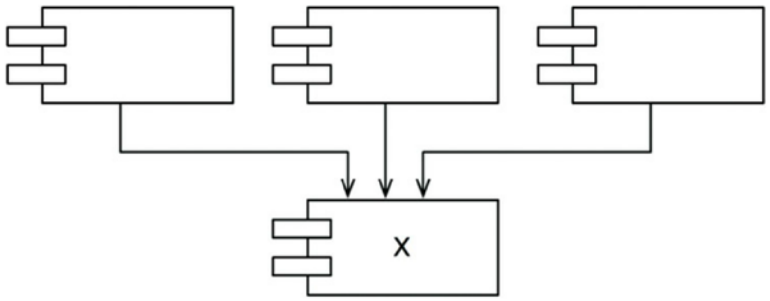
- In fact, omponent dependency diagrams have very little do to with describing the function of the application (high-level functional decompositions). Instead, they are a map to the buildability and maintainability of the application.

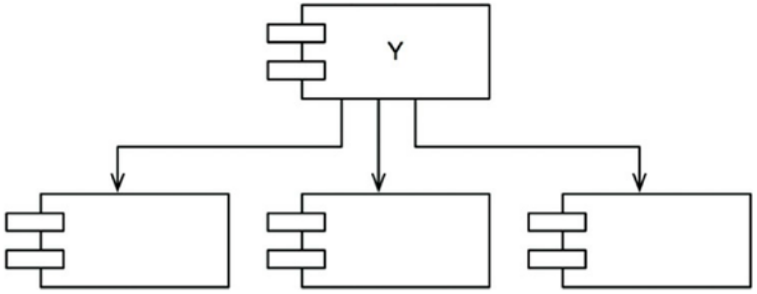
- If we tried to design the component dependency structure before we designed any classes, we would not know much about common closure, be unaware of any reusable elements, and almost certainly create components that produced dependency cycles.

THE STABLE DEPENDENCIES PRINCIPLE $(SDP)_1$ - STABILITY

- A component with lots of incoming dependencies is very **stable** because it requires a great deal of work to reconcile any changes with all the dependent components.
  - A table is very stable because it takes a considerable amount of effort to turn it over.

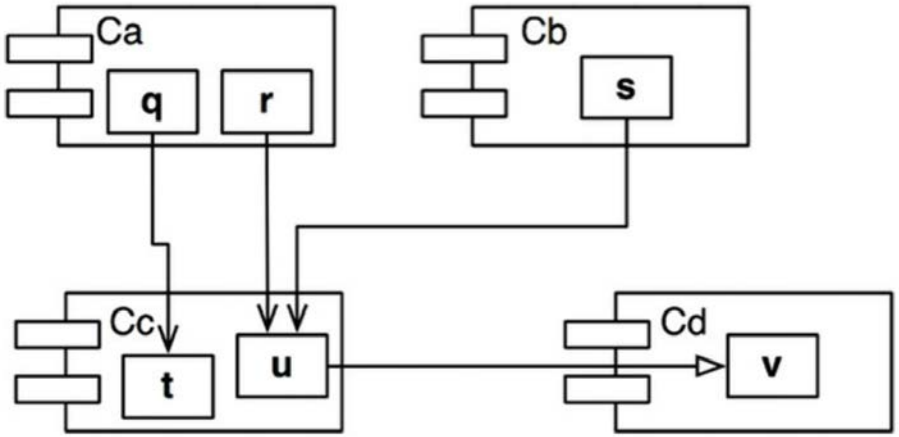
- X is a stable component, and Y is a very unstable component





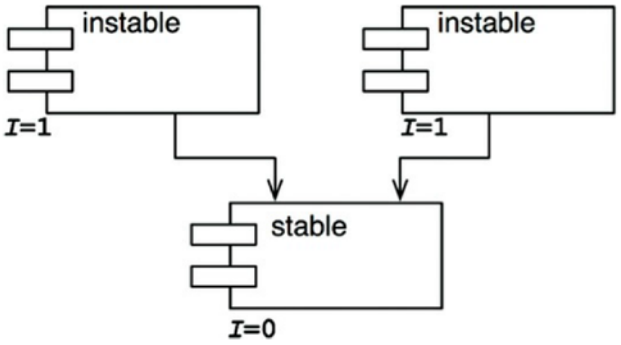
THE STABLE DEPENDENCIES PRINCIPLE $(SDP)_2$ – STABILITY METRICS

- ***Fan-in***: Incoming dependencies. The number of classes outside this component that depend on classes within the component.
- ***Fan-out***: Outgoing dependencies. The number of classes inside this component that depend on classes outside the component.
- ***Instability***: I = Fan-out / (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 SDP says that the I metric of a component should be larger than the I metrics of the components that it depends on.**
- For component `Cc`, Fan-in = 3, Fan-out = 1 and I = 1/4.

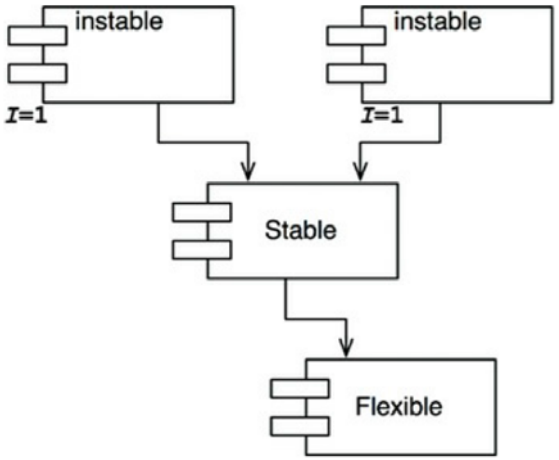


THE STABLE DEPENDENCIES PRINCIPLE $(SDP)_3$ – NOT ALL COMPONENTS SHOULD BE STABLE

- If all the components in a system were maximally stable, the system would be unchangeable
- Example: An ideal configuration for a system with three components.

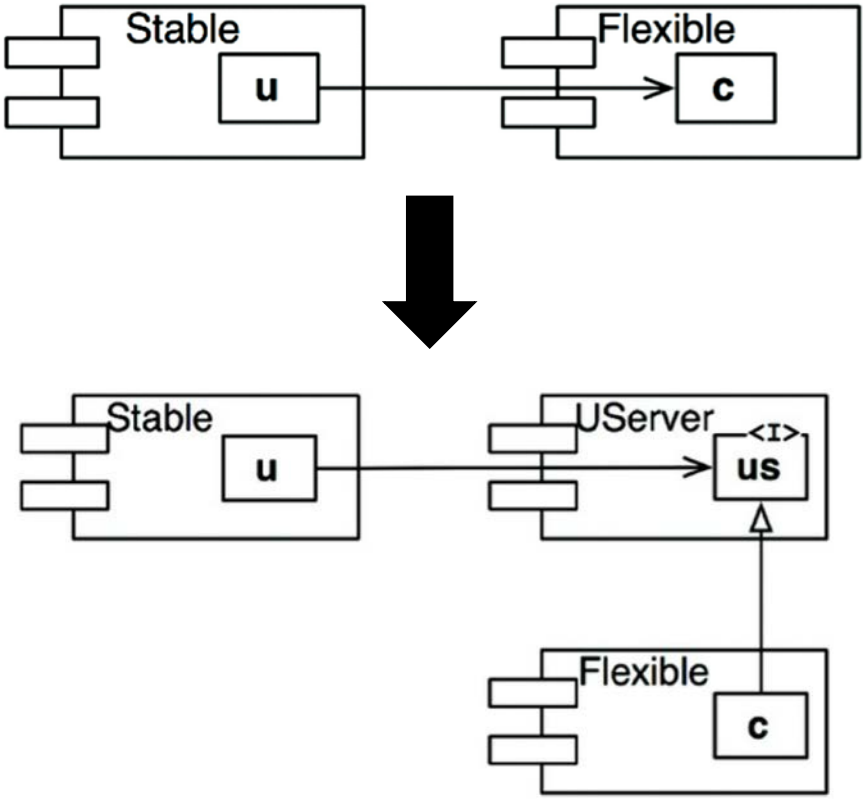


- Example: SDP violation



THE STABLE DEPENDENCIES PRINCIPLE $(SDP)_4$ – NOT ALL COMPONENTS SHOULD BE STABLE

- Using abstract components to fix the SDP violation
  - `us` contains nothing but an interface. These abstract components are very stable and, therefore, are ideal targets for less stable components to depend on.



THE STABLE ABSTRACTIONS PRINCIPLE $(SAP)_1$

- ***A component should be as abstract as it is stable***

- A stable component should also be abstract so that its stability does not prevent it from being extended
- An unstable component should be concrete since it its instability allows the concrete code within it to be easily changed

THE STABLE ABSTRACTIONS PRINCIPLE $(SAP)_2$ - MEASURING ABSTRACTION

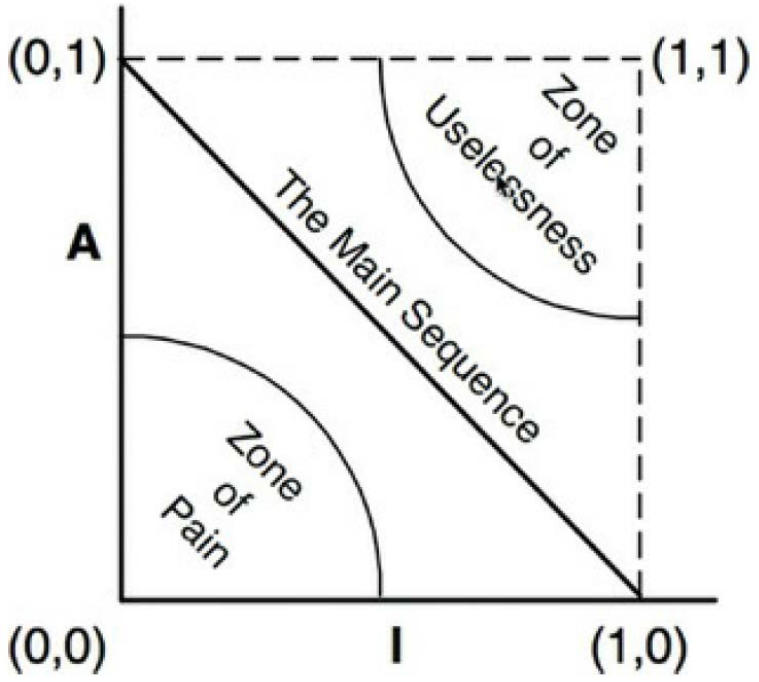
- Nc: The number of classes in the component.
- Na: The number of abstract classes and interfaces in the component.
- A: Abstractness. A = Na ÷ Nc

- The A metric ranges from 0 to 1.
  - 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

THE STABLE ABSTRACTIONS PRINCIPLE $(SAP)_3$ – MAIN SEQUENCE

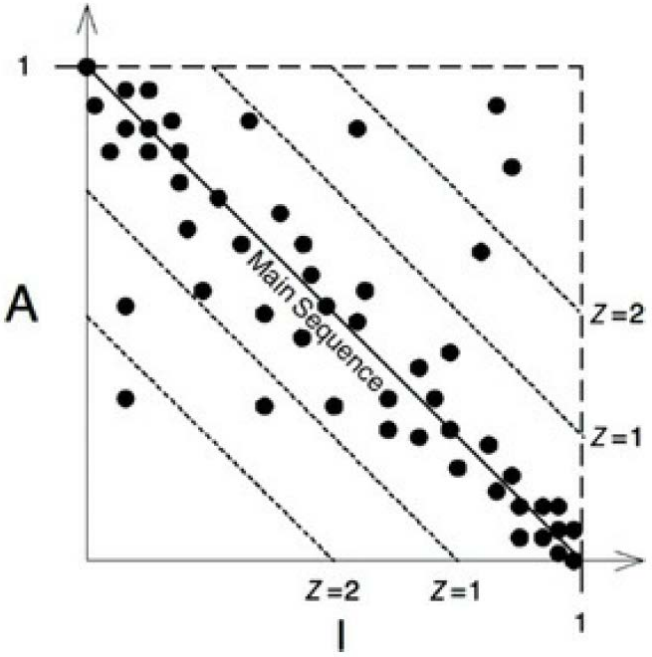
- **The Zone of Pain**
  - Database schemas are notoriously volatile, extremely concrete, and highly depended on. This is one reason why schema updates are generally painful
- **The Zone of Uselessness**
  - The software entities that inhabit this region are often leftover abstract classes that no one ever implemented.

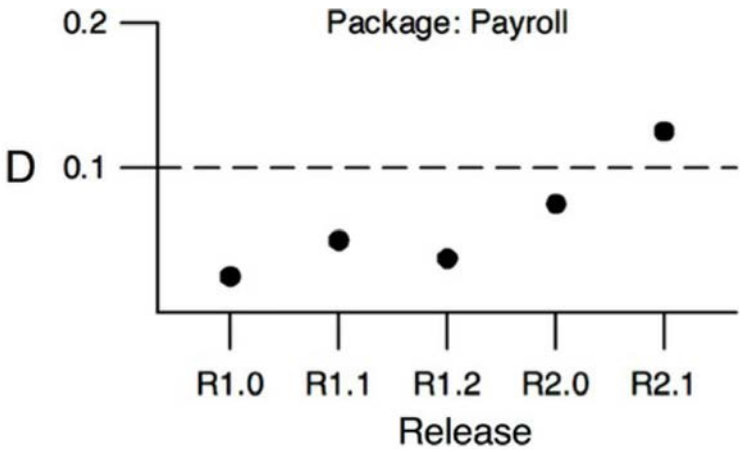
- Good architects strive to position the majority of their components at those endpoints.
  - However, in my experience, some small fraction of the components in a large system are neither perfectly abstract nor perfectly stable.



THE STABLE ABSTRACTIONS PRINCIPLE $(SAP)_4$ – DISTANCE FROM THE MAIN SEQUENCE

- 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 as possible from the Main Sequence.
- Some of them are more than one standard deviation (Z = 1) away from the mean. These aberrant components are worth examining more closely





ARCHITECTURE

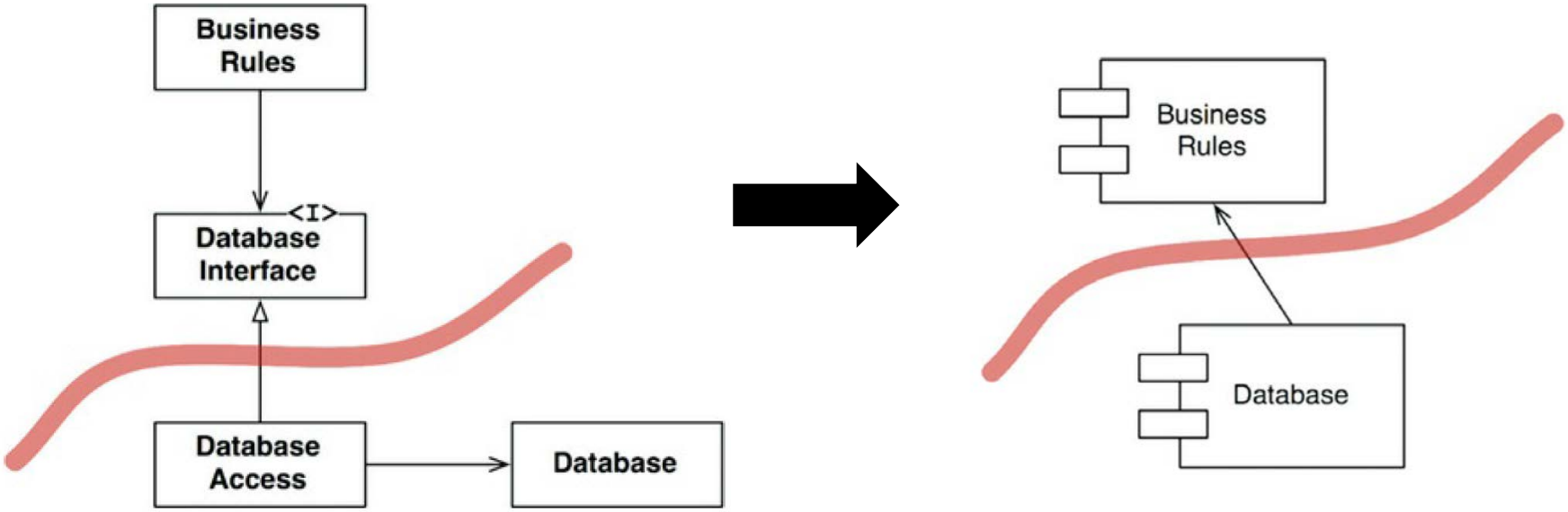
BOUNDARIES: DRAWING LINES

- ***Software architecture is the art of drawing lines that I call boundaries***
  - Those boundaries separate software elements from one another, and restrict those on one side from knowing about those on the other.

- The goal of an architect is to minimize the human resources required to build and maintain the required system. What it is that saps this kind of people-power?
  - ***Coupling—and especially coupling to premature decisions***
    - Decisions about frameworks, databases, web servers, utility libraries, dependency injection, and the like.

WHICH LINES DO YOU DRAW, AND WHEN DO YOU DRAW THEM?

- You draw lines between things that matter and things that don't
  - The database doesn't matter to the business rules, so there should be a line between them.



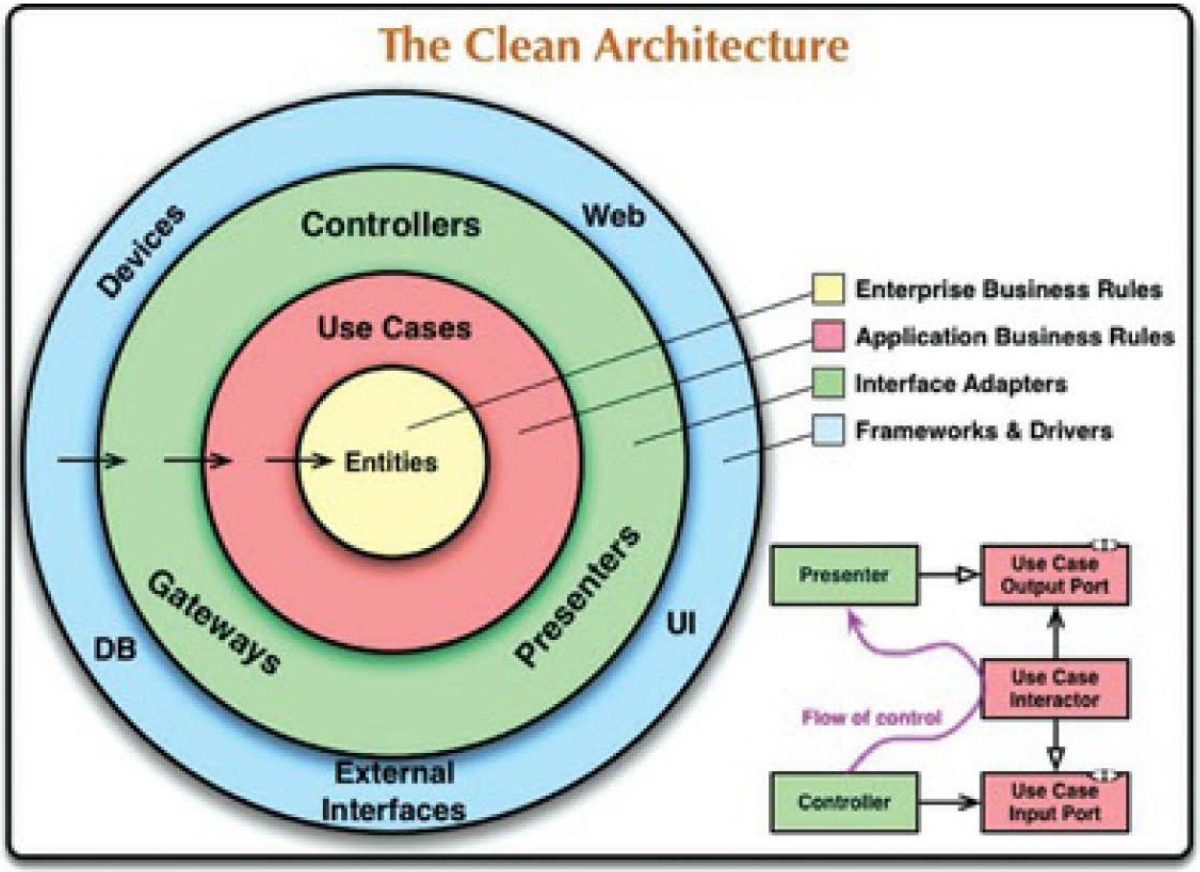
THE CLEAN ARCHITECTURE

- **Independent of frameworks**. The architecture does not depend on the existence of some library of feature-laden software. This allows you to use such frameworks as tools, rather than forcing you to cram your system into their limited constraints.
- **Testable**. The business rules can be tested without the UI, database, web server, or any other external element.
- **Independent of the UI**. The UI can change easily, without changing the rest of the system. A web UI could be replaced with a console UI, for example, without changing the business rules.
- **Independent of the database**. You can swap out Oracle or SQL Server for Mongo, BigTable, CouchDB, or something else. Your business rules are not bound to the database.
- **Independent of any external agency**. In fact, your business rules don’t know anything at all about the interfaces to the outside world.

THE CLEAN ARCHITECTURE

- **Dependency Rule**: *Source code dependencies must point only inward, toward higher-level policies*
- There's no rule that says you must always have just these four. However, the Dependency Rule always applies

The outer circles are mechanisms. The inner circles are policies



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.
- 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.
  - For example, you would not expect these objects to be affected by a change to page navigation or security.

USE CASES

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

INTERFACE ADAPTERS

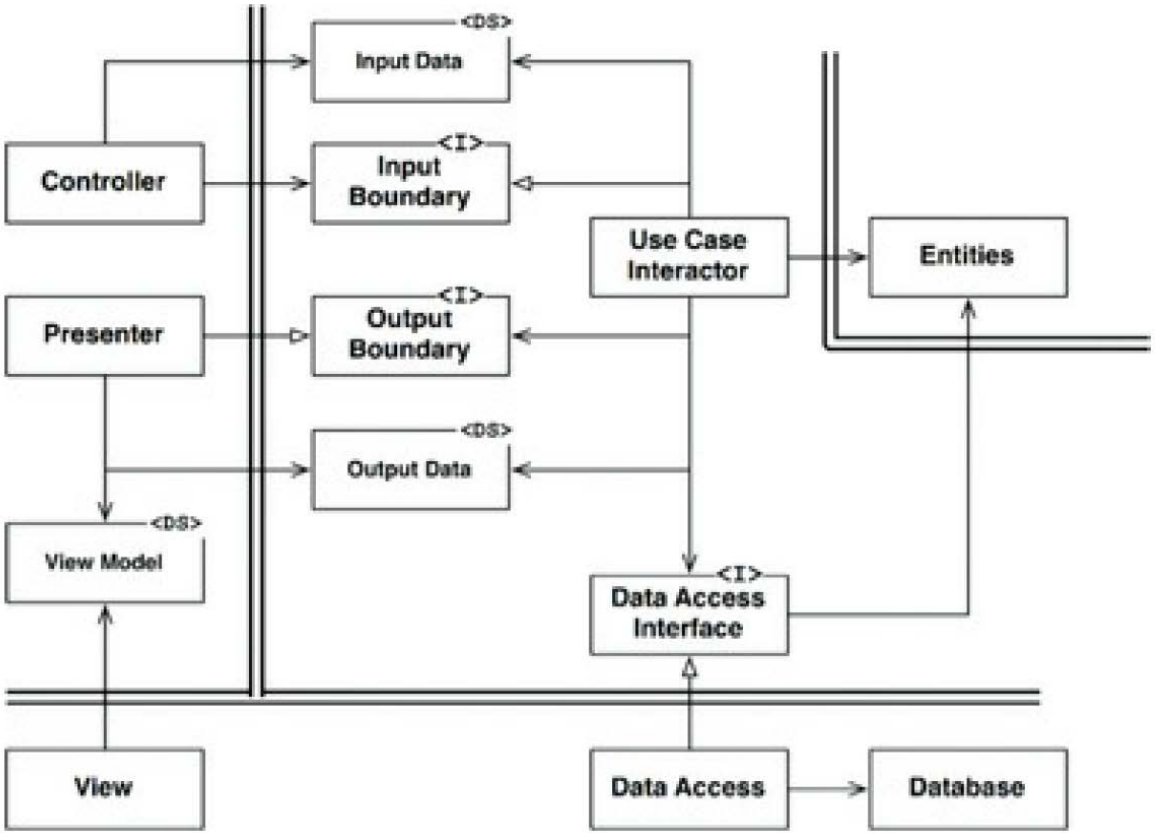
- The software in the interface adapters layer is a set of adapters that convert data from the format most convenient for the use cases and entities, to the format most convenient for some external agency such as the database or the web.
- It is this layer, for example, that will wholly contain the MVC 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.

FRAMEWORKS AND DRIVERS

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

A TYPICAL SCENARIO

- All dependencies cross the boundary lines pointing inward, following the Dependency Rule.



COMPONENT ARCHITECTURE