20

Component Principles

If the SOLID principles tell us how to arrange the bricks into walls and rooms, then the component principles tell us how to arrange the rooms into buildings. Large software systems, like large buildings, are built out of smaller components.

In this chapter, we will discuss what software components are, what elements they should be composed of, and how they should be composed together into systems.

This chapter is an overview. The principles are described in detail in [PPP02] and again in [Clean Arch].

Components

Components are the units of deployment. They are the smallest entities that can be deployed as part of a system. 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. In interpreted languages, they are aggregations of source files. In all languages, they are the granule of deployment.

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 plug-ins, such as *.jar* or *.dll* or *.exe* files. However, regardless of how they are eventually deployed, well-designed components always retain the ability to be independently deployable and therefore *independently developable*.

A Brief History of Components

In the early years of software development, programmers controlled the memory location and layout of their programs. One of the first lines of code in a program would have been the origin statement, which declared the address at which the program was to be loaded. Consider this simple PDP-8 program. It consists of a subroutine named `GETSTR`, which inputs a string from the keyboard and saves it in a buffer. It also has a little unit test program to exercise `GETSTR`.

```
        *200
        TLS
START,  CLA
        TAD BUFR
        JMS GETSTR
        CLA
        TAD BUFR
        JMS PUTSTR
        JMP START

BUFR,   3000

GETSTR, 0
        DCA PTR
NXTCH,  KSF
        JMP .-1
        KRB
        DCA I PTR
        TAD I PTR
        AND K177
```

```
            ISZ PTR
            TAD MCR
            SZA
            JMP NXTCH

K177,       177
MCR,        -15
```

Note the `*200` at the start of this program. This tells the compiler to generate code that will be loaded at address 2008.[1]

---

1. The PDP-8 had 4096 12-bit words. We used octal representation in those days.

---

2. My first employer kept several dozen decks of the subroutine library source code on a shelf. When you wrote a new program, you simply grabbed one of those decks and slapped it onto the end of your deck.

---

3. This video, which I uploaded to YouTube in 2022, shows the process: https://www.youtube.com/playlist?list=PLINHgpdx-_Cq0bImYF-IWVO8U2XERZuit.

---

4. Actually, most of those old machines used core memory, which did not get erased when you powered the computer down. So we often just left the function library loaded for days at a time.

This is a foreign concept for most programmers today. They don't usually have to think about where a program is loaded in the memory of the computer. But in the early days, this was one of the first decisions a programmer needed to make. In those days, programs were not *relocatable*.

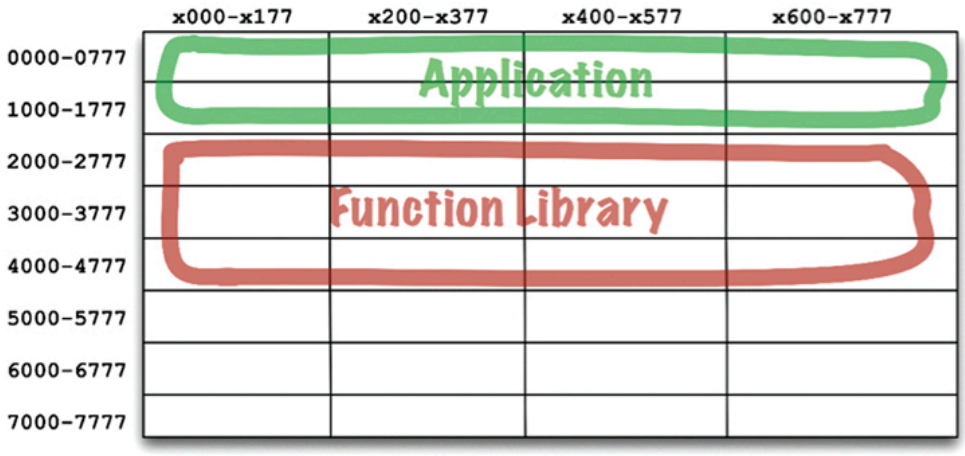
How did you access a library function in those days? The code above shows you how we did it. We included the source code of the library functions with our application code and compiled them all as a single program.[2] Libraries were kept in source, not in binary.

The problem with this was that devices in those days were slow, and memory was expensive and therefore limited. Compilers needed to make

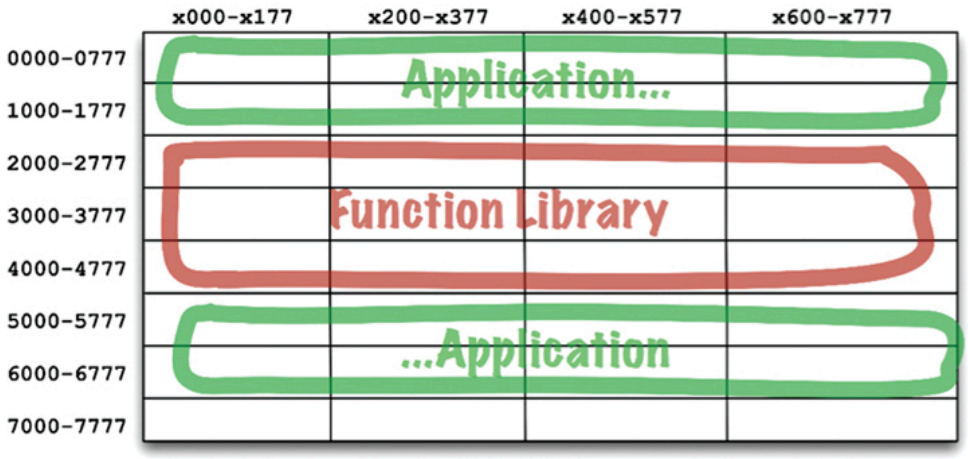
several passes over the source code, but memory was too limited to keep all the source code resident. So the compiler had to read in the source code several times using those slow devices.

This took a long time; and the larger your function library was, the longer the compiler took. Compiling a large program could take hours.[3] In order to shorten the compile times, we separated the source code of the function library from the applications. We compiled the function library separately and loaded the binary at a known address; say, 20008. We created a symbol table for the function library and compiled that with our application code. When we wanted to run an application, we would load the binary function library,[4] and then load the application.

Memory looked like this:



This worked fine so long as the application could fit between 00008 and 17778. But soon applications grew to be larger than the space allotted. So then, programmers had to split their applications into two address segments, jumping around the function library.



It should be clear that this was not a sustainable situation. As we added more functions to the function library, it exceeded its bounds, and we had to allocate more space for it near 70008. And this fragmentation of programs and libraries necessarily continued as computer memory grew.

Clearly, something had to be done.

Relocatability

The solution was relocatable binaries. The idea was very simple. We changed the compiler to output binary code based at address zero. The loader would be told where to load the relocatable code, and the loader would add the load address to all the addresses in the binary code. This was a little more complicated than it sounds, but we found ways to make it work.

Now we could tell the loader where to load the function library and where to load the application. In fact, the loader would accept several binary inputs and simply load them in memory one right after the other, relocating them as it loaded them. This allowed us to load only those functions that we needed.

The compiler was also changed to emit the names of the functions as metadata in the relocatable binary. If a program called a library function, the compiler would emit that name as an external reference. If a program defined a function, the compiler would emit that name as an external definition. Then, the loader could *link* the external references to the external definitions once it had determined where it had loaded those definitions.

And the linking loader was born.

Linkers

The linking loader allowed us to divide our programs onto segments that can be separately compiled and loaded. This worked great when we had relatively small programs being linked with relatively small libraries. However, in the late ’60s and early ’70s, we got more ambitious, and programs got a lot bigger.

Eventually, the linking loaders were too slow to tolerate. This was because our function libraries were stored on slow devices such a magnetic tape. Even the disks back then were quite slow. Using these relatively slow devices, the linking loaders had to read dozens, if not hundreds, of binary libraries in order to resolve the external references. As programs grew larger and larger, and more library functions accumulated in our libraries, a linking loader could take well over an hour just to load the program.

So the loader and the linker were separated into two phases. We took the slow part, the part that did that linking, and we put it into a separate application called the *linker*. The output of the linker was a linked relocatable that a relocating loader could load very quickly. This allowed us to prepare an executable using the slow linker; but then we could load it quickly, anytime we wanted to.

Then came the ’80s. We were working in C or some other high-level language. As our ambitions grew, so did our programs. Programs containing hundreds of thousands of lines of code were not unusual.

We compiled our source modules from *.c* files into *.o* files, and fed them into the linker to create executable files that could be quickly loaded. Compiling each individual module was relatively fast; but compiling all the modules took a bit of time. And then the linker would take the most time. Turnaround had again grown to an hour or more in many cases.

It seemed we were doomed to endlessly chase our tail. Throughout the ’60s, ’70s, and ’80s, all the changes we had made to speed up our workflow were thwarted by our ambitions and the size of the programs we wrote. We could not seem to escape from the hourlong turnaround times. Loading time remained fast; but compile-link times were the bottleneck.

We were, of course, experiencing Murphy’s law of program size:

Programs will grow to fill all available compile and link time.

But Murphy was not the only contender in town. Along came Moore,[5] and in the late ’80s, the two battled it out. Moore won that battle. Disks started to shrink and get exponentially faster. Computer memory started to get so

ridiculously cheap that much of the data on disk could be cached in RAM. Computer clock rates increased from 1MHz to 100MHz.

---

5. Moore's law: Computer speed, memory, and density double every 18 months. This law held from the 1950s to 2000; but then slowed to a crawl, if not a full stop.

By the mid-'90s, the time spent linking had begun to shrink faster than our ambitions could make programs grow. In many cases, link time dropped to a matter of *seconds*. For small jobs, the idea of a linking loader became feasible again.

This was the era of ActiveX, shared libraries, and the beginnings of *.jar* and *.dll* files. Computers and devices had gotten so fast that we could once again do the linking at load time. We could link together several *.jar* files or several shared libraries in a matter of seconds, and execute the resulting program.

And so the component plug-in architecture was born.

Today, we routinely ship *.jar* files or `DLL`s or shared libraries as plug-ins to existing applications. If you want to create a mod to *Minecraft*, for example, you simply include your custom *.jar* files in a certain folder. If you want to plug ReSharper into Visual Studio, you simply include the appropriate `DLL`s.

These dynamically linked files, which can be plugged together at runtime, are the software components of our architectures. It has taken 50 years, but we have arrived at a place where component plug-in architecture can be the casual default as opposed to the herculean effort it once was.

Component Cohesion

Which classes belong in which components? This is an important decision, and it requires guidance from good software engineering principles. Unfortunately, over the years, this decision has been made in an ad hoc manner based almost entirely on context.

In this section, we will discuss the three principles of component cohesion:

- REP: The Reuse/Release Equivalence Principle
- CCP: The Common Closure Principle
- CRP: The Common Reuse Principle

The Reuse/Release Equivalence Principle (REP)

The granule of reuse is the granule of release.

The past decade has seen the rise of a menagerie of module management tools, such as Maven, Leiningen, the `tools.deps` library, and RVM. These tools have grown in importance because, during that time, a vast number of reusable components and component libraries have been created. We are now living in the age of software reuse—a fulfillment of one of the oldest promises of OO.

The REP is a principle that seems obvious, in hindsight. 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. It is also because software developers need to know when new releases are coming and what changes those new releases will bring.

It is not uncommon for developers to be alerted about a new release and decide, based on the changes made in that release, to continue to use the old release instead. Therefore, the release process must produce the appropriate notifications and release documentation so that users can make informed decisions about when and whether to integrate.

From a software design and architecture point of view, this principle means that the classes and modules that are formed into a component must belong to a cohesive group. The component cannot simply consist of a random hodgepodge of classes and modules. There must be some overarching theme or purpose that those modules all share.

Of course, this should be obvious. However, there is another way to look at this that is perhaps not quite so obvious. 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 the users.

Clearly this is weak advice. Saying that something should "make sense" is just a way of waving your hands in the air and trying to sound authoritative. The advice is weak because it is hard to precisely explain the glue that holds the classes and modules together into a single component. Weak though the advice may be, the principle itself is important, because violations are easy to detect—they don't "make sense." If you violate the REP, your users will know, and will not be impressed with your architectural skills.

The weakness of this principle is more than compensated for by the strength of the next two. Indeed, the next two principles strongly constrain this principle.

The Common Closure Principle (CCP)

Gather into components those modules that change for the same reasons and at the same times. Separate into different components those modules that change for different reasons or at different times.

This is the Single Responsibility Principle (SRP) restated for components. Just as the SRP says that a *module* should not contain multiples reasons to change, this principle says that a component should not have multiple reasons to change.

For most applications, maintainability is more important that reusability. If the code in an application must change, you would rather the changes occur all in one component than being distributed through many components. If changes are focused into a single component, then we need only redeploy the one changed component. Other components that don't depend upon the changed component do not need to be revalidated or redeployed.

The CCP prompts us to gather together in one place all the modules that are likely to change for the same reasons. If two modules 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.

This principle is closely associated with the Open–Closed Principle (OCP). For it is "closure" in the OCP sense of the word that this principle is dealing with. The OCP states that modules should be closed for modification but open for extension. But 100% closure is not attainable. Closure must be strategic. We design our modules such that they are closed to the most common kinds of changes that we expect or have experienced.

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

Similarity with SRP

As I stated above, the CCP is the component form of the SRP. The SRP tells us to separate methods into different modules if they change for different reasons. The CCP tells us to separate modules into different components if they change for different reasons.

Different Times?

The caption at the start of this section implies that a component should not have elements that change at different times, even though they might change for the same reasons. Some changes are very high priority. Other changes are lower priority. Some changes are volatile with a high frequency. Some have a much lower volatility. We'd like to avoid components that have both high- and low-volatility elements.

The Common Reuse Principle (CRP)

Don't force users of a component to depend on things they don't need.

This is yet another principle that helps us decide which modules should be placed into a component. It states that classes and modules that tend to be

reused together belong in the same component.

Modules are seldom reused in isolation. Generally, reusable modules collaborate with other modules that are part of the reusable abstraction. The CRP states that these modules belong together in the same component. In such a component, we would expect to see modules that have lots of dependencies upon each other.

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

But the CRP tells us more than just what modules to put together into a component. It also tells us what modules not to keep together in a component. When one component uses another, a dependency is created between the components. It may be that the using component only uses one module within the *used* component; but that doesn't weaken the dependency at all. The using component still depends upon 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 may even be true if the change to the used component is something that the using component doesn't care about.

Thus, we want to make sure that when we depend upon a component, we depend upon the majority of modules in that component. To say this another way, we want to make sure that the modules that we put into a component are inseparable; that it is unlikely that we can depend upon some and not the others. Otherwise, we will be redeploying more components than is necessary, and wasting significant effort.

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

Relation to ISP

The CRP is the generic version of the ISP. The ISP advises us not to depend upon classes that have methods we don't use. The CRP advises us not to depend upon components that have modules we don't use. All of this advice can be reduced to a single sound bite:

Don't depend on things you don't need.

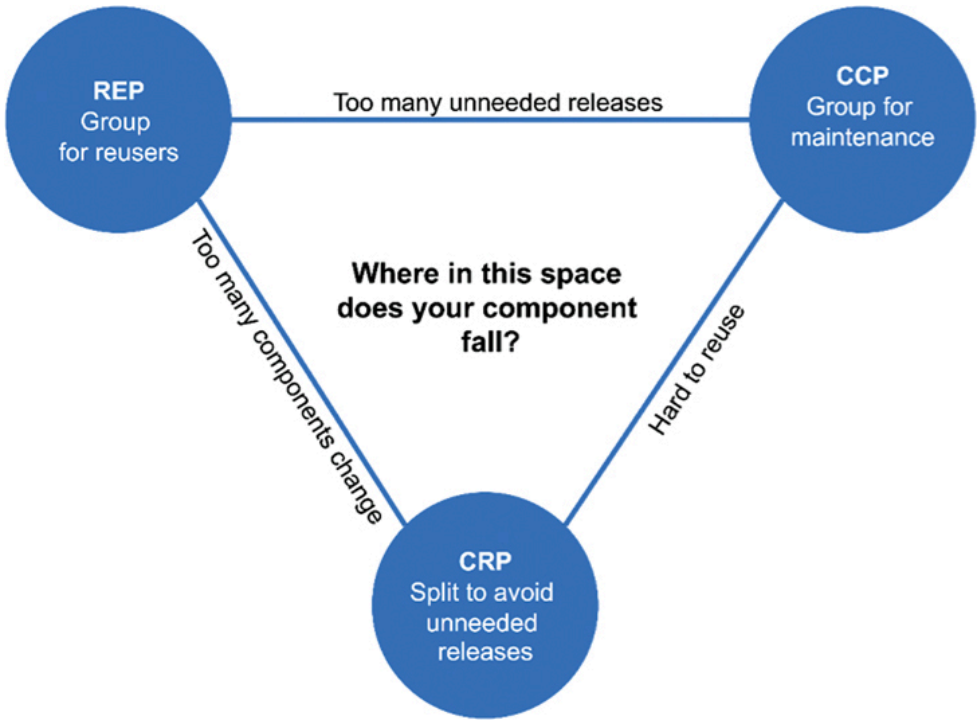
The Tension Diagram for Component Cohesion

You may have already realized that the three cohesion principles tend to fight each other. The REP and CCP are inclusive principles. They both tend to make components larger. The CCP is an *exclusive* principle, driving components to be smaller. It is the tension between these principles that we seek to resolve.

Below is a tension diagram[6] 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.

---

6. Thanks to Tim Ottinger for this idea.



So, a programmer who focuses on the REP and the CRP will find that too many components are impacted when simple changes are made. An architect who focuses too strongly on the CCP and the REP will cause too many unneeded releases to be made.

The goal is to find a position in that tension triangle that meets the *current* concerns of the development team, and shift that position as the system evolves over time. For example, early in the development of a project, the CCP is much more important than the REP, because developability is more important than reuse.

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 the 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.

In Summary…

In the past, our view of cohesion was much simpler than the last three principles have implied. We used to think that cohesion was simply the attribute of a module to perform one, and only one, function. However, the three principles of component cohesion describe a much more complex variety of cohesion. In choosing the modules to group together into components, we must consider the opposing forces involved in reusability and developability. 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 today might not be appropriate next year. Thus, the composition of the components will likely jitter and evolve with time as the focus of the project changes from developability to reusability.

This means that the component structure of the system cannot be determined up front, and must incrementally evolve as the system grows and changes. And this, of course, means that the changes to the component structure will be small but frequent. They become a normal part of any Agile iterative cycle.

Component Coupling

The next three principles deal with the relationships between components. Here again we will run into the tension between developability and logical design. The forces that impinge upon the architecture of a component structure are technical, political, and volatile.

The Acyclic Dependencies Principle (ADP)

Allow no cycles in the component dependency graph.

Have you ever worked all day, gotten some stuff working and then gone home, only to arrive the next morning to find that your stuff no longer works? Why doesn't it work? Because somebody stayed later than you and changed something you depend upon! I call this *the morning-after syndrome*.

The morning-after syndrome occurs in development environments where many developers are modifying the same source files. In relatively small projects with just a few developers, it isn't too big a problem. But as the sizes of the project and the development team grow, the mornings after can get pretty nightmarish. It is not uncommon for weeks to go by without being able to build a stable version of the project. Instead, everyone keeps on changing and changing their code, trying to make it work with the last changes that someone else made.

Eliminating Dependency Cycles

The solution to this problem is to partition the development environment into releasable components. The components become units of work, which can be the responsibility of a developer or a team of developers. When developers get a component 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.

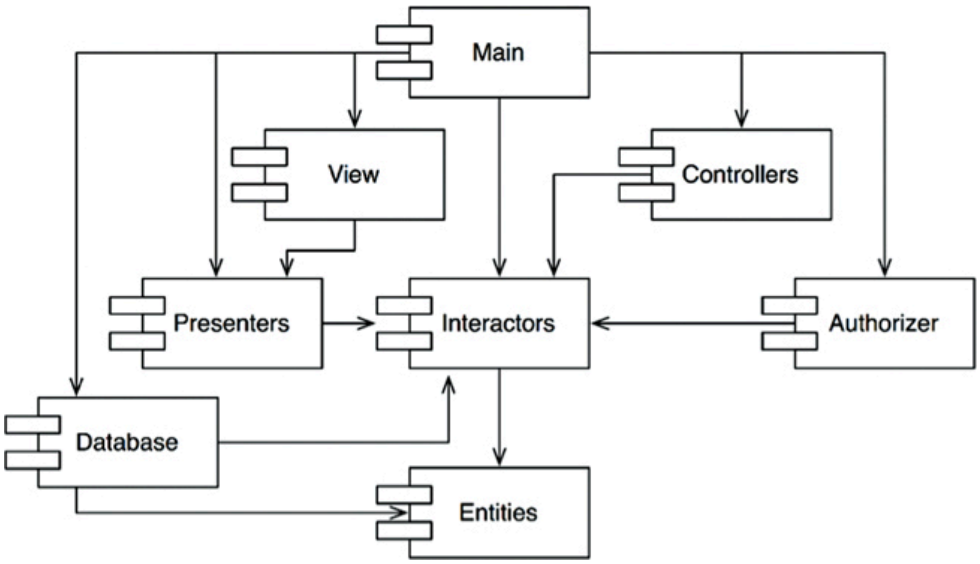
As new releases of a component are made, other teams can decide whether or not to immediately adopt the new release. If they decide not to, they simply continue using the old release. Once they decide that they are ready, they begin to use the new release.

Thus, none of the teams are at the mercy of the others. Changes made to one component do not need to have an immediate effect on other teams. Each team can decide for itself when to adapt its components to new releases of the components they use. Moreover, integration happens in small increments. There is no single point in time when all developers must come together and integrate everything they are doing.

This is a very simple and rational process; and it is widely used. However, to make it work you must manage the dependency structure of the components. There can be no cycles.

If there are cycles in the dependency structure, then the morning-after syndrome cannot be avoided.

Consider the following component diagram.



Here we see a rather typical structure of components assembled into an application. The function of this application is unimportant for the purpose of this example. What *is* important is the dependency structure of the components. Notice that this structure is a directed graph. The components are the nodes, and the dependency relationships are the directed edges.

Now notice one more thing. Regardless of which component you 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).

Now notice what happens when the team responsible for `Presenters` makes a new release of their component. It is easy to find out who is affected by this release; you just follow the dependency arrows backward. Thus, `View` and `Main` are both going to be affected. The developers currently working on those components will have to decide when they should integrate with the new release of `Presenters`.

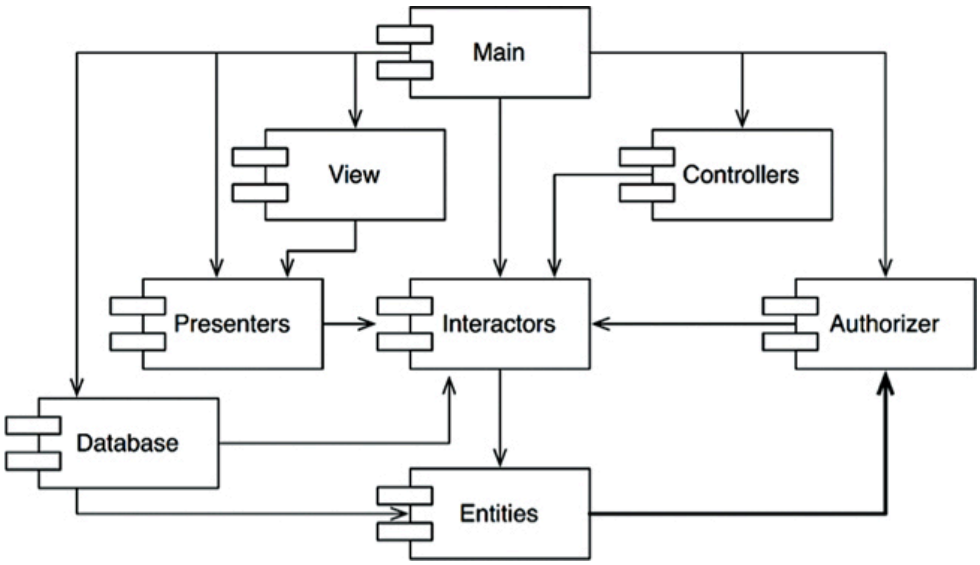
Notice also that when `Presenters` is released, it has utterly no effect upon many of the other components in the system. They don't know about `Presenters`, and they don't care when it changes. This is nice. It means that the impact of releasing `Presenters` is relatively small.

When the developers working on the `Presenters` component would like to run a test of that component, all they need to do is 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. This is nice. It means that the developers working on `Presenters` have relatively little work to do to set up a test and that there are relatively few variables for them to consider.

When it is time to release the whole system, it is done from the bottom up. First, the `Entities` component is compiled, tested, and released; then, `Database` and `Interactors`. These are followed by `Presenters`, `View`, `Controllers`, and then `Authorizer`. `Main` is last. This process is very clear and easy to deal with. We know how to build the system because we understand the dependencies between its parts.

The Effect of a Cycle in the Component Dependency Graph

Let us say that a new requirement forces us to change one of the modules in `Entities` such that it makes use of a module in `Authorizer`. For example, let's say that the `User` class in `Entities` uses the `Permissions` class in `Authorizer`. This creates a dependency cycle, as shown below.

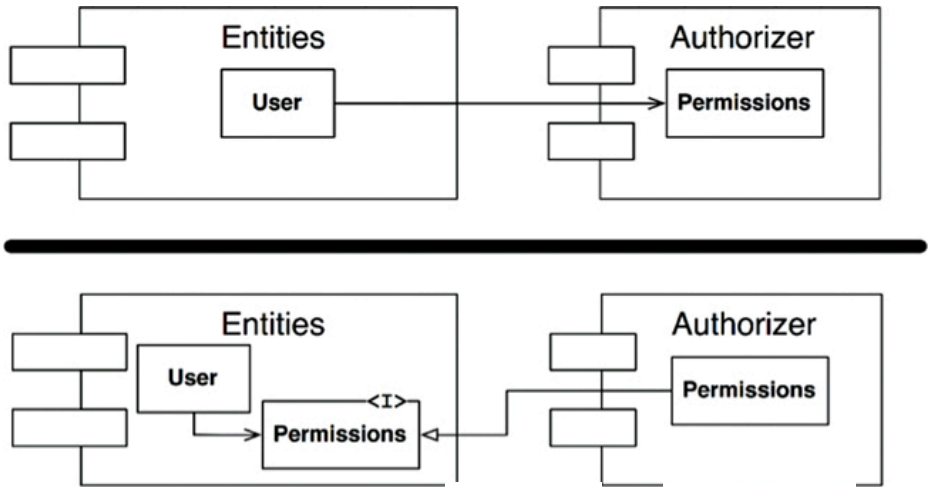


This cycle effectively combines `Interactors`, `Entities`, and `Authorizer` into a single component. Any component that uses one depends upon them all. And this makes building and testing much more difficult. It also begs the question of the correct build order. You can't build `Entities` first, because it depends upon `Authorizer`. But `Authorizer` depends upon `Entities` through `Interactors`. So there is no correct build order. Any system built without a correct build order functions by accident rather than by plan.

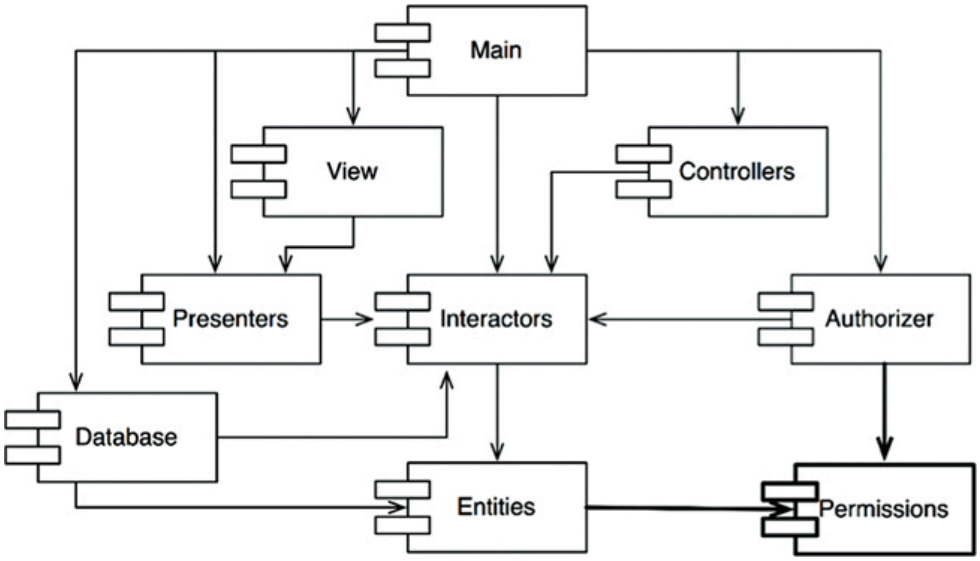
Breaking the Cycle

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

1. Apply the Dependency Inversion Principle (DIP). In the case below, we could create an interface that has the methods that `User` needs. We could then put that interface into `Entities` and inherit it into `Authorizer`. This inverts the dependency between `Entities` and `Authorizer`, thus breaking the cycle. See the before-and-after view next.



2. Create a new component that both `Entities` and `Authorizer` depend upon. Move the module(s) that they both depend upon into that new component. See below.



The “Jitters”

The second solution implies that the component structure is volatile in the presence of changing requirements. Indeed, as the application grows, 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.

Top-Down Design

The issues we have discussed so far lead to an inescapable conclusion. The component structure cannot be designed from the top down. This means that is it not one of the first things about the system that is designed. Indeed, it evolves as the system grows and changes.

Some of you may find this to be counterintuitive. We have come to expect that large-grained decompositions, like components, are also high-level functional decompositions.

When we see a large-grained grouping, such as a component dependency structure, we feel that the components ought to somehow represent the functions of the system. Yet this does not seem to be an attribute of component dependency diagrams.

In fact, component dependency diagrams have very little do to with describing the function of the application. Instead, they are a map to the buildability and maintainability of the application. This is why they aren't designed at the start of the project. There is no software to build or maintain, and so there is no need for a build and maintenance map. But as more and more modules accumulate in the early stages of implementation and design, there is a growing need to manage the dependencies so that the project can be developed without the morning-after syndrome. Moreover, we want to keep changes as localized as possible, so we start paying attention to the SRP and the CCP and collocate modules that are likely to change together.

One of the overriding concerns of this dependency structure is the isolation of volatility. We don't want components that change frequently and for capricious reasons to affect components that otherwise ought to be stable. For example, we don't want cosmetic changes to the GUI to have an impact on our business rules. We don't want the addition or modification of reports to have an impact on our highest-level policies. So the component dependency graph is created and molded by developers to protect stable high-value components from volatile lower-level components.

As the application continues to grow, we start becoming concerned about creating reusable elements. That is when the CRP begins to influence the composition of the components. Finally, as cycles appear, the ADP is applied and the component dependency graph jitters and grows.

If we were to try to design the component dependency structure before we had designed any modules, we would likely fail rather badly. We would not know much about common closure, we would be unaware of any reusable elements, and we would almost certainly create components that produced

dependency cycles. Thus, the component dependency structure grows and evolves with the logical design of the system.

The Stable Dependencies Principle (SDP)

Depend in the direction of stability.

Designs cannot be completely static. Some volatility is necessary if the design is to be maintained. We accomplish this by conforming to the CCP. Using this principle, 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 upon by a component that is difficult to change! Otherwise, the volatile component will also be difficult to change.

It is the perversity of software that a module that you have designed to be easy to change can be made hard to change by someone else who simply hangs a dependency upon it. Not a line of source code in your module need change; and yet your module will suddenly be hard to change. By conforming to the SDP, we ensure that modules that are intended to be easy to change are not depended upon by modules that are harder to change.

Stability

What is meant by stability? Stand a penny on its side. Is it stable in that position? You'd likely say that it was not. However, unless disturbed, it will remain in that position for a very long time. Thus, stability has nothing directly to do with *frequency* of change. The penny is not changing, but it is hard to think of it as stable.

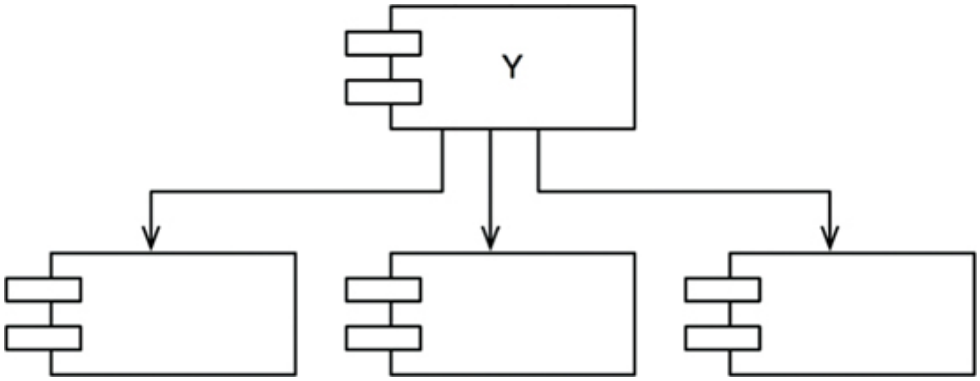
Webster says that something is stable if it is "not easily moved." Stability is related to the amount of work required to make a change. The penny is not very stable, because it requires very little work to topple it. On the other hand, a table is very stable because it takes a considerable amount of effort to turn it over.

Stability is a continuum, not a boolean value.

How does this relate to software? There are many factors that make a software component hard to change—its size, complexity, clarity, and so on. We are going to ignore all those factors and focus upon something different. One sure way to make a software component difficult to change is to make lots of other software components depend upon it. 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.

The following diagram shows X, a stable component. This component has three components depending upon it, and therefore it has three good reasons not to change. We say that it is responsible to those three components. On the other hand, X depends upon nothing, so it has no external influence to make it change. We say that X is responsible and independent: It is an adult.

The next diagram, on the other hand, shows Y, a very unstable component. Y has no other components depending upon it; we say that it is irresponsible. Y also has three components that it depends upon, so changes may come from three external sources. We say that Y is dependent and irresponsible: It is a teenager.



Stability Metrics

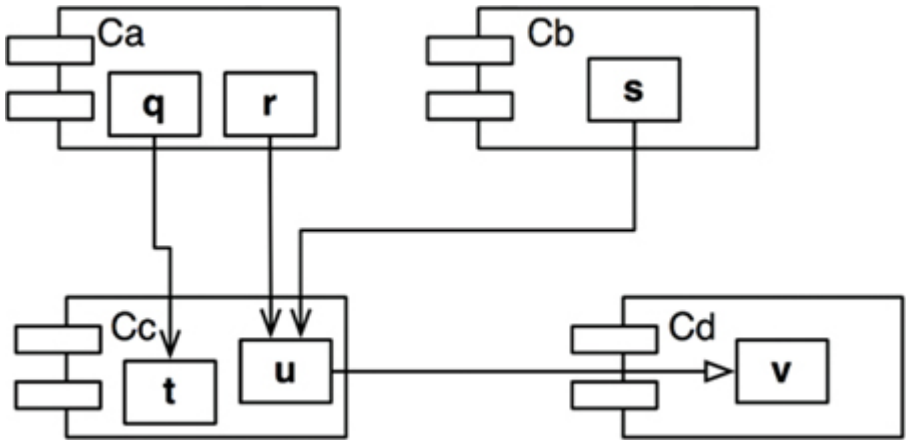
How can we measure the stability of a component? One way is to count the number of dependencies that enter and leave that component. These counts will allow us to calculate the positional stability of the component.

- Fan-in: Incoming Couplings—This is the number of modules outside this component that depend upon modules within this component.
- Fan-out: Outgoing Couplings—This is the number of modules inside this component that depend upon modules outside this component.
- *I*: 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 Fan-in and Fan-out metrics[7] are calculated by counting the number of modules outside the component in question that have dependencies with the modules inside the component in question. Consider the example below:

---

7. In previous publications I used the names Efferent and Afferent couplings (Ce, and Ca) for Fan-out and Fan-in respectively. That was just hubris on my part … I liked the metaphor of the central nervous system.



Let's say we want to calculate the stability of the component Cc. We find that there are three modules outside Cc that depend upon modules in Cc. Thus, Fan-in = 3. Moreover, there is one module outside Cc that modules in Cc depend upon. Thus, Fan-out = 1, and *I* = 1/4.

In C++, these dependencies are typically represented by `#include` statements. Indeed, the *I* metric is easiest to calculate when you have

organized your source code such that there is one module in each source file. In Java, the *I* metric can be calculated by counting `import` statements and qualified names. Other languages have similar mechanisms such as `using` or `require`.

When the *I* metric is 1, it means that no other component depends upon this component (Fan-in = 0); and this component depends upon other components (Fan-out > 0). Thus, it is as unstable as a component can get; it is a teenager, irresponsible and dependent. Its lack of dependents gives it no reason not to change, and the components that it depends upon may give it ample reason to change.

On the other hand, when the *I* metric is zero, it means that the component is depended upon by other components (Fan-in > 0), but it does not itself depend upon any other components (Fan-out = 0). It is a responsible and independent adult. Such a component is as stable as it can get. Its dependents make it hard to change, 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 upon.

I metrics should decrease in the direction of dependency.

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 desirable situation. Indeed, we want to design our component structure so that some components are unstable and some are stable. The following diagram shows a conformant configuration for a system with three components.

The changeable components are on top and depend upon the stable component at the bottom. Putting the unstable components at the top of the diagram is a useful convention since any arrow that points *up* is violating the SDP (and, as we shall see in the next chapter, the ADP).

The next diagram shows how the SDP can be violated.

`Flexible` is a component that we have designed to be easy to change. We want `Flexible` to be unstable. However, some developer, working in the component named `Stable`, has hung a dependency upon `Flexible`. This violates the SDP since the *I* metric for `Stable` is much lower than the *I* metric for `Flexible`. As a result, `Flexible` will no longer be easy

to change. A change to `Flexible` will force us to deal with `Stable` and all its dependents.

To fix this, we somehow have to break the dependence of `Stable` upon `Flexible`. Why does this dependency exist? Let's assume that there is a module `C` within `Flexible` that another module `U` within `Stable` needs to use.

We can fix this by employing the DIP. We create an interface called `US` and put it in a component named `UServer`. We make sure that this interface declares all the methods that `U` needs to use. We then make `C` implement this interface, as shown below. This breaks the dependency of `Stable` upon `Flexible`, and it forces both components to be dependent upon `UServer`. `UServer` is very stable ($I = 0$), and `Flexible` retains its necessary instability ($I = 1$). All the dependencies now flow in the direction of decreasing I.

Abstract Components

You may find it strange that we have created a component, like `UService`, that contains nothing but an interface. Such a component contains no executable code! It turns out, however, that this is a very common, and necessary, tactic when using statically typed languages like

Java or C#. These abstract components are very stable and are therefore ideal targets for less stable components to depend upon.

When using dynamically typed language like Ruby or Python, these abstract components don't exist at all. Nor do the dependencies that would have targeted them. Dependency structures in these languages are much simpler because dependency inversion does not require either the declaration or the inheritance of interfaces.

The Stable Abstractions Principle (SAP)

A component should be as abstract as it is stable.

What follows is going to get a bit mathematical and theoretical. Don't let that scare you off. The theory is important, even if the practice is only loosely bound to it. Read through to the end and you'll see what I mean.

Where Do We Put the High-Level Policy?

Some software in the system should not change very often. This software represents high-level architecture and policy decisions. We don't want these business and architectural decisions to be volatile. Thus, the software that encapsulates the high-level policies of the system should be placed 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.

However, if the high-level policies are placed into stable components, then the source code that represents those policies will be difficult to change. This could make the overall architecture inflexible. How can a component that is maximally stable ($I = 0$) be flexible enough to withstand change? The answer is to be found in the OCP. This principle tells us that it is possible and desirable to create modules that are flexible enough to be extended without requiring modification. What kinds of modules conform to this principle? Abstract modules.

The SAP sets up a relationship between stability and abstractness. It says that a stable component should also be abstract so that its stability does not prevent it from being extended. On the other hand, it says 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 also consist of interfaces and abstract modules so that it can be extended. Stable components that are extensible are flexible and do not overly constrain the architecture.

The combination of the SAP and the SDP amounts to the DIP for components. This is true because 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.

Measuring Abstraction

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

- Nm: The number of modules in the component.
- Na: The number of abstract modules and interfaces in the component.
- A: Abstractness.
  A = Na ÷ Nm. The *A* metric ranges from 0 to 1. Zero implies that the component has no abstract classes at all. A value of 1 implies that the component contains nothing but abstract modules and/or interfaces.

The Main Sequence

We are now in a position to define the relationship between stability (*I*) and abstractness (*A*). We can create a graph with *A* on the vertical axis and *I* on the horizontal axis. If we plot the two “good” kinds of components on this graph, we will find the components that are maximally stable and abstract (the adults) at the upper left at (0,1). The components that are maximally unstable and concrete (the teenagers) are at the lower right at (1,0).

Not all components can fall into one of these two positions. Components have degrees of abstraction and stability. For example, it is very common for one abstract module to derive from another abstract module. The derivative is an abstraction that has a dependency. Thus, though it is maximally abstract, it will not be maximally stable. Its dependency will decrease its stability.

Since we cannot enforce 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 can infer what that locus is by finding the areas where components should not be—that is, zones of exclusion.

The Zone of Pain

Consider a component in the area of (0,0). This is a highly stable and concrete component. Such a component is not desirable, 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 components sitting near (0,0). The area around (0,0) is a zone of exclusion called the *Zone of Pain*.

It should be noted that certain software entities do, in fact, fall within the Zone of Pain. An example would be a database schema. Database schemas are notoriously volatile, extremely concrete, and highly depended upon. This is one of the reasons that the interface between `OO` applications and databases is so difficult, and that schema updates are generally painful.

Another example of software that sits on (0,0) is a concrete utility library. Although such a library has an *A* metric of 0, it may in fact be nonvolatile. Consider the `String` component, for example. Even though all the classes within it are concrete, it is so commonly used that changing it would create chaos. Therefore, `String` is nonvolatile.

Nonvolatile components are harmless in the (0,0) zone since they are not likely to be changed. For that reason, it is only software components that are volatile that are also problematic in the Zone of Pain. The more volatile a component in the Zone of Pain is, the more "painful" it is. Indeed, we might consider volatility to be a third axis of the graph; and then the graph above shows only the most painful plane, where volatility = 1.

The Zone of Uselessness

Consider a component near (1,1). This location is undesirable because it is maximally abstract and yet has no dependents. This is like an interface that has no implementation. Such components are useless. Thus, this 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. We find them in systems from time to time, sitting in the codebase, unused.

Or they may be superfluous interfaces—interfaces that a few classes implemented but no class ever used.

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

The Main Sequence

We'd like our most volatile components to be as far from both zones of exclusion as possible. The locus of points that is maximally distant from each zone is the line that connects (1,0) and (0,1). I call this line the *main sequence*.[8]

---

8. The author begs the reader's indulgence for the arrogance of borrowing such an important term from astronomy.

---

9. In previous publications, I called this metric D'. I see no reason to continue that practice.

A component 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 upon to the extent that it is abstract, and it depends upon others to the extent that it is concrete.

The most desirable positions for a component are at one of the two endpoints of the main sequence. We strive to position the majority of 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. Those components have the best characteristics if they are on or close to the main sequence.

Distance from the Main Sequence

This leads us to our last metric. If it is desirable for components to be on or close to the main sequence, we can create a metric that measures how far away a component is from this ideal.

- D[9]: Distance. $D = |A+I-1|$. This metric ranges from [0,1]. Zero indicates that the component is directly on the main sequence. One indicates that the component is as far away as 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.

Conclusion

If your head is reeling at this point, I suggest you breathe. I am not suggesting that you should measure and calculate all these metrics. It’s good to know that F = ma, but you don’t often do that math when you drive your car. Theory is one thing. Practice is another.

I certainly don’t make a practice of gathering the necessary statistics, doing all the math, and generating plots. In most projects, I can *see* these relationships in my head and do a simple mental evaluation—and I don’t mean that I’m doing the math in my head either. Rather, I simply envision

the relationships and infer how stable and how abstract each component is. If you wish, consider it a “feeling.”

It’s good to understand the above theory. There are cases where it may even be beneficial to do the above calculations. There have been a few extreme cases where I certainly have. But in most cases, understanding the theory and using it to “feel” the relationships is good enough.

You may find that your IDE does many of these calculations for you. I’ve found it helpful, in larger projects, to look over the numbers that the IDE has calculated. But my “feel” is pretty good, and I am seldom surprised.