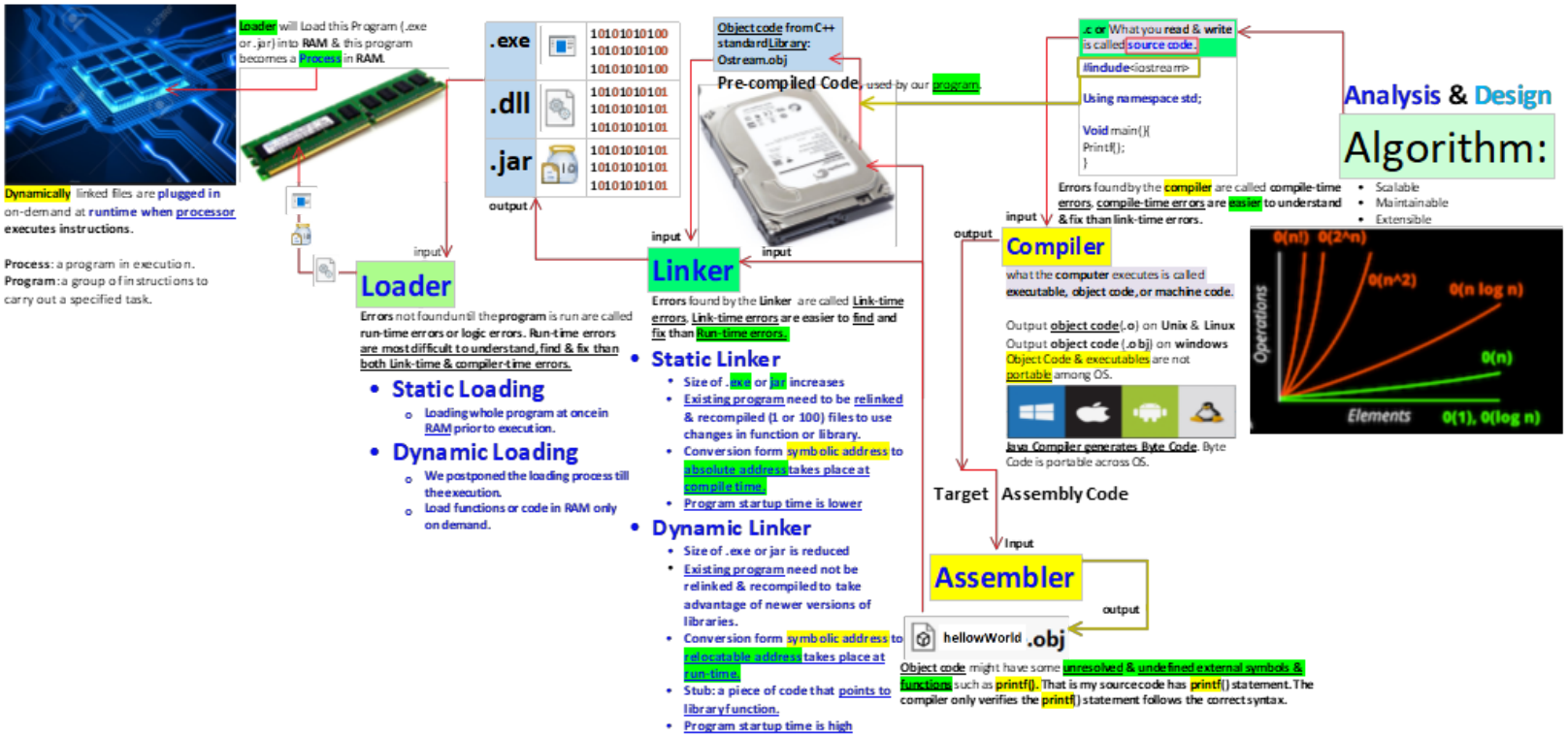
Components

Thursday, January 2, 2020 8:48 AM

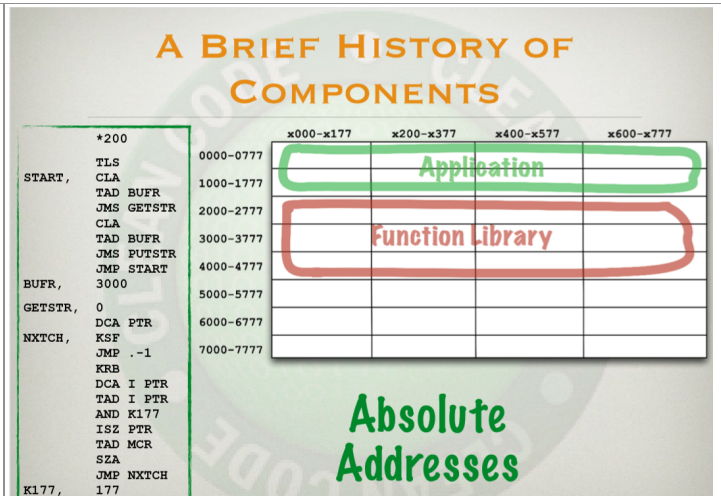


If the SOLID principles tell us how to arrange the bricks into walls and rooms, then the component principles tell us how to arrange the rooms into buildings. Large software systems, like large buildings, are built out of smaller components.

Components are:

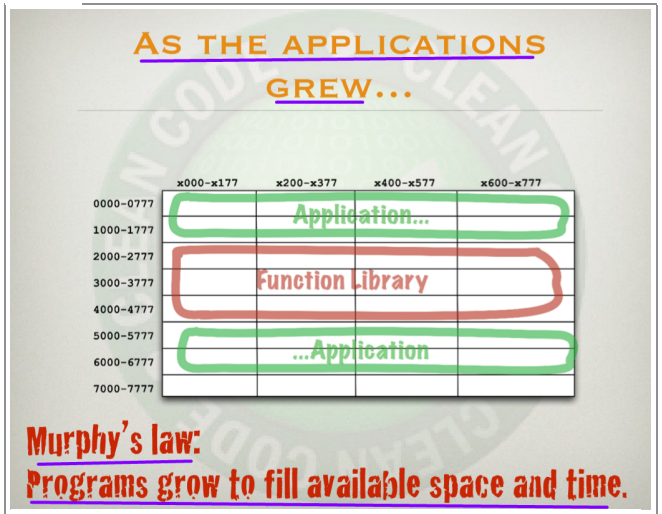
• Components are linked, **statically** or **dynamically**, into an **executable**.

- 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**.



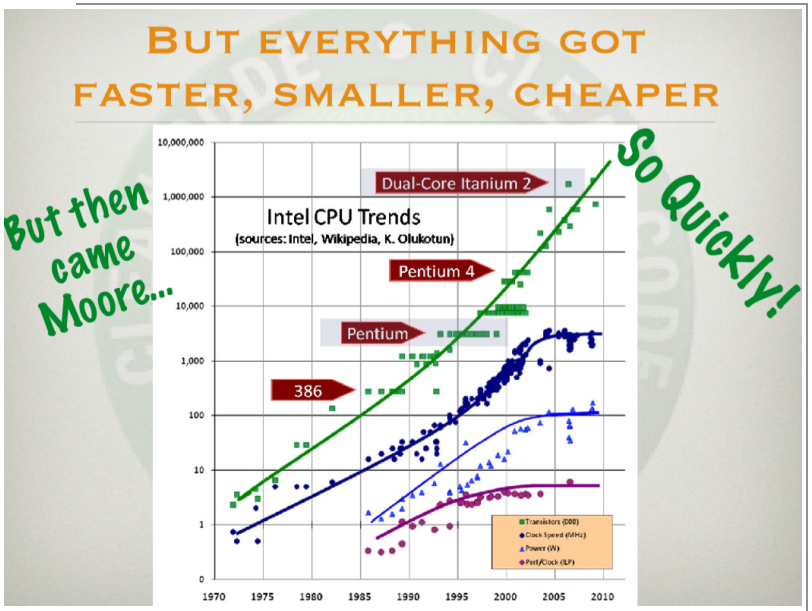
- In Ruby, they are **gem files**.
- In .NET, they are **DLLs**.
- In **compiled languages**, they are **aggregations(collection) of binary files**.
- In **interpreted languages**, they are **aggregations of source files.**
- In **all languages,** they are the **granule of deployment.**

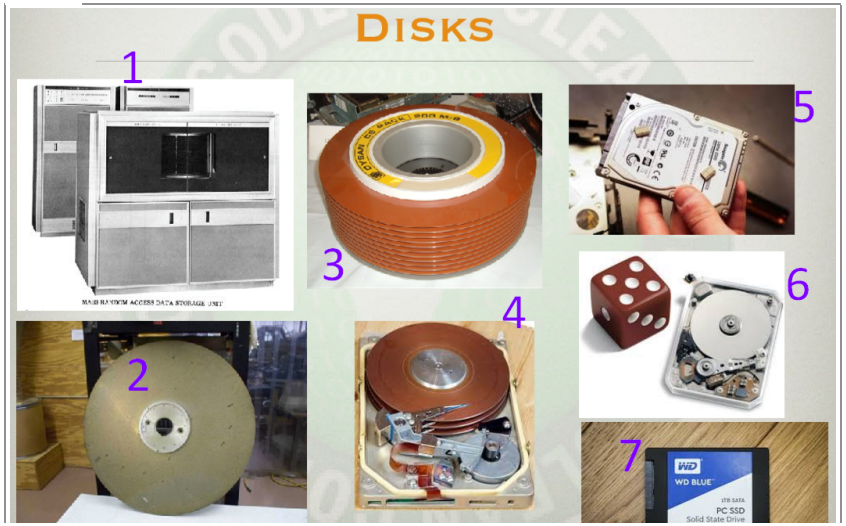
```
MCR,     -15
```



Interim(temporary) Solutions to battle Murphy:

- Relocate-ability
- Linking loaders — got slow.
- Linkers. — Better, still slow.





Murphy loses!

- **Murphy**: A program will grow to fill all available **space and time.**
- **Moore,** time and space will double every 18 months.
- Moore won!
- Mid '90s: Link time shrank back down to load time.

True Dynamically Loadable Components.

- Active X. DLLs.
- Shared libraries.
- JAR files.
- Dynamic linking rules everywhere.

Package separability - dependencies between different packages should be minimized so that packages can be deployed in different modules. In particular, there should be no circular dependencies between packages.

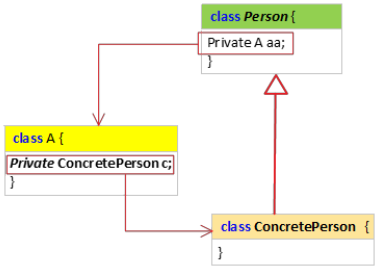
Interface separability - dependencies between **abstract classes and interfaces** and their **implementing concrete types** should be minimized, so that abstract types and implementation types can be part of different modules. This facilitates the compatibility of different implementations and makes it easier to replace a particular implementation within an application.

The question arises how existing applications can be refactored to modular designs based on one of these platforms.

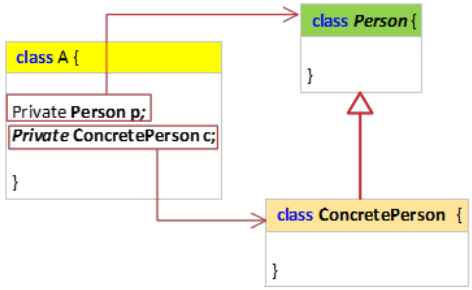
The State of Affairs

To answer this question, we have investigated a large set of open-source Java programs (the qualitas corpus) in order to find out how many of those programs suffer from dependency related problems. The short answer is: almost all of them. In this experiment, we checked the dependency graph extracted from the respective program for instances of the following anti-patterns which compromise package and interface separability, respectively:

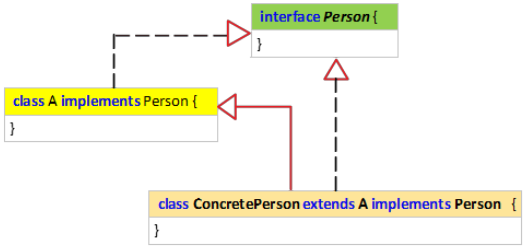
1. ***Strong circular dependencies between packages (CD)***: dependency chains starting in a package A, traversing some other packages and the returning into A. This is a strong version of circular dependency caused by one reference chain that creates the package dependencies. In particular, this pattern cannot be broken by splitting packages.
2. ***Strong circular dependencies between jars (CDC)***: dependency chains starting in a **jar A**, traversing some other **jars** and the returning into **A**.
3. ***Subtype knowledge (STK)*:** super-types (classes or interfaces) (indirectly) referencing their own subtypes.



4. ***Abstraction without decoupling (AWD)*:** classes referencing both abstract types and their implementation types.



3. ***Degenerated inheritance (DEGINH)*:** multiple paths from subtypes to super types (in Java, this is possible because **interfaces** support multiple inheritance).



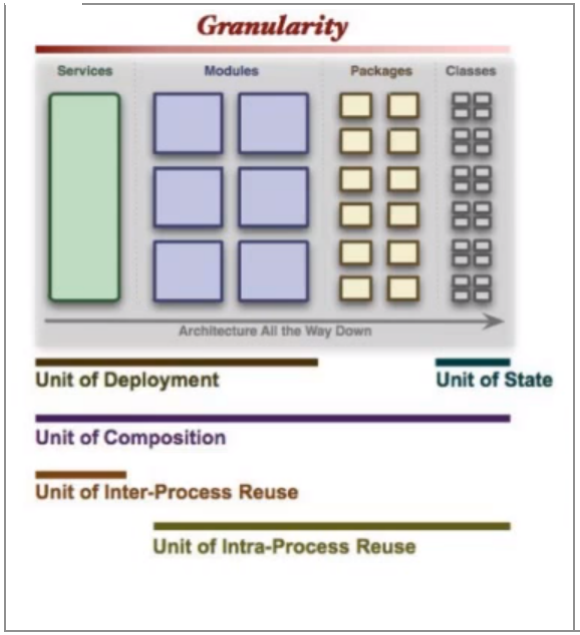
Detecting Dependency-Related Problems

There are a number of **tools** that can extract, display and analyze the dependency graph. There are two approaches to detect critical dependencies: **metrics** and **patterns**.

Metrics associate artifacts and their relationships with numerical values expressing quality. a metric for packages. "It states that **abstract packages** should have relatively many incoming dependencies (high responsibility), while **concrete (implementation) packages** should have relatively few incoming dependencies but will depend on other packages (high instability and low responsibility)". The classic tool to detect the D metric is JDepend. JDepend computes several package dependency metrics, and can be easily embedded into IDEs and build scripts. It also supports the detection of circular dependencies between packages.

Module Defined A software module is a deployable, manageable, natively reusable, compose-able, stateless unit of software that provides a concise interface to consumers. On the Java platform, a module is a JAR file, as depicted in the diagram. The **module design patterns** help you design modular software and realize the

benefits of modularity.



In Java Module is :
- .Jar
- .Ear
- .War

In C# Module is:
- Assembly file
- .dll

Modules, Packages, Classes	**Unit of Intra-Process Reuse** **Intra (within, inside)**
Services	**Unit of Inter-Process Reuse**
Services ,Modules, Packages, Classes	**Unit of Composition**
Services or Modules or Both	**Unit of Deployment**
Classes	**Unit of state** directly above the **Code**.
Code	

It's critical to keep in mind that modularizing a software system without runtime module support will be challenging. In the absence of a runtime system, the tools you'll use are critical. **Visualization** tools that help you understand your **structure**, **build** tools that help enforce the **structure**, and **dependency management** tools that help you **manage dependencies** are a necessity.

Additionally, the patterns in this book help you modularize a system. For instance, the Levelize Build pattern helps you enforce your module relationships at **compile time**. With good tools and strong discipline, modular architecture is something you can achieve. Furthermore, after your modular architecture is in place, you can take advantage of runtime support once it's available. However, without runtime support, you are simply unable to accomplish some things. A runtime module system often provides support for the following:

- **Encapsulation**

  In standard Java, any public class in a package found on the **class-path** is available to anything else on the **class-path**. In other words, there is no way to hide implementation details. Everything is global, which impedes modular design. A runtime module system provides the ability to hide implementation details. For instance, OSGi uses µServices that expose an interface to the underlying implementation details.

- **Dynamic deployment**: In standard Java, updating the software typically involves restarting the JVM. A runtime module system supports hot deploys.
- **Versioning**: In standard Java, it's not possible to deploy multiple versions of a class. A module system allows for multiple versions to be deployed.
- **Dependency management**: In standard Java, there is no way to enforce the dependency structure of the modules. Build tools such as Maven attempt to solve this problem through repositories of JAR files that describe their dependencies. A runtime module system enforces runtime dependency management.

Conclusion There are two facets to modularity: the runtime model and the development model. **The development model is comprised** of the **programming model and the design paradigm**. All facets are important, but failing to understand how to design modular software will detract from the benefits of using a modular runtime or framework.

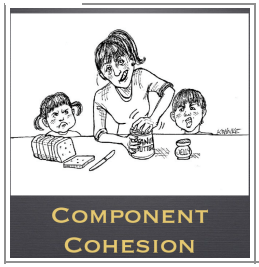
The Two Facets of Modularity consists of two aspect:

1. The **runtime model**
2. The **development model**.

The Development Model The development model deals with how developers use the framework to build their software applications. The development model can be further broken down into two categories:

1. the programming model (SP, FP, OOP)
2. the design paradigm.

Nowadays, emphasis is on the runtime model, with frameworks emerging that provide runtime support for modularity. But eventually, as the runtime model gains adoption, the importance of the **development model** will take center stage. The patterns in this book focus on an aspect of the development model referred to as the ***design paradigm***.	Each is important in helping developers build more modular software applications. If the **design paradigm** isn't understood, with **principles** and **patterns** that guide how developers leverage the technology, the benefits of the runtime model will not be realized. Modularity is a key element of designing software systems with a flexible and adaptable architecture. There is a need for modularity on the Java platform, especially in developing large enterprise software systems. But if we do not begin to understand how to design more modular applications today, we'll face significant challenges when platform support for modularity arrives.



Principles that govern Component innards(insides, intestines, guts).

• **REP**, **CCP**, **CRP**. The criteria for deciding which **classes** and **modules** to group together, and which to keep separate.

REP: Release/Reuse Equivalence Principle.
• Maven et. al. The era of software reuse has truly begun.
• In order to reuse, you must release.
• Releasing is a human, clerical, activity.
• Classes and modules "reused" together.
• Weak principle without support of other two.

The Reuse/Release Equivalence Principle (REP) says:

> The unit of reuse is the unit of release. Effective reuse requires tracking of releases from a change control system. The package is the effective unit of reuse and release.

The unit of reuse is the unit of release

Code should not be reused by copying it from one class and pasting it into another. If the original author fixes any bugs in the code, or adds any features, you will not automatically get the benefit. You will have to find out what's changed, then alter your copy. Your code and the original code will gradually diverge.

Instead, code should be reused by including a released **library** in your code. The original author retains responsibility for maintaining it; you should not even need to see the source code.

Effective reuse requires tracking of releases from a change control system

The author of a library needs to identify releases with numbers or names of some sort. This allows users of the library to identify different versions. This requires the use of some kind of release tracking system.

The package is the effective unit of reuse and release

It might be possible to use a class as the unit of reuse and release, however there are so many classes in a typical application, it would be burdensome for the release tracking system to keep track of them all. A larger-scale entity is required, and the package fits this need well.

From Clean Architecture, by Robert Martin.

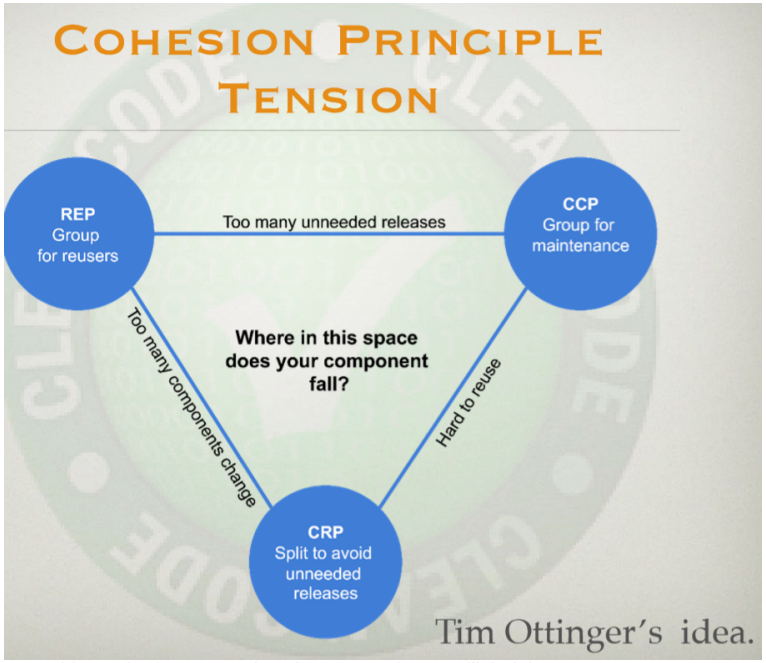
The Reuse/Release Equivalence Principle (REP) is a principle that seems obvious, at least in hindsight. People who want to reuse software components cannot, and will not, do so unless those components are tracked through a release process and are given release numbers.

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. 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.

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 the new release.

The Reuse/Release Equivalence Principle (REP) states that: The granule of reuse is the granule of release. Only components that are released through a tracking system can be

The Tension Diagram for Component Cohesion



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.

A good architect finds a position in that tension triangle that meets the current concerns of the development team, but is also aware that those concerns will change over time. For example, early in the development of a project, the CCP is much more important than the REP, because develop-ability is more important than reuse.

effectively **reused**. The granule is the package. (in other words REP you can reuse as much only as much you release.)

CCP: Common Closure Principle
- Gather together those things that change for the same reasons and at the same time.
- Separate those things that change for different reasons or at different times.
- in other words incorporate **SRP** for components.
- **Closure** is in the sense of **OCP**.

CRP: Common Reuse Principle
- Don't force the users of a component to depend on things they don't need.
- Negative cohesion. **CRP** tells us what not to include in a component.
- Related to **ISP**.

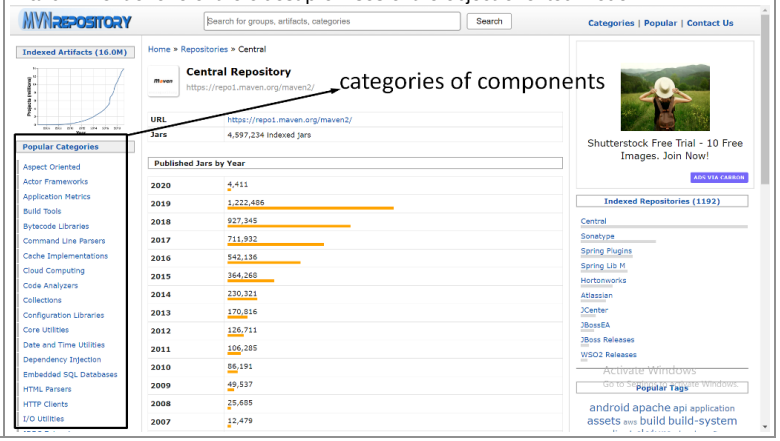
com/google/guava/28.1-jre

../		
guava-28.1-jre-javadoc.jar	2019-08-28 20:53	6669550
guava-28.1-jre-javadoc.jar.asc	2019-08-28 20:53	488
guava-28.1-jre-javadoc.jar.md5	2019-08-28 20:53	32
guava-28.1-jre-javadoc.jar.sha1	2019-08-28 20:53	40
guava-28.1-jre-sources.jar	2019-08-28 20:53	1643584
guava-28.1-jre-sources.jar.asc	2019-08-28 20:53	488
guava-28.1-jre-sources.jar.md5	2019-08-28 20:53	32
guava-28.1-jre-sources.jar.sha1	2019-08-28 20:53	40
guava-28.1-jre.jar	2019-08-28 20:53	2759505
guava-28.1-jre.jar.asc	2019-08-28 20:53	488
guava-28.1-jre.jar.md5	2019-08-28 20:53	32
guava-28.1-jre.jar.sha1	2019-08-28 20:53	40
guava-28.1-jre.pom	2019-08-28 20:53	8715
guava-28.1-jre.pom.asc	2019-08-28 20:53	488
guava-28.1-jre.pom.md5	2019-08-28 20:53	32
guava-28.1-jre.pom.sha1	2019-08-28 20:53	40

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.

Conclusion In the past, our view of cohesion was much simpler than the REP, CCP, and CRP implied. We once thought that cohesion was simply the attribute that a module performs one, and only one, function. However, the three principles of component cohesion describe a much more complex variety of cohesion. 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 today might not be appropriate next year. As a consequence, the composition of the components will likely jitter and evolve with time as the focus of the project changes from **develop-ability** to **reusability**.

The last decade has seen the rise of a menagerie of module management tools, such as **Maven**, **Leiningen**, and **Ruby Version Manager** (**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 the object-oriented model.



Apache Maven

Software

Maven is a build automation tool used primarily for Java projects. Maven can also be used to build and manage projects written in C#, Ruby, Scala, and other languages. The Maven project is hosted by the Apache Software Foundation, where it was formerly part of the Jakarta Project. Wikipedia

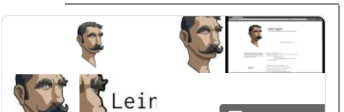
License: Apache License 2.0

Initial release: 13 July 2004; 15 years ago

Developed by: Apache Software Foundation

Written in: Java

Stable release: 3.6.3 / 25 November 2019; 2 months ago



Leiningen

Software

Leiningen is a build automation and dependency management tool for the simple configuration of software projects written in the Clojure programming language. Leiningen was created by Phil Hagelberg. Wikipedia

Developer(s): Jean Niklas L'orange

Stable release: 2.9.1 / February 26, 2019; 11 months ago

Written in: Clojure

Initial release: November 17, 2009

Original author(s): Phil Hagelberg

Ruby Version Manager

Software

Ruby Version Manager, often abbreviated as RVM, is a software platform for unix-like operating systems designed to manage multiple installations of Ruby on the same device. The entire ruby environment including the Ruby interpreter, installed RubyGems, and documentation is partitioned. Wikipedia

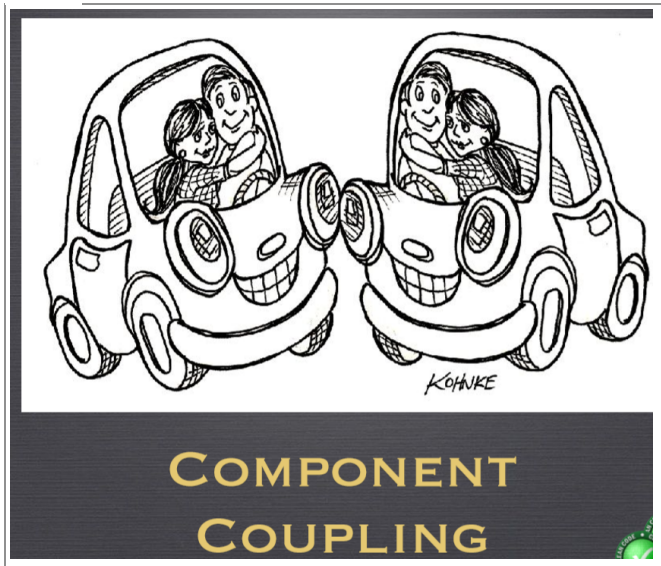
Initial release date: October 2007

Stable release: 1.29.6 / December 13, 2018; 13 months ago

Operating system: Unix-like

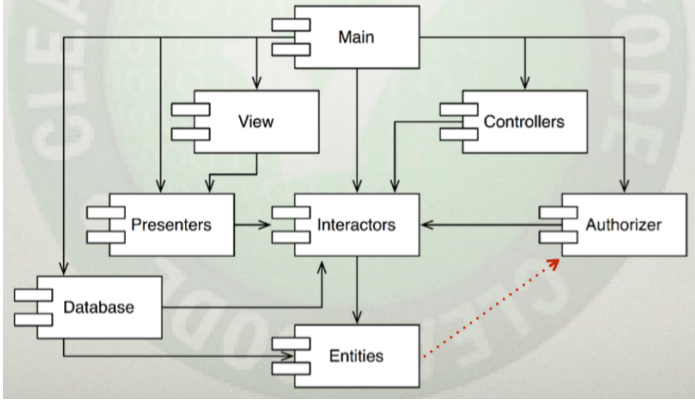
License: Apache License 2.0

Written in: Bash, Ruby

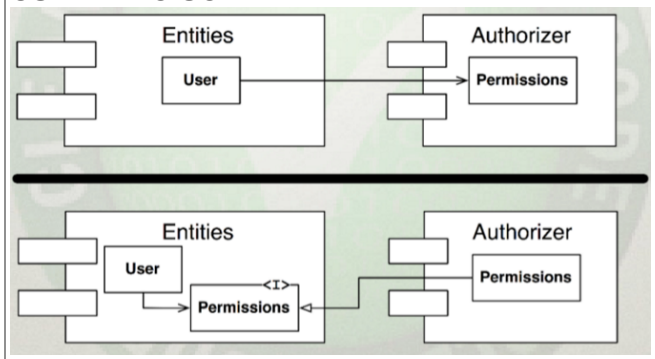


ADP: Acyclic Dependencies Principle

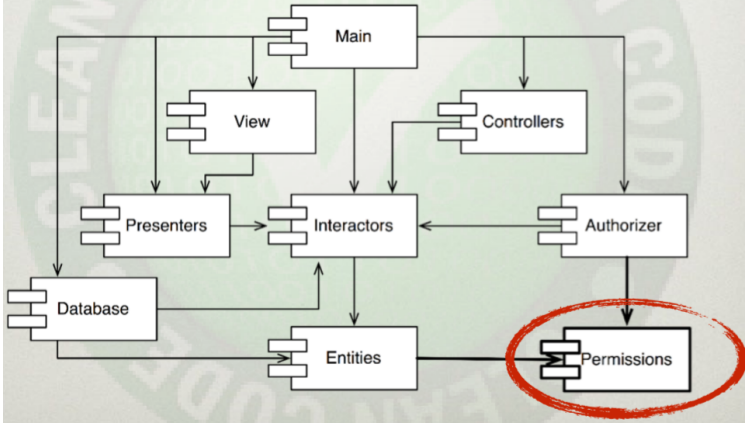
- The Morning After
- The Weekly Build



USE DIP TO SOLVE

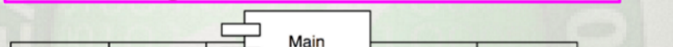


Create new component to solve



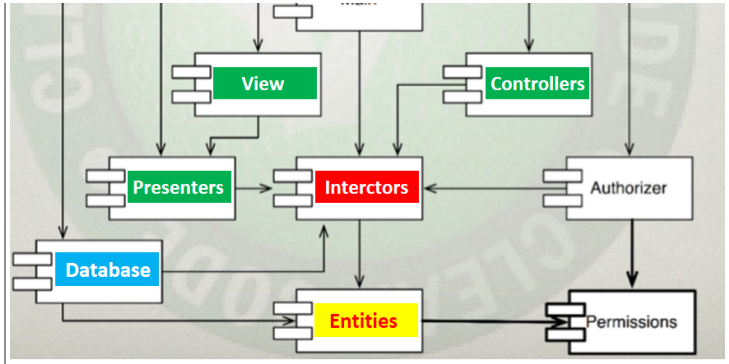
Not Top-Down Design

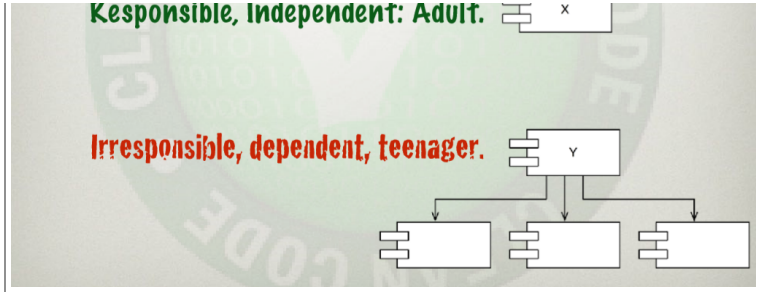
- The component graph evolves over time.
- It is not entirely the product of an up front design.



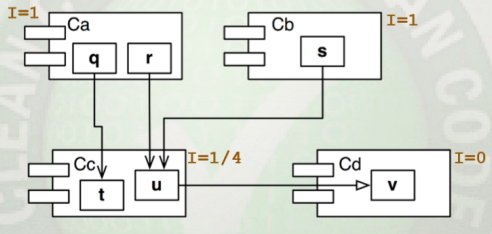
SDP: Stable Dependencies Principle

- What is Stability?





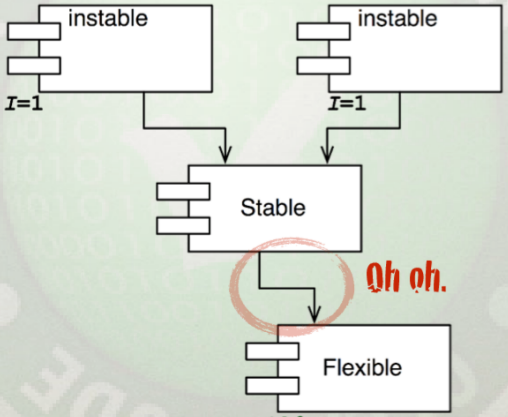
Instability Metric



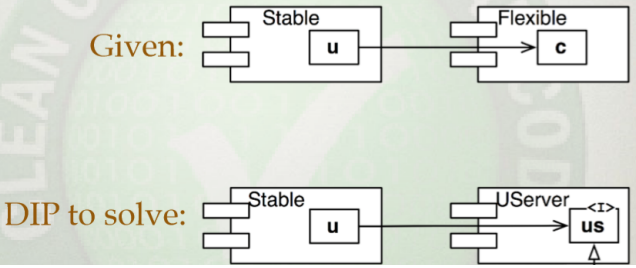
I = FanOut / (FanOut + FanIn)
[0,1]

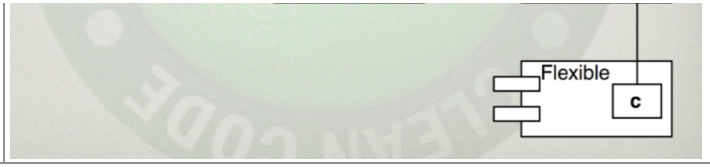
SDP: Depend in the direction of *decreasing* I.

An Example of an SDP Violation



DIP Solution





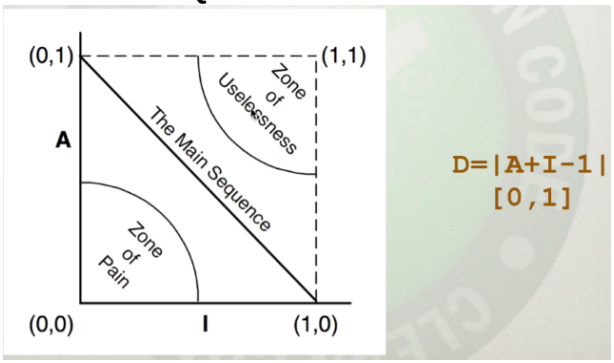
SAP: Stable Abstractions Principle

- We put **high level policy** into stable components. (**DIP**).
- That makes high level policy inflexible.
- Unless we invoke the **OCP**.
- **Abstract components** are open for extension!

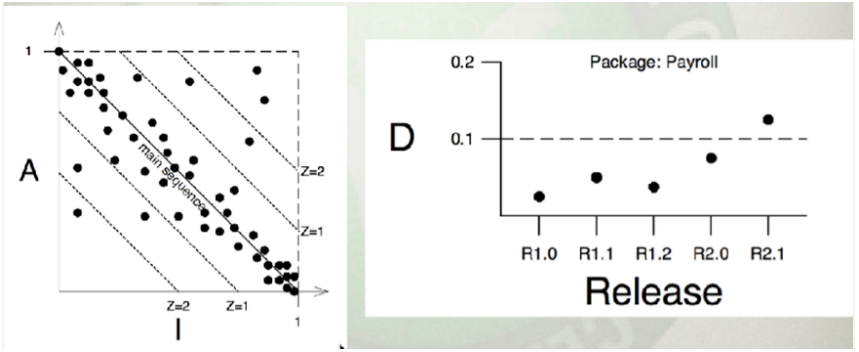
Measuring Abstractness

- Nc: Number of classes in component.
- Na: Number that are abstract.
- A = Na/Nc : [0,1]
- SAP: Depend in the direction of increasing A.

THE MAIN SEQUENCE



USES OF D



Interface-based programming promotes *encapsulation* , or the hiding of information from the client. The less a client knows about the way an object is implemented, the better. The more the details of an implementation are encapsulated, the greater is the likelihood that you can change a method or property without affecting the client code. Interfaces maximize encapsulation because the client interacts with an abstract service definition instead of an actual object. **Encapsulation** is key to successfully applying both **object-oriented** and **component-oriented methodologies.**

Another important concept that originated with object-oriented programming is ***polymorphism***. Two objects are said to be polymorphic with respect to each other when both derive from a common base type (such as an interface) and implement the exact set of operations defined by that base type. If a client is written to use the operations of a base type, the same client code can interact with any object that is polymorphic with that base type. When polymorphism is used properly, changing from one object to another has no effect on the client; it simplifies maintenance of the application to which the client and object belong.

Principles of Component-Oriented Programming from Book "Programming .NET Components, 2nd Edition by Juval Lowy"

Component-oriented programming requires both systems that support the approach and programmers that adhere to its discipline and its core principles. However, it's often hard to tell the difference between a true principle and a mere feature of the component technology being used. As the supporting technologies become more powerful, no doubt software engineering will extend its understanding of what constitutes component-oriented programming and embrace new ideas, and the core principles will continue to evolve. The most important principles of component-oriented programming include:

- **Separation of interface and implementation**
- **Binary compatibility**

- **Language independence**
- **Location transparency**
- **Concurrency management**
- **Version control**
- **Component-based security**

The following subsections discuss these seven important principles. As discussed in the next section, .NET enables complying with all of the core principles, but it does not necessarily enforces these principles.

Separation of Interface from Implementation

The fundamental principle of component-oriented programming is that the basic unit in an application is a binary-compatible interface. The interface provides an abstract service definition between a client and the object. This principle contrasts with the object-oriented view of the world, which places the object, rather than its interface, at the center. An *interface* is a logical grouping of method definitions that acts as the *contract* between the client and the service provider. Each provider is free to provide its own interpretation of the interface—that is, its own implementation. The interface is implemented by a black-box binary component that completely encapsulates its interior. This principle is known as *separation of interface from implementation*.

To use a component, the client needs only to know the interface definition (i.e., the service contract) and to be able to access a binary component that implements that interface. This extra level of indirection between the client and the object allows one implementation of an interface to be replaced by another without affecting the client code. The client doesn't need to be recompiled to use a new version. Sometimes the client doesn't even need to be shut down to do the upgrade. Provided the interface is immutable, objects implementing the interface are free to evolve, and new versions can be introduced smoothly and easily. To implement the functionality promised by an interface inside a component, you use traditional object-oriented methodologies, but the resulting class hierarchies are usually simpler and easier to manage.

Interfaces also facilitate reuse. In object-oriented programming, the basic unit of reuse is the object. In theory, different clients should be able to use the same object. Each reuse instance saves the reusing party the amount of time and effort that would have been spent implementing the object. Reuse initiatives have the potential for significant cost reductions and reduced product-development cycle time. One reason why the industry adopted object-oriented programming so avidly was its desire to reap the benefits of reuse.

In reality, however, objects are rarely reusable. They are often specific to the problems and particular contexts for which they were developed, and unless the objects are "nuts and bolts"—that is, simple and generic—they can't be reused even in very similar contexts. This is also true in many engineering disciplines, including mechanical and electrical engineering. For example, consider the computer mouse you use with your workstation. Each part of this mouse is designed and manufactured specifically for your make and model. For reasons of branding and electronics specifics, parts such as the body case can't be used in the manufacturing of any other type of mouse (even very similar ones), whether made by the same manufacturer or others. However, the interface between mouse and human hand is well defined, and any human (not just yourself) can use the mouse. Similarly, the typical USB interface between mouse and computer is well defined, and your mouse can plug into almost any computer adhering to that interface. The basic units of reuse in the computer mouse are the interfaces with which the mouse complies, not the mouse parts themselves.

In component-oriented programming, the basic unit of reuse is the interface, not a particular component. By separating interfaces from implementation in your application, and using predefined interfaces or defining new interfaces, you enable that application to reuse existing components and enable reuse of your new components in other applications.

Binary Compatibility Between Client and Server

Another core principle of component-oriented programming is *binary compatibility* between client and server. Traditional object-oriented programming requires all the parties involved—clients and servers—to be part of one monolithic application. During compilation, the compiler inserts the addresses of the server entry points into the client code. Component-oriented programming, in contrast, revolves around packaging code into components (i.e., binary building blocks). Changes to the component code are contained in the binary unit hosting the component; you don't need to recompile and redeploy the clients. However, the ability to replace and plug in new binary versions of the server implies binary compatibility between the client and the server, meaning that the client's code must interact at runtime with exactly what it expects as far as the binary layout in memory of the component entry points. This binary compatibility is the basis for the contract between the component and the client. As long as the new version of the component abides by this contract, the client isn't affected. In Chapter 2, you will see how .NET provides binary compatibility.

Language Independence

Unlike in traditional object-oriented programming, in component-oriented programming, the server is developed independently of the client. Because the client interacts with the server only at runtime, the only thing that binds the two is binary compatibility. A corollary is that the programming languages that implement the client and server should not affect their ability to interact at runtime. *Language independence* means exactly that: when you develop and deploy components, your choice of programming language should be irrelevant. Language independence promotes the interchangeability of components, and their adoption and reuse. .NET achieves language independence through an architecture and implementation called the Common Language Runtime (CLR), which is discussed further in Chapter 2.

Location Transparency

A component-based application contains multiple binary components. These components can all exist in the same process, in different processes on the same machine, or on different machines on a network. With the advent of web services, components can also now be distributed across the Internet.

The underlying component technology is required to provide a client with *location transparency* , which allows the client code to be oblivious to the actual locations of the objects it uses. Location transparency means there is nothing in the client's code pertaining to where the objects execute. The same client code must be able to handle all cases of object location (see Figure 1-3), although the client should be able to insist on a specific location as well. Note that in the figure, the object can be in the same process (e.g., Process 1 on Machine A), in different processes on the same machine (e.g., Process 1 and Process 2 on Machine A), on different machines in the same local network (e.g. Machine B), or even on different machines across the Internet (e.g., Machine C).

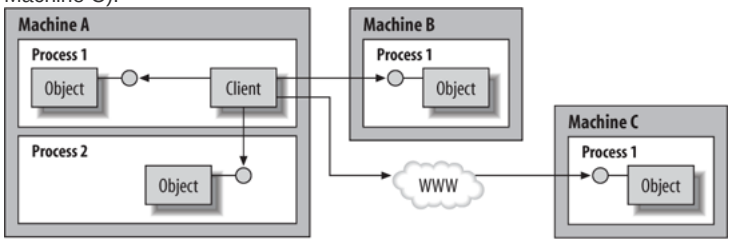


Figure 1-3. With location transparency, the client is oblivious to the actual object location

Location transparency is crucial to component-oriented programming for a number of reasons. First, it lets you develop the client and components locally (which leads to easier and more productive debugging), yet deploy the same code base in distributed scenarios. Second, the choice of whether to use the same process for all components or multiple processes for multiple machines has a significant impact on performance and ease of management versus scalability, availability, robustness, throughput, and security. Organizations have different priorities and preferences for these trade-offs, yet the same set of components from a particular vendor or team should be able to handle all scenarios. Third, the locations of the components tend to change as an application's requirements evolve over time. If the locations are transparent to the client, this movement does not cause problems.

To minimize the cost of long-term maintenance and extensibility, you should avoid having client code make any assumptions regarding the locations of the objects it uses and avoid making explicit calls across processes or across machines. *.NET remoting* is the name of the technology that enables remote calls in .NET. Chapter 10 is dedicated to .NET remoting and discusses .NET support for location transparency.

Concurrency Management

A component developer can't possibly know in advance all the possible ways in which a component will be used, and particularly whether multiple threads will access it concurrently. The safest course is to assume that the component will be used in concurrent situations and to prepare for this eventuality. One option is to provide some mechanism inside the component for synchronizing access. However, this approach has two flaws. First, it may lead to deadlocks; if every component in the application has its own synchronization lock, a deadlock can occur if two components on different threads try to access each other. Second, it's an inefficient use of system resources for all components in the application to be accessed by the same thread.

To get around these problems, the underlying component technology must provide a *concurrency management* service—that is, a way for components to participate in some application-wide synchronization mechanism, even when the components are developed separately. In addition, the underlying component technology should allow components and clients to provide their own synchronization solutions for fine-grained control and optimized performance. .NET concurrency management support is discussed in Chapter 8 as part of developing multithreaded .NET applications.

Versioning Support

Component-oriented programming must allow clients and components to evolve separately. Component developers should be able to deploy new versions (or just fixes) of existing components without affecting existing client applications. Client developers should be able to deploy new versions of the client application and expect them to work with older versions of components. The underlying component technology should support *versioning* , which allows a component to evolve along different paths and allows different versions of the same component to be deployed on the same machine, or even in the same client process, *side by side*. The component technology should also detect incompatibility as soon as possible and alert the client. .NET's solution to version control is discussed in Chapter 5.

Component-Based Security

In component-oriented programming, components are developed separately from the client applications that use them. Component developers have no way of knowing how a client application or end user will try to use their work, so a benign component could well be used maliciously to corrupt data or transfer funds between accounts without proper authorization or authentication. Similarly, a client application has no way to know whether it's interacting with a malicious component that will abuse the credentials the client provides. In addition, even if both the client and the component have no ill intent, the end application user can still try to hack into the system or do some other damage (even by mistake).

To lessen the danger, a component technology must provide a security infrastructure to deal with these scenarios, without coupling components and client applications to each other. Security requirements, policies, and events (such as new users) are among the most volatile aspects of the application lifecycle, and security policies vary between applications and customers. A productive component technology should allow for the components to have as few security policies as possible and for there to be as little security awareness as possible in the code itself. It should also allow system administrators to customize and manage the application security policy without requiring changes to the code. .NET's rich security infrastructure is the subject of chapter 12.

Version Control and DLL Hell

Historically, the versioning problem has been the source of much aggravation. Early attempts at component technology using DLL and DLL-exported functions created the predicament known as *DLL Hell*. A typical DLL Hell scenario involves two client applications, say A1.0 and B1.0, each using version C1.0 of a component in the *mydll.dll* file. Both A1.0 and B1.0 install a copy of *mydll.dll* in some global location, such as the *System* directory. When version A1.1 is installed, it also installs version C1.1 of the component, providing new functionality in addition to the functionality defined in C1.0. Note that *mydll.dll* can contain C1.1 and still serve both old and new client application versions, because the old clients aren't aware of the new functionality, and the old functionality is still supported. Binary compatibility is maintained via strict management of ordinal numbers for the exported functions (a source for another set of problems associated with DLL Hell). The problem starts when Application B1.0 is reinstalled. As part of installing B1.0, version C1.0 is reinstalled, overriding C1.1. As a result, A1.1 can't execute.

Interestingly enough, addressing the issue of DLL Hell was one of the driving forces behind COM. Even though COM makes wide use of objects in DLLs, COM can completely eliminate DLL Hell. However, COM is difficult to learn and apply and consequently can be misused or abused, resulting in problems similar to DLL Hell.

Like COM, .NET was designed with DLL Hell in mind. .NET doesn't eliminate all chances of DLL Hell, but does reduce its likelihood substantially. While it's true that the default versioning and deployment policies of .NET don't allow for DLL Hell, .NET is, after all, an extensible platform. You can choose to override its default behavior to meet some advanced need or to provide your own custom version control policy, but in doing so you risk creating DLL Hell.