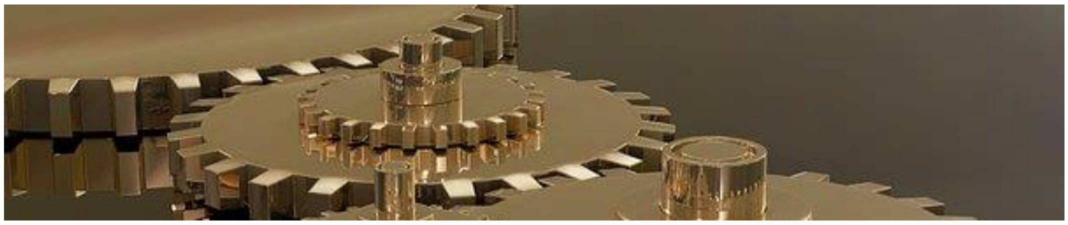


Reduce Technical Debt Through Better Software Package Design

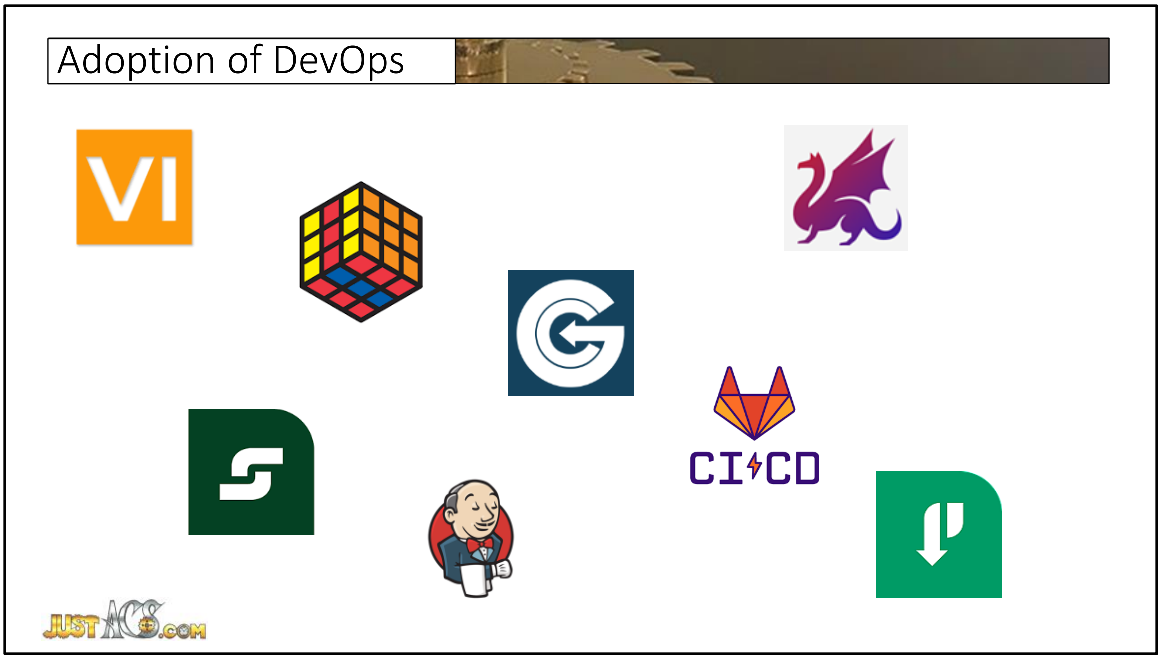
Allen C Smith, CLA
Consulting Software Architect
justACS

Jon McBee, CLA
President
Composed Systems

Thank you for attending my session, and it's great to see you all here at GDevCon.

I'm Allen C Smith, etc.

Jon McBee and I first gave this presentation in 2017 at NIWeek. So why am I returning to the topic now?


Adoption of DevOps

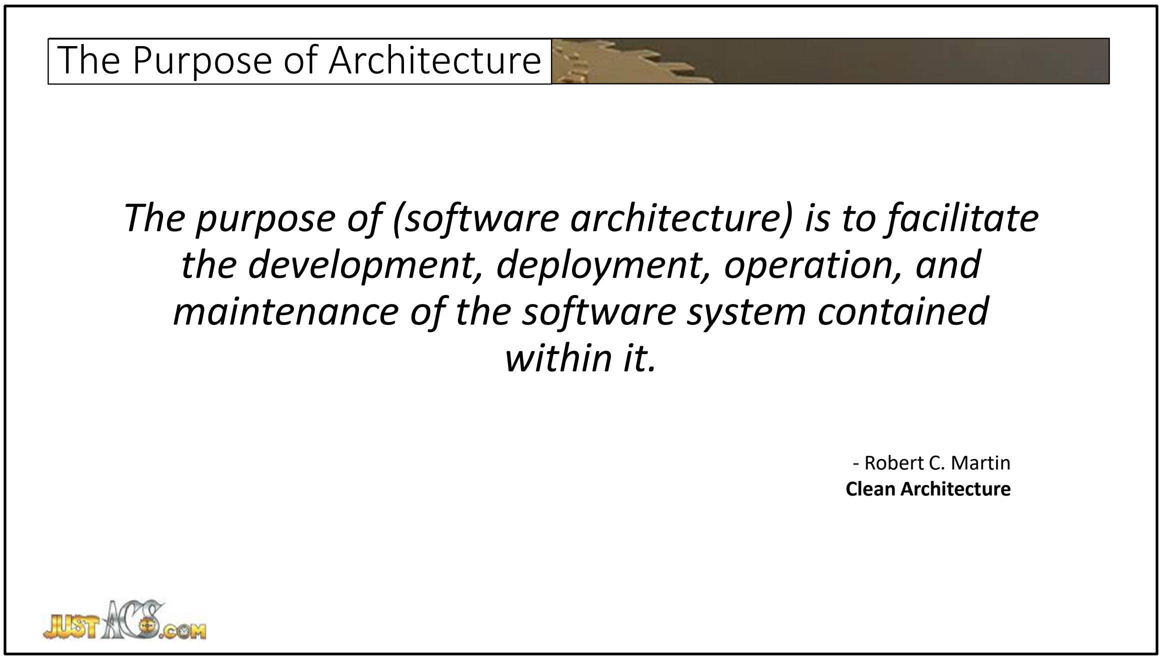
CI/CD
JUST ACS.COM

The advanced LabVIEW community had been talking about DevOps for a few years hen Jon and I first gave this presentation.

[Build]: VI Package Manager was by far the most commonly used DevOps tool in our space.

[Build]: NI Package Manager was still fairly new.

[Build]: The space has only grown since then. GCentral as an effort to standardize code reuse/release. Deployment tools like SystemLink, the MGI Solution Explorer, and JKI's Dragon project. Integration with CI/CD tools like Jenkins and GitLab.



The Purpose of Architecture

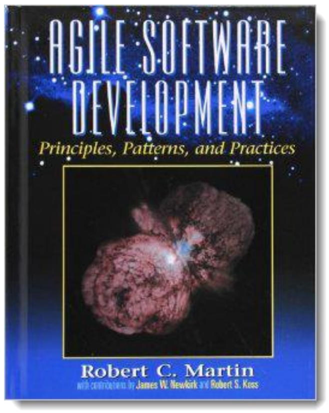
The purpose of (software architecture) is to facilitate the development, deployment, operation, and maintenance of the software system contained within it.

- Robert C. Martin
Clean Architecture

But these are just tools. They are means to an end: the efficient delivery of software.

That is, of course, the purpose of our software architecture. If we want to get good use from our tools, we need to pay close attention to the design of the software we deliver.

Package Management

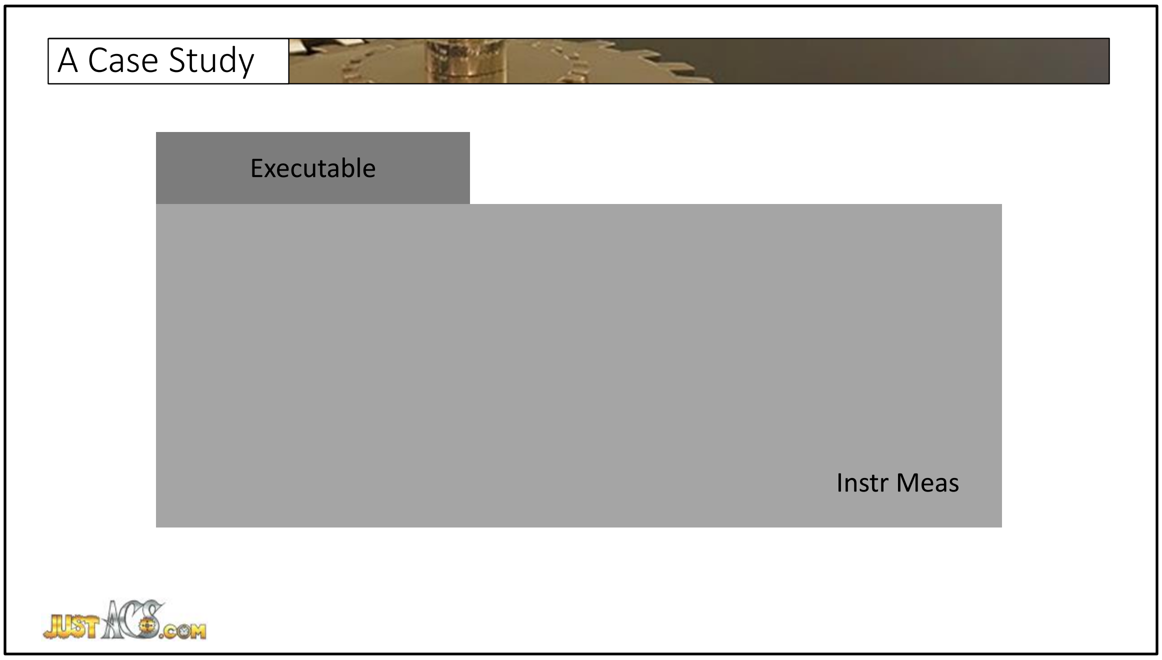

AGILE SOFTWARE DEVELOPMENT
Principles, Patterns, and Practices
Robert C. Martin
with contributions by James W. Newkirk and Robert S. Koss

ACS

Sustainable development practices are the subject of Robert C. Martin's book, Agile Software Development. This is an important book, and I strongly recommend that you add it to your library.

One section of "Uncle Bob's Book" talks about packaging and distributing large software applications, and that is the subject of this presentation.

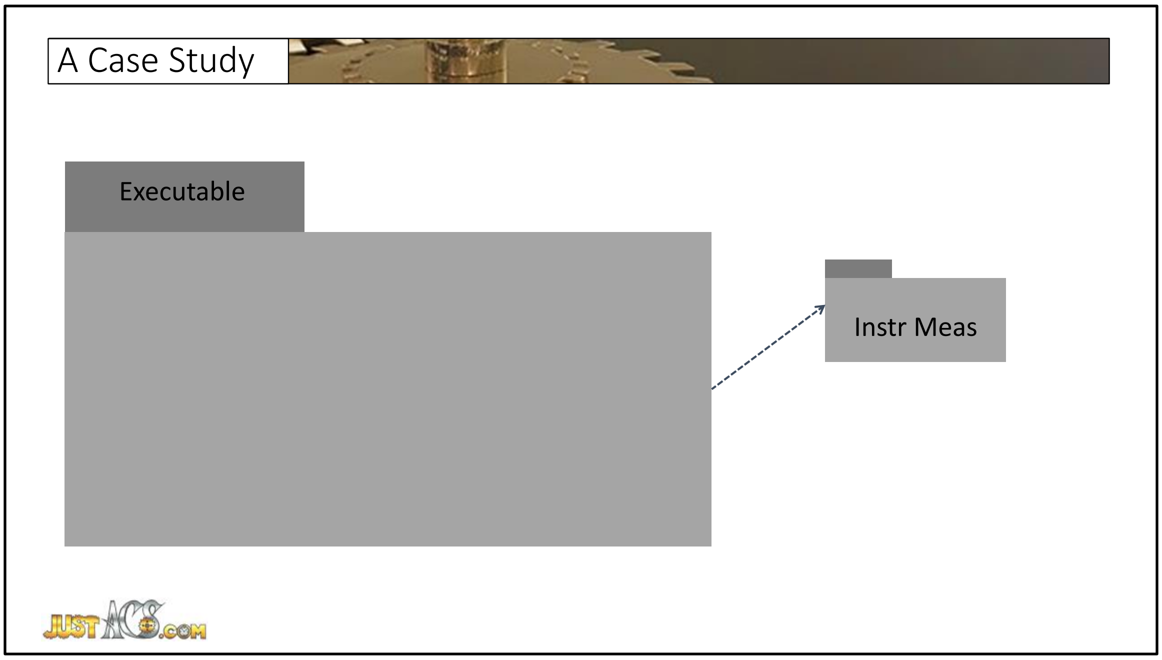
Proper package management plays a key role in the smooth deployment and maintenance of software, especially in larger systems.


A Case Study

ACS

Let’s start with a hypothetical example based on design work I’ve done with customers.

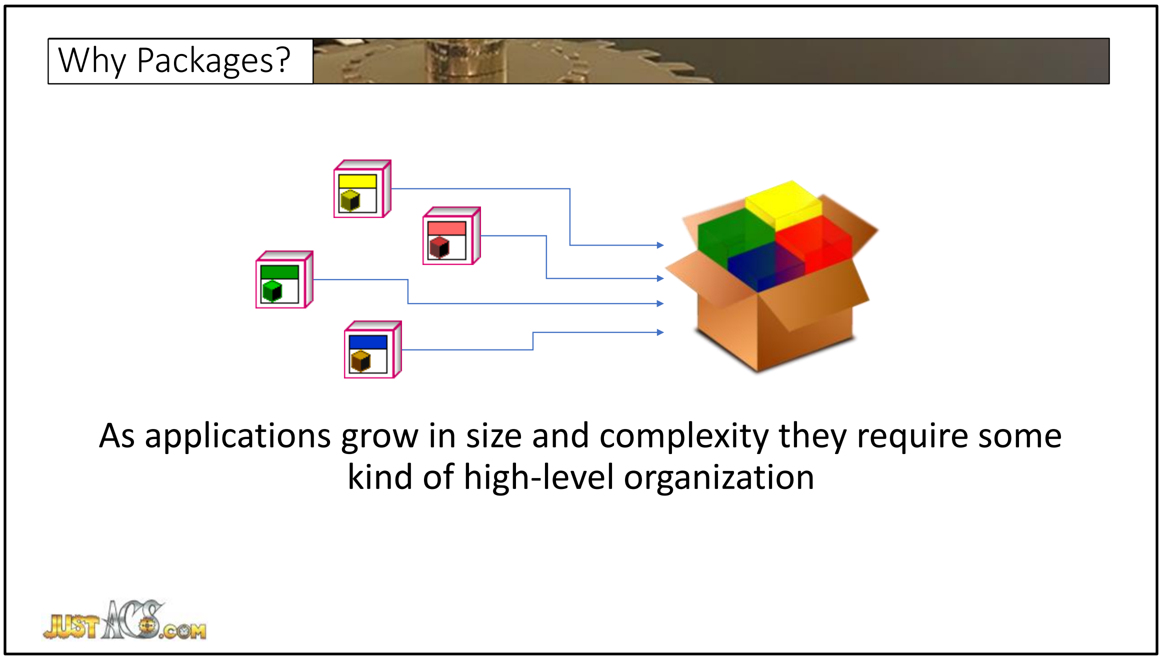
The customer has a legacy system , built up over years. They decide to extend the system by adding a large instrumentation package to handle new kinds of measurements. They find, to their dismay, that build time increases from 45 minutes to 90 minutes, and that’ something of a problem for them.


A Case Study

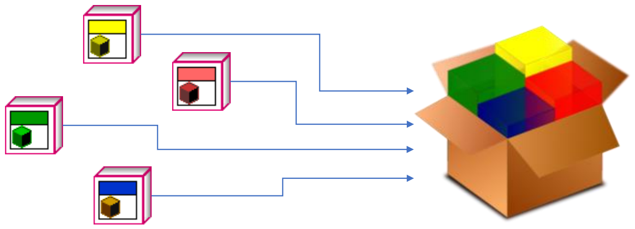
ACS

They opt to migrate the system to a plug-in architecture, with the instrumentation library as the first plug-in. This will allow them to build and revision the executable and the instrumentation package separately. While the total build time is still about 90 minutes, the hope is that there will be less churn in the plug-in, so it will only need to be built intermittently, and then only by the team responsible for the plug-in.

This customer has taken the first steps toward moving from a monolithic application, which must be built and tested as a unit, to a set of software packages, each with their own build cycle.



Why Packages?



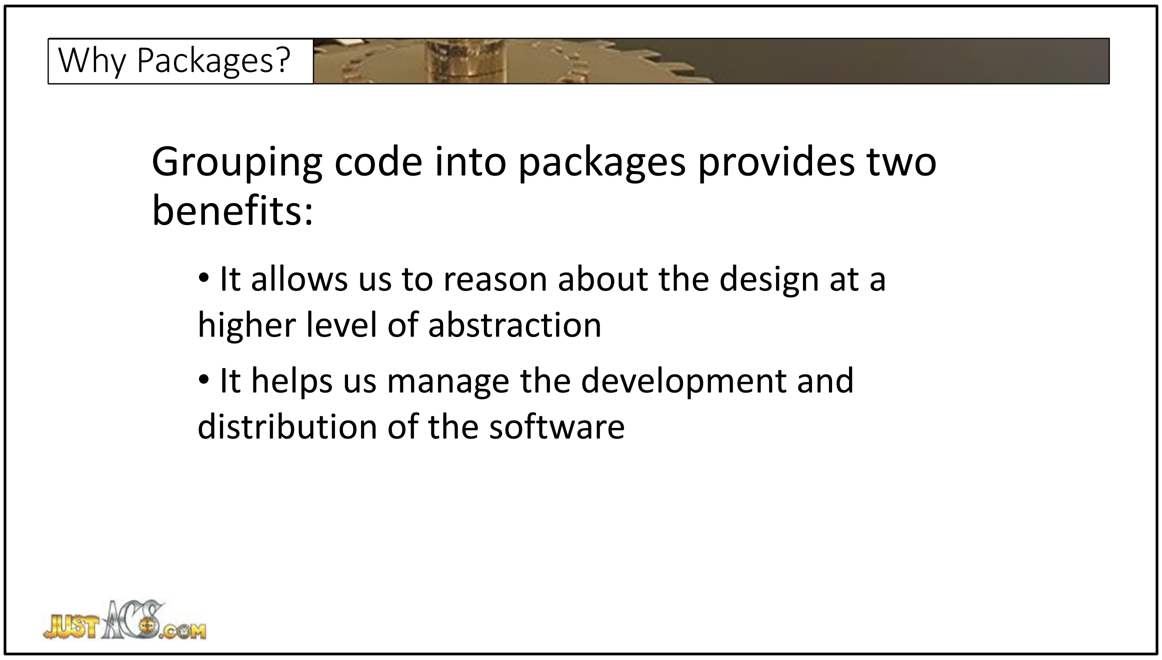
As applications grow in size and complexity they require some kind of high-level organization

JUST ACS.COM

ACS

Classes are great for organizing code at the level where functionality is implemented. But when the application gets to a certain size, it becomes convenient or even necessary to think about organizing the code for reuse, maintenance, and delivery.

The basic unit of work at this level is the package, which is a group of classes and related code.

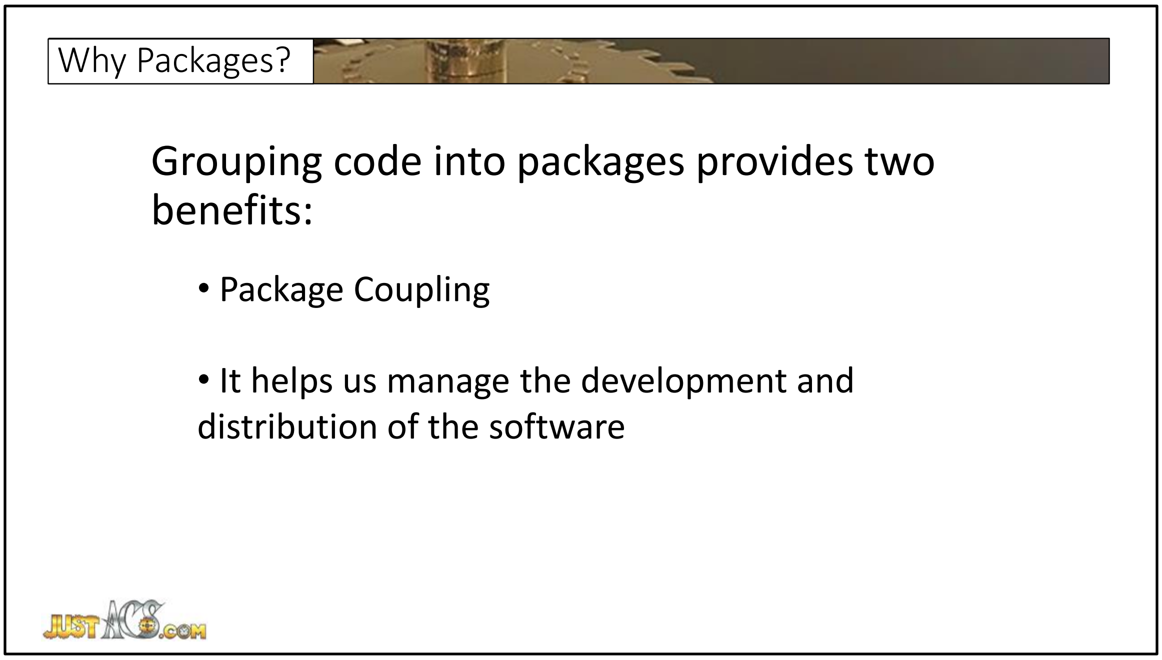


Why Packages?

Grouping code into packages provides two benefits:

- It allows us to reason about the design at a higher level of abstraction
- It helps us manage the development and distribution of the software

ACS

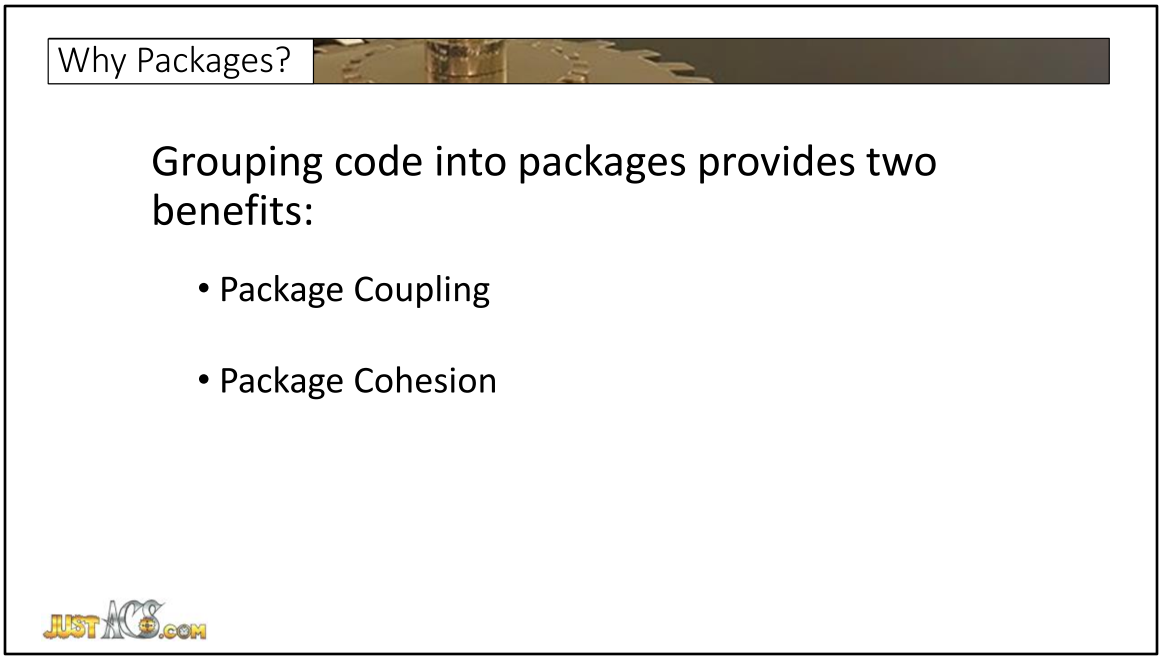


Why Packages?

Grouping code into packages provides two benefits:

- Package Coupling
- It helps us manage the development and distribution of the software

ACS

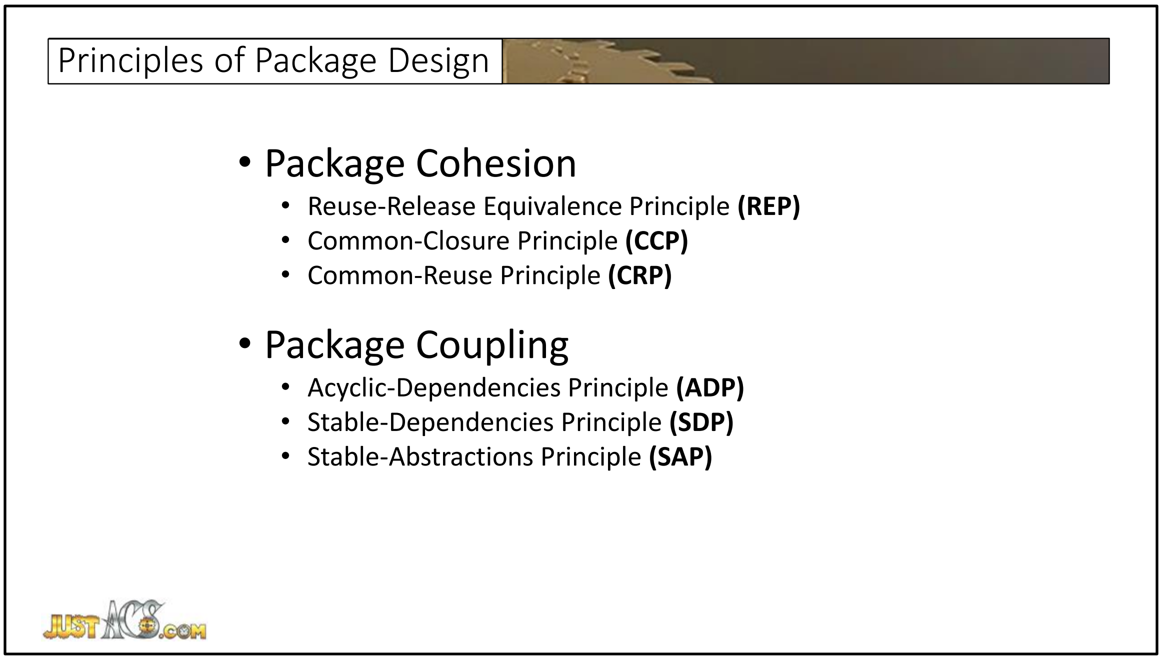


Why Packages?

Grouping code into packages provides two benefits:

- Package Coupling
- Package Cohesion

ACS

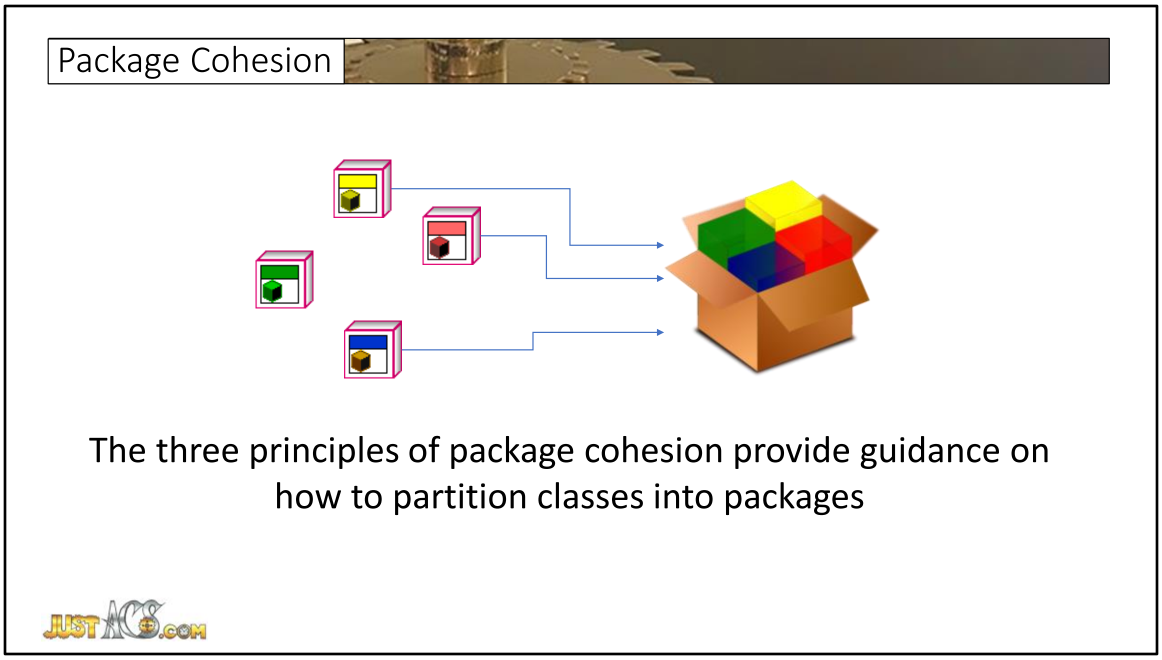


Principles of Package Design

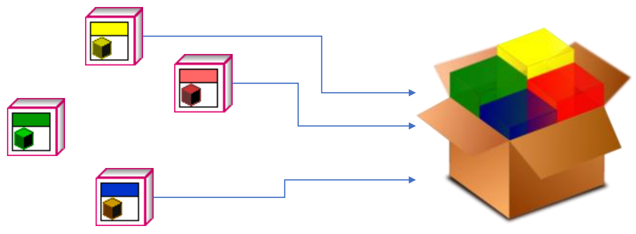
- Package Cohesion
  - Reuse-Release Equivalence Principle **(REP)**
  - Common-Closure Principle **(CCP)**
  - Common-Reuse Principle **(CRP)**
- Package Coupling
  - Acyclic-Dependencies Principle **(ADP)**
  - Stable-Dependencies Principle **(SDP)**
  - Stable-Abstractions Principle **(SAP)**

ACS

If we are to achieve these objectives, we need a set of criteria to help us determine how to allocate our classes to packages (package cohesion) and how our packages should be interrelated (package coupling).

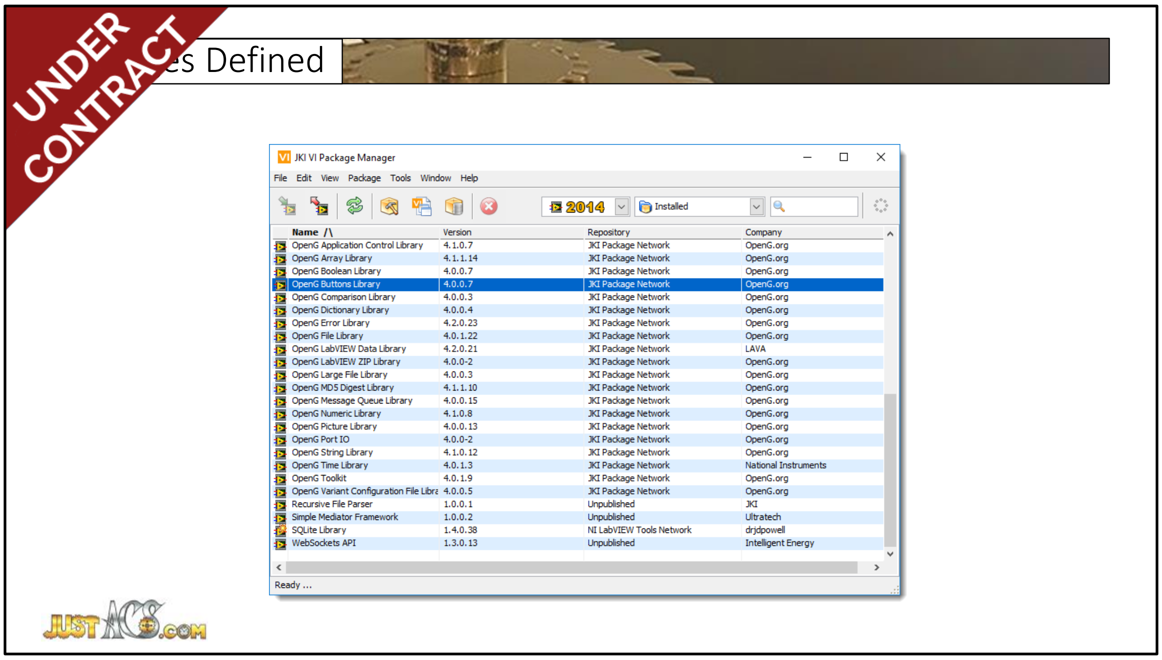


Package Cohesion



The three principles of package cohesion provide guidance on how to partition classes into packages

ACS



UNDER CONTRACT

es Defined

JKI VI Package Manager

Name /\	Version	Repository	Company
OpenG Application Control Library	4.1.0.7	JKI Package Network	OpenG.org
OpenG Array Library	4.1.1.14	JKI Package Network	OpenG.org
OpenG Boolean Library	4.0.0.7	JKI Package Network	OpenG.org
OpenG Buttons Library	4.0.0.7	JKI Package Network	OpenG.org
OpenG Comparison Library	4.0.0.3	JKI Package Network	OpenG.org
OpenG Dictionary Library	4.0.0.4	JKI Package Network	OpenG.org
OpenG Error Library	4.2.0.23	JKI Package Network	OpenG.org
OpenG File Library	4.0.1.22	JKI Package Network	OpenG.org
OpenG LabVIEW Data Library	4.2.0.21	JKI Package Network	LAVA
OpenG LabVIEW ZIP Library	4.0.0-2	JKI Package Network	OpenG.org
OpenG Large File Library	4.0.0.3	JKI Package Network	OpenG.org
OpenG MD5 Digest Library	4.1.1.10	JKI Package Network	OpenG.org
OpenG Message Queue Library	4.0.0.15	JKI Package Network	OpenG.org
OpenG Numeric Library	4.1.0.8	JKI Package Network	OpenG.org
OpenG Picture Library	4.0.0.13	JKI Package Network	OpenG.org
OpenG Port IO	4.0.0-2	JKI Package Network	OpenG.org
OpenG String Library	4.1.0.12	JKI Package Network	OpenG.org
OpenG Time Library	4.0.1.3	JKI Package Network	National Instruments
OpenG Toolkit	4.0.1.9	JKI Package Network	OpenG.org
OpenG Variant Configuration File Libra	4.0.0.5	JKI Package Network	OpenG.org
Recursive File Parser	1.0.0.1	Unpublished	JKI
Simple Mediator Framework	1.0.0.2	Unpublished	Ultratech
SQLite Library	1.4.0.38	NI LabVIEW Tools Network	drjdpowell
WebSockets API	1.3.0.13	Unpublished	Intelligent Energy

Ready ...

JUSTACS.COM

ACS

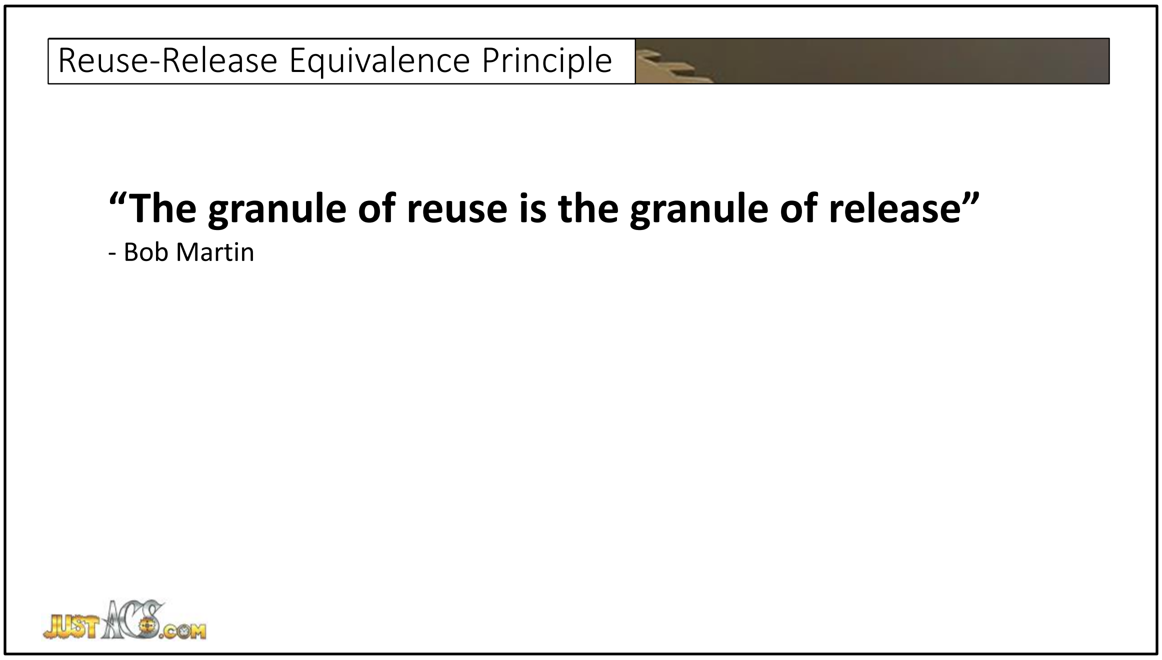
Package design must be considered from the viewpoint of the intended end user. For a package to be useful, the end user must be able to make a few key assumptions about it: that it will be maintained by the developer, that the developer will notify the user when the package changes, and that the end user can decide to adopt the new version or not.

Practically, this means that you, the package developer, must commit to versioning your packages, tracking changes to them, and allowing for the use of some number of older versions.

[BUILD]: You are offering a contract to your users to maintain and version your code, and to notify them of changes.

This is where package design starts: how much clerical and support effort are you willing to provide so that others will reuse your code?

This leads to the first principle of package cohesion.



Reuse-Release Equivalence Principle

“The granule of reuse is the granule of release”
- Bob Martin

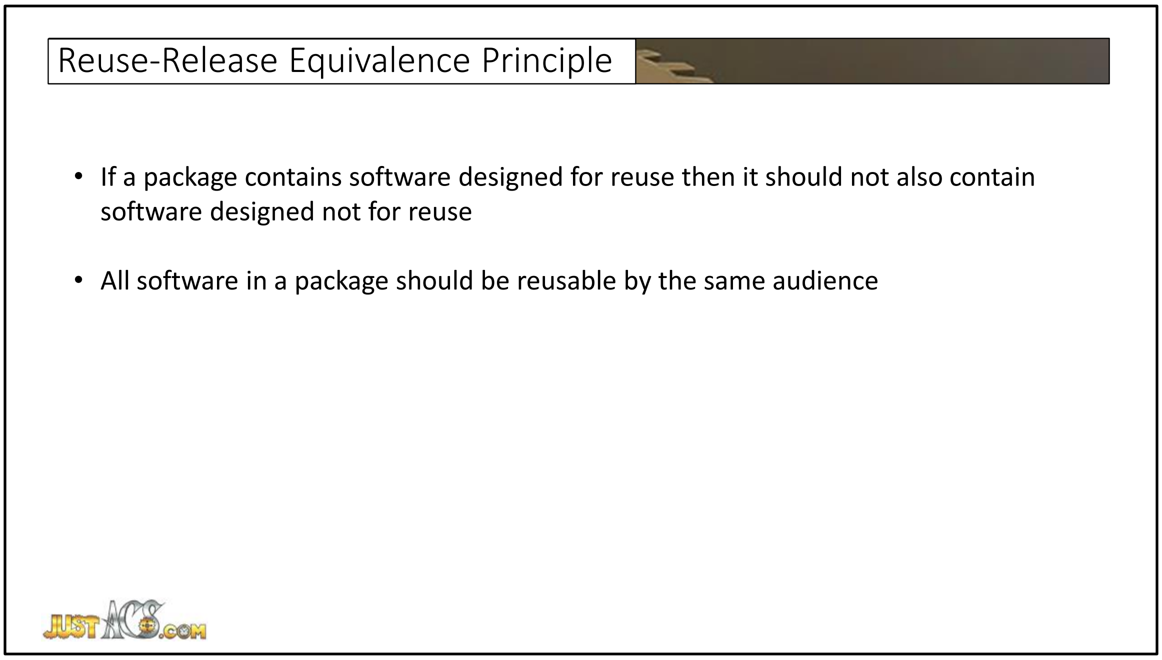
ACS

Read this as: “the granule of reuse *can be no smaller than* the granule of release.”

The Reuse-Release Equivalence Principle (REP) - The REP tells us that classes should be packaged together because they are used together. It urges us to group things together for user convenience.

This seems obvious, but many package structures are instead based around ideas like "functional areas," "architectural layers," or "originating team.“ As the later principles will show us, however, organizing around anything other than the end user is a pretty bad idea.

An entire system shoehorned into a single package/library would comply with this principle. But if the rate of change in the library was not extremely low, it would suffer from the problems addressed by the remaining principles.



Reuse-Release Equivalence Principle

- If a package contains software designed for reuse then it should not also contain software designed not for reuse
- All software in a package should be reusable by the same audience

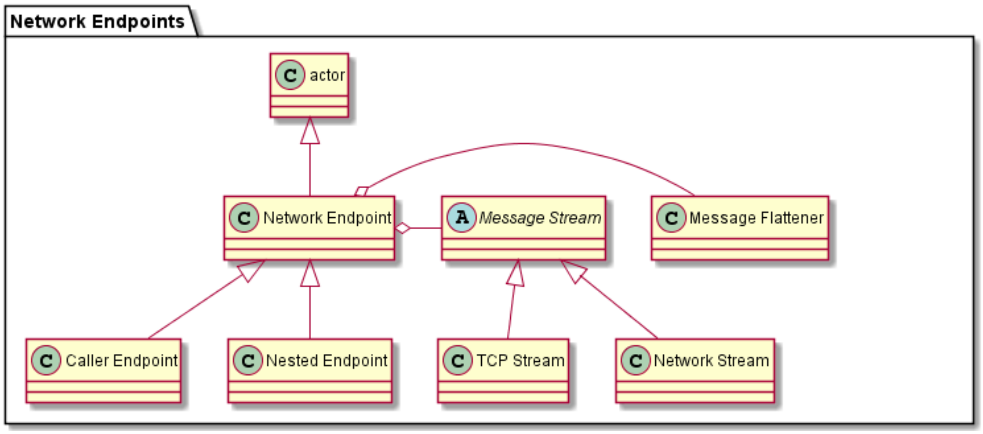
ACS

These two corollaries remind us to focus on the *end user* of the package, not the functional design of the software or the team that is writing it.

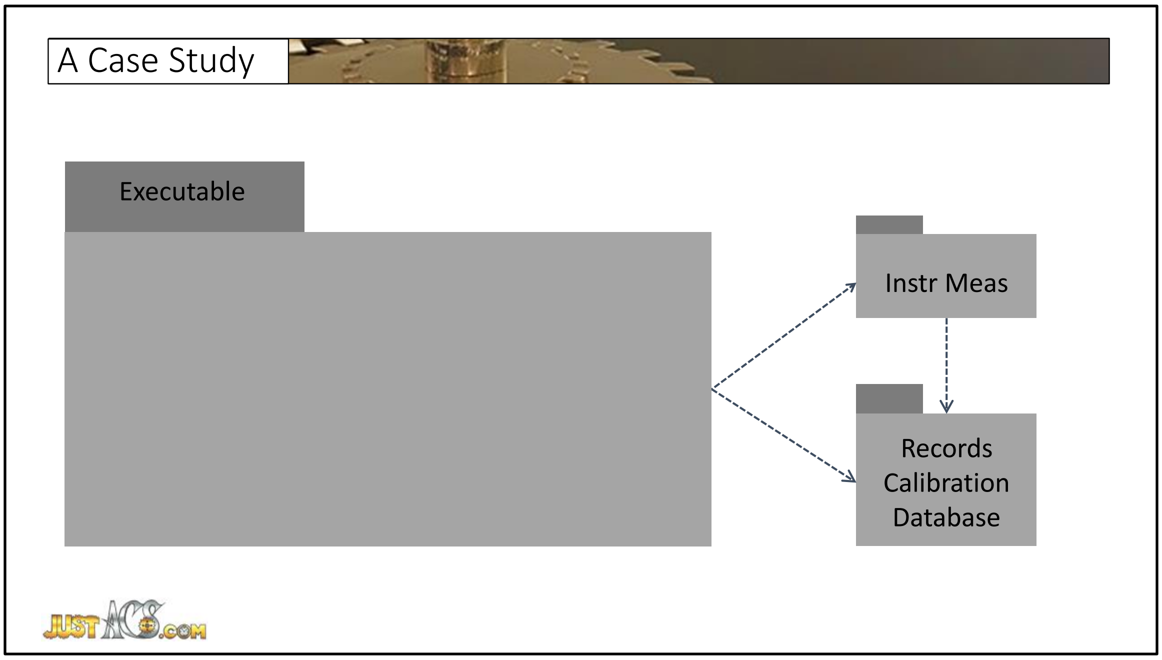
From the user’s perspective, a reuse package should *only* contain reuse code. And all of the code in that package should be of interest to him, i.e. relevant to the work he is doing.

REP tells us what should be included in a package. There are also rules for what should be left out.

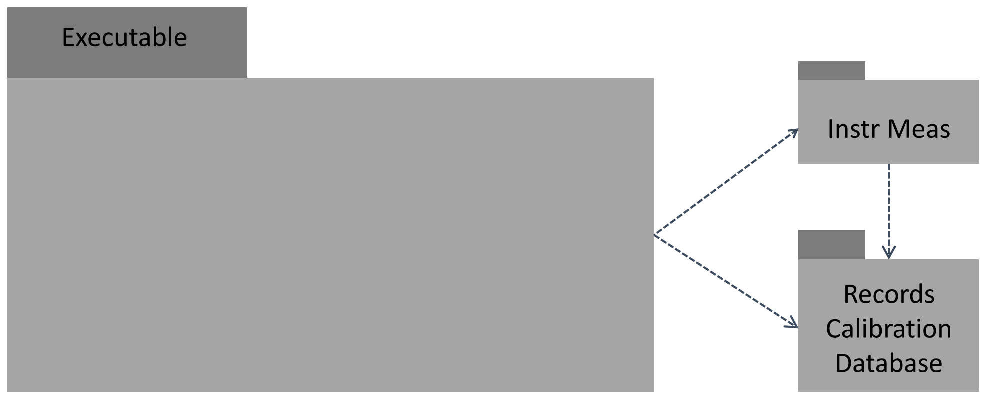
Example of A Package



Here is an example of an actual package. You can download Network Endpoints on the Tools Network. The current version is 3.0; 4.0 is pending. All of the code in the package is intended for reuse by a single target audience – developers who wish to connect actor trees running on separate targets. Code related to this package but NOT intended for reuse, such as the set of unit tests for the package, has been excluded.



A Case Study

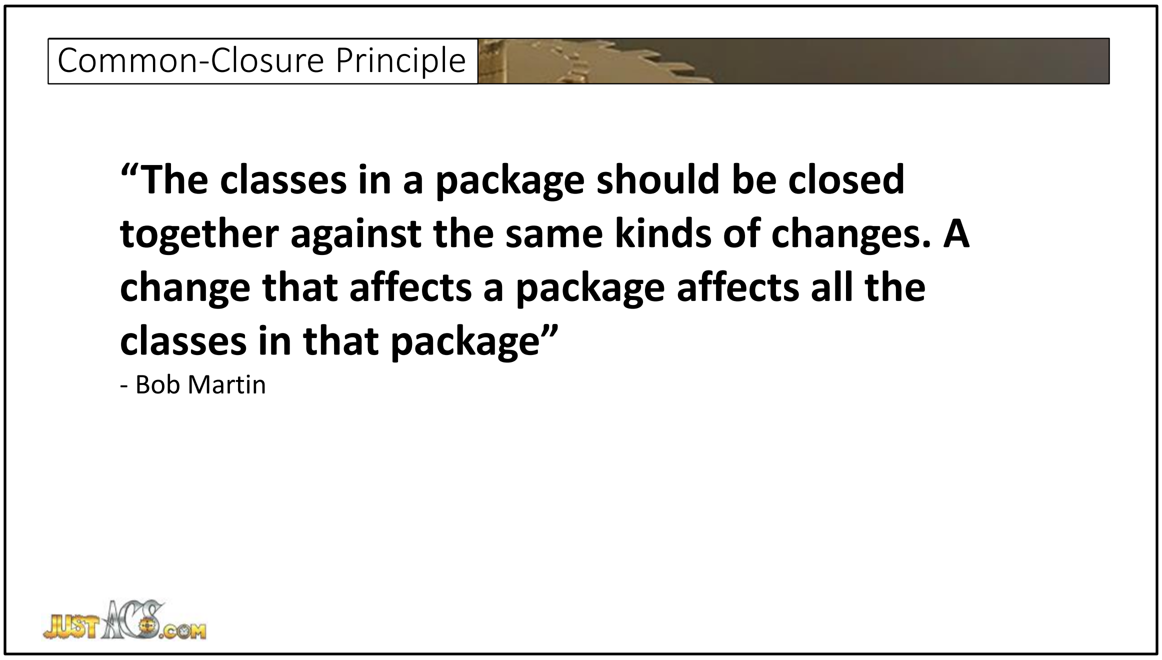


JON

Let’s go back to our case study. When we pulled the instrumentation package out of the executable, we found that we had to pull out several other classes as well, because a LabVIEW plug-in can’t depend on code inside an executable. Clearly, an additional package was called for.

However, looking at the full set of classes, it became clear that putting them in a single package would not be the best idea, from a maintenance perspective. We would like to be able to add a new record type and release the records module without having to also release new versions of the calibration and database code.

We had discovered the next principle of package design.



Common-Closure Principle

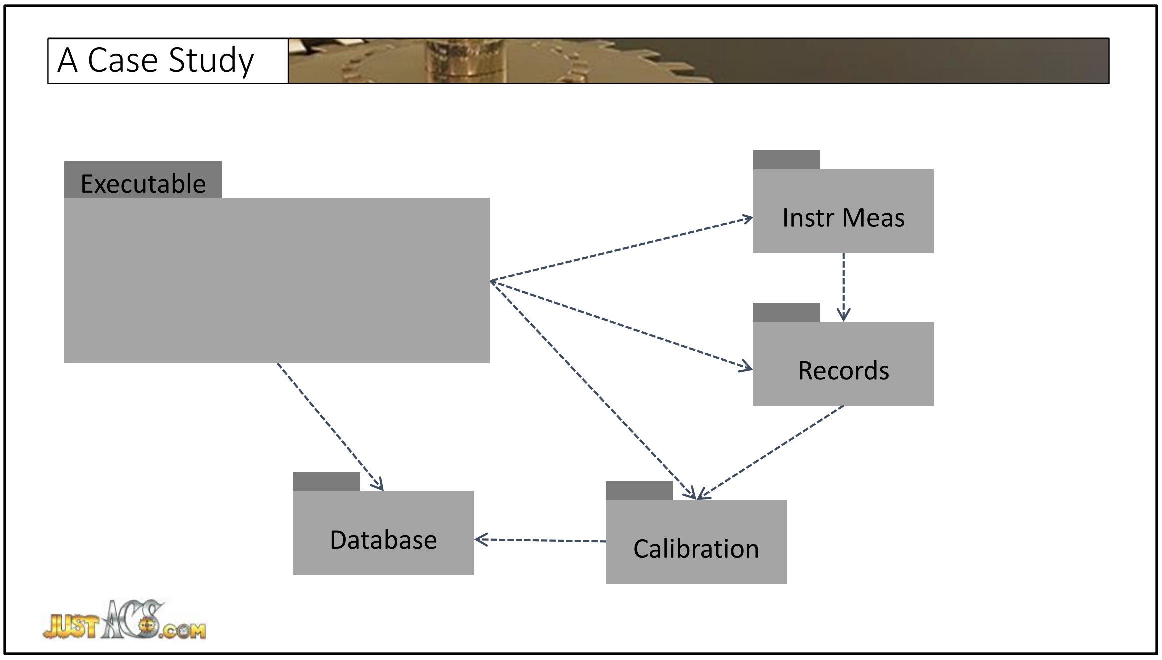
"The classes in a package should be closed together against the same kinds of changes. A change that affects a package affects all the classes in that package"
- Bob Martin

JUSTACS.COM

JON

The Common-Closure Principle (CCP) - The CCP is an application of the Open-Closed Principle at the next level up. This principle suggests that you should group your classes around the impact of change. A change should optimally impact only one package; as many other packages as possible should be closed to that change. CCP recommends grouping classes around the way the code is maintained, rather than the way it is used. Modules with a high rate of change might need to be grouped by closure rather than by use. Highly stable packages might be better conglomerated (REP).

The CCP says to group classes together that are likely to change for the same reason. Two classes that always change together should be in the same package, to minimize the work load of releasing and distributing the change.

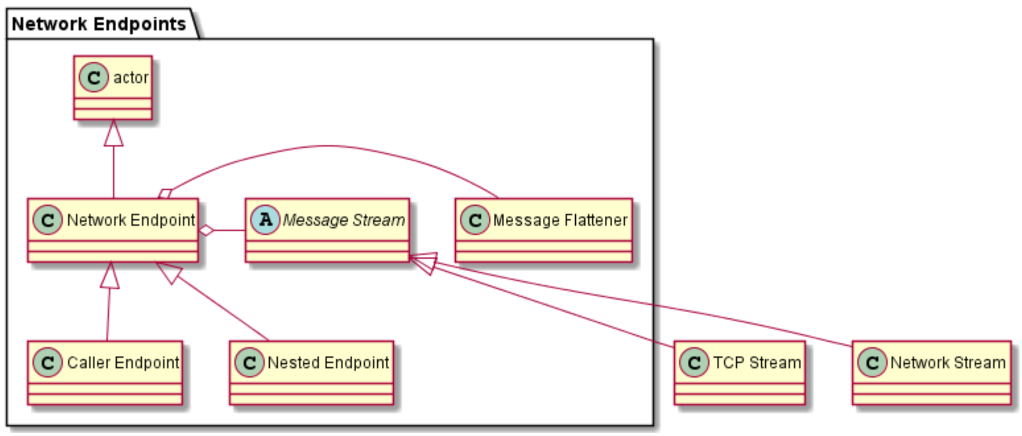


A Case Study

JON

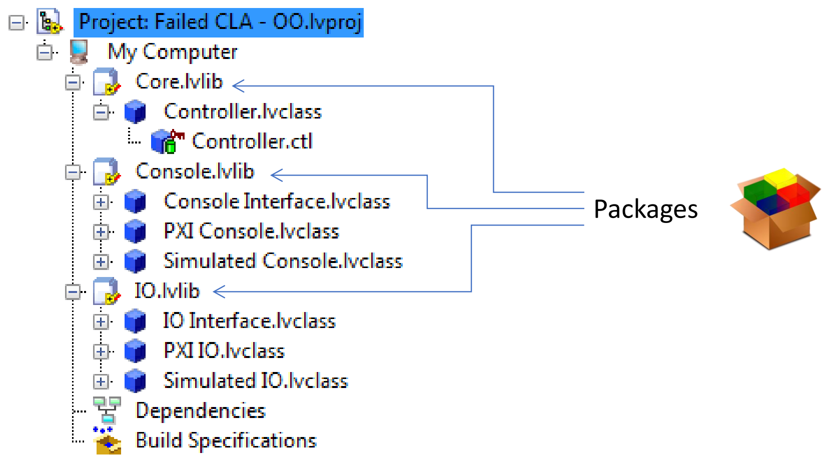
Going back to our example, we have grouped classes together that are likely to change for the same reason. When the inevitable change order comes, the work will hopefully be confined to one or two packages. This is the essence of modular software.

Example of A Package



The specific stream implementations – TCP Stream and Network Stream – would have different reasons to change than the code that adapts these classes to actor space. The Common Closure Principle suggests that we consider moving them out of the package, though I ultimately chose not to do so.

Another Example


Project: Failed CLA - OO.lvproj
My Computer
Core.lvlib
Controller.lvclass
Controller.ctl
Console.lvlib
Console Interface.lvclass
PXI Console.lvclass
Simulated Console.lvclass
IO.lvlib
IO Interface.lvclass
PXI IO.lvclass
Simulated IO.lvclass
Dependencies
Build Specifications

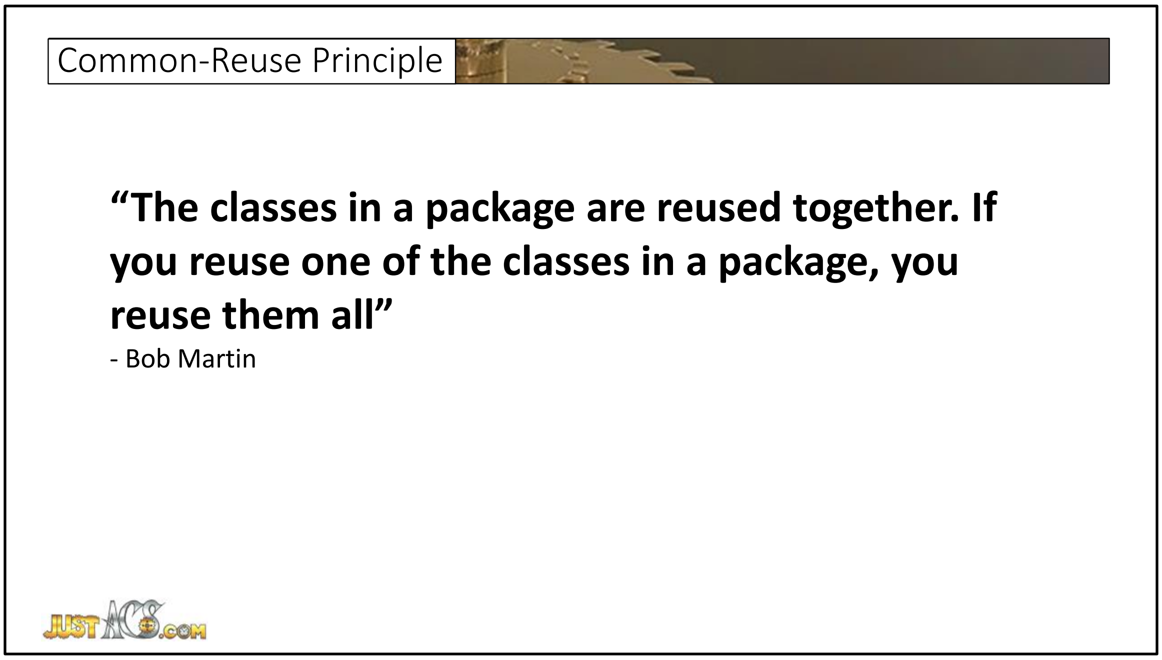
JON

Let’s take a look at a common failing in CLA exam submissions. The project shown is a demonstration, not part of an actual student submission.

The classes in this project have been grouped into three packages: Core, Console, and IO. Core represents the central state handler for the system, Console represents UI elements, and IO represents hardware. Console and IO are intended to be plug-ins; the user can select between simulated IO or PXI instrument drivers, for example.

One can argue that this package design satisfies the CCP, because the classes of each package subject to similar kinds of change. If the message traffic between the user and the core code changes, for example, that change could affect all available UIs. But this is *not* how you would want to deliver code to the customer.

The problem is one of delivery: it is unlikely that the end user of a deployed system would want the simulator libraries. In the same vein, the software test team would likely prefer to test the core code without having to install hardware. This leads us to our next design principle.



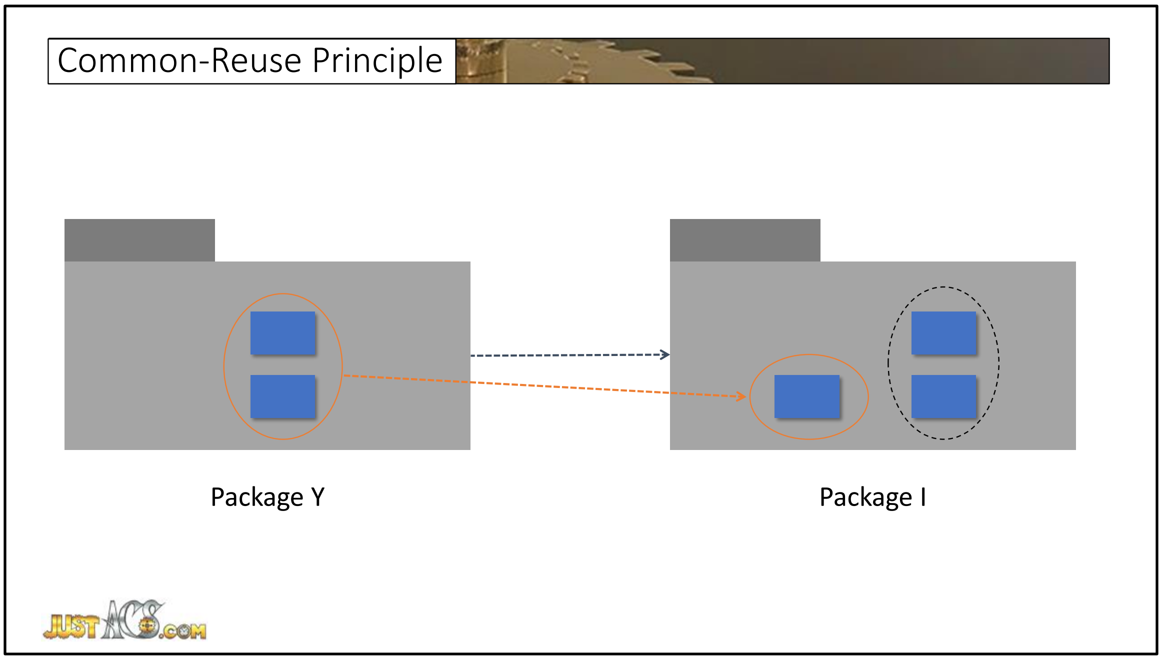
Common-Reuse Principle

“The classes in a package are reused together. If you reuse one of the classes in a package, you reuse them all”
- Bob Martin

JUST ACS.COM

JON

The Common-Reuse Principle (CRP) - This almost seems like a restatement of the REP, but the emphasis here is on *limiting* the impact to consumers. Imagine an API package with two sets of reusable class clusters. A programmer might choose to consume only one reusable component, but changes to the other component necessitate redistribution of the entire package. An unwanted release unnecessarily burdens the consuming programmer, who must re-integrate and re-test the entire library in order to stay current. Where the REP leads us to conglomerate, the CRP leads us to split packages apart. This tension between the principles leads us to find the level of granularity that provides greatest convenience (least negative impact) to users.
In spirit, the CRP is a package-level restatement of the Interface Segregation Principle, which says to keep interfaces small and focused for similar reasons.



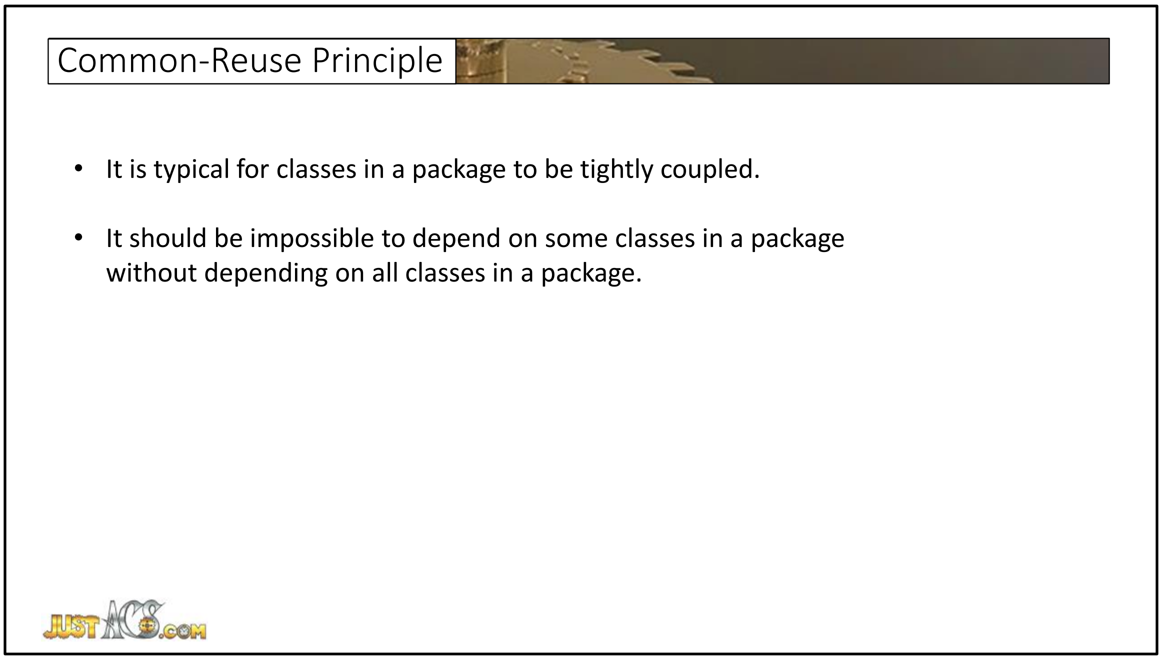
Common-Reuse Principle

JON

When a user is only interested in a few classes of a package

the user code still depends on all dependencies of the package

the user code must be recompiled/relinked and retested after a new release of the package, even if the actually used classes didn't change

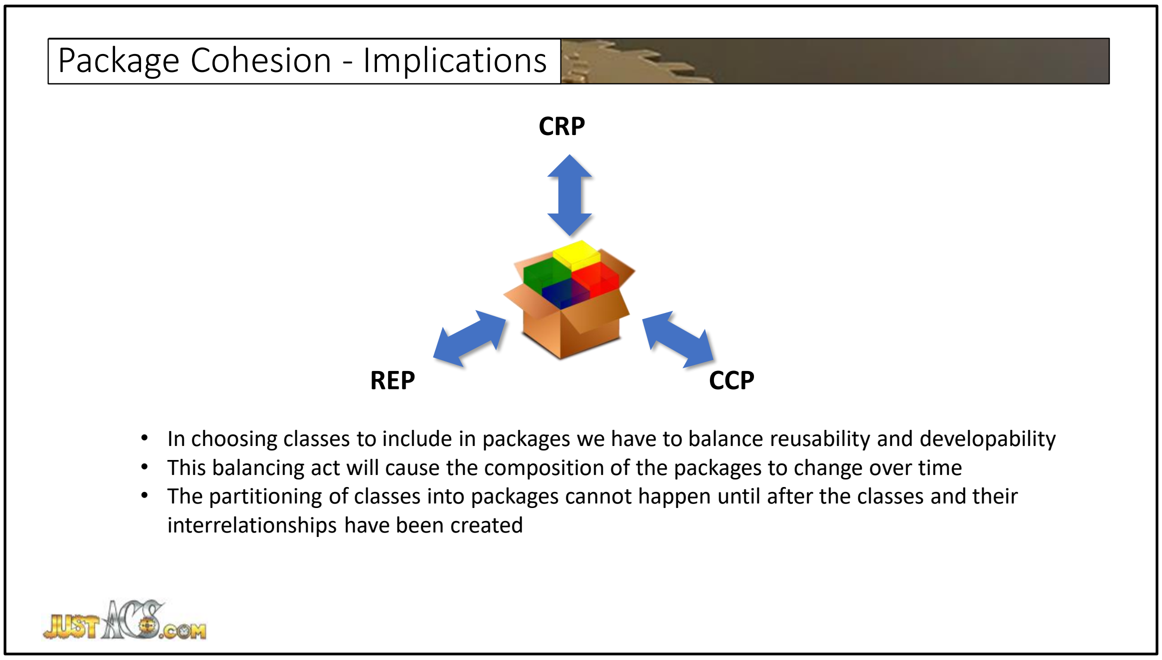


Common-Reuse Principle

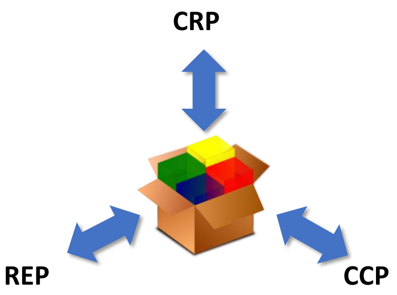
- It is typical for classes in a package to be tightly coupled.
- It should be impossible to depend on some classes in a package without depending on all classes in a package.

JON

When a user is only interested in a few classes of a package
    the user code still depends on all dependencies of the package
    the user code must be recompiled/relinked and retested after a new release of the package, even if the actually used classes didn't change



Package Cohesion - Implications



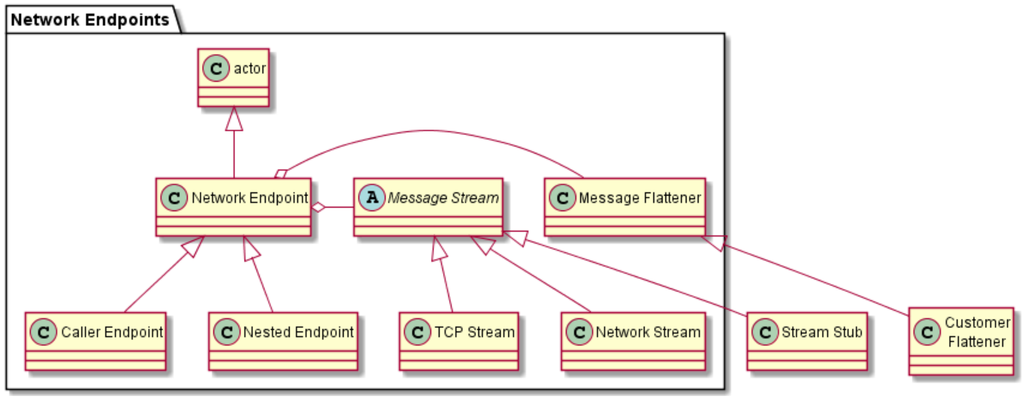
- In choosing classes to include in packages we have to balance reusability and developability
- This balancing act will cause the composition of the packages to change over time
- The partitioning of classes into packages cannot happen until after the classes and their interrelationships have been created

JON

The principles of package cohesion are not absolute. Sometimes a packaging may satisfy all three principles at once, but often the principles represent competing principles. The development team will have to make trade-offs in order to find a balance that works. In general, smaller (and even smaller) packages provide the best chance for adherence to the CCP and CRP principles, although the REP says there is a limit to how small you will want to go.

Following these principles will require occasional re-packaging, which is upsetting to many users. However, correcting less-than-optimal packaging is a single deep cut that can halt the "death of a thousand paper cuts" caused when changes ripple across packages, or when users of a package have to deal with frequent irrelevant updates.

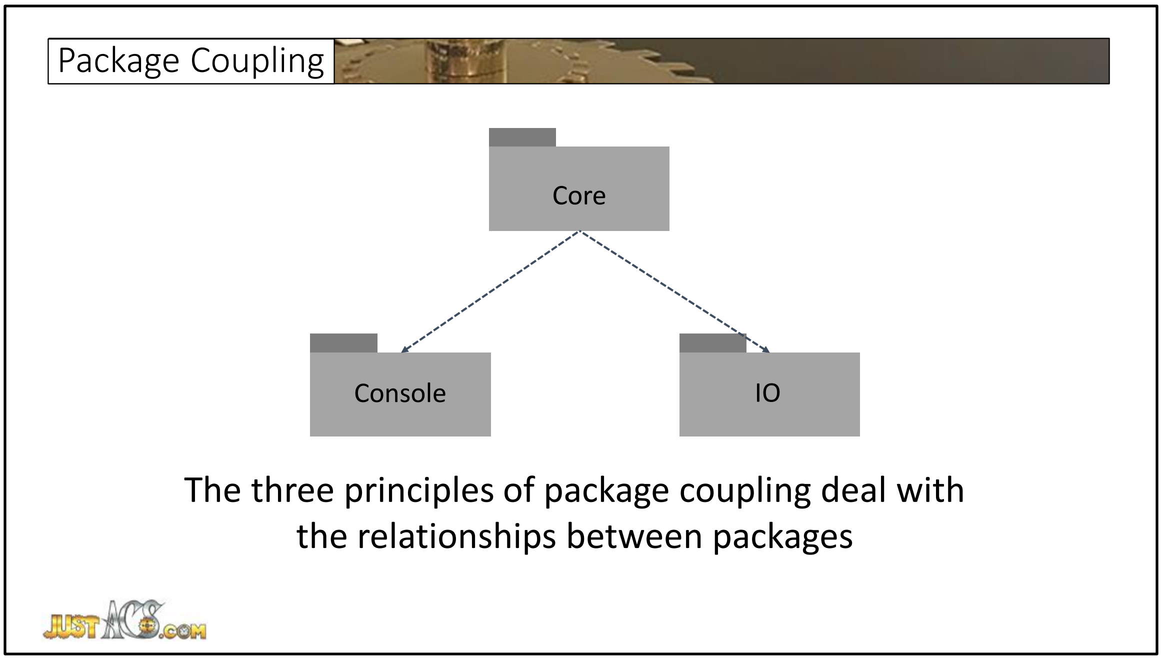
Example of A Package



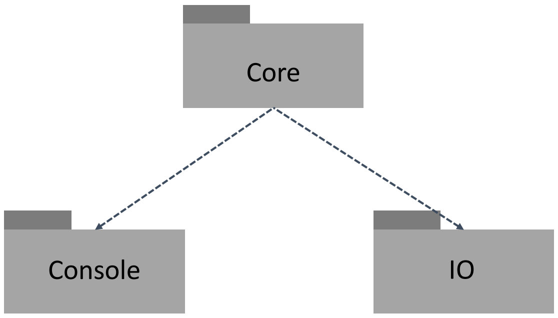
Ultimately, this tension drove the two stream implementations back into the package, as this would be the most convenient form for most of my target audience. Almost all of my end users will want one or the other of these streams.

Stream Stub, on the other hand, is only of interest to testers.

One of my customers requested a custom algorithm for flattening messages. That class, and the custom message type it supports form a separate package, distributed only to that customer.

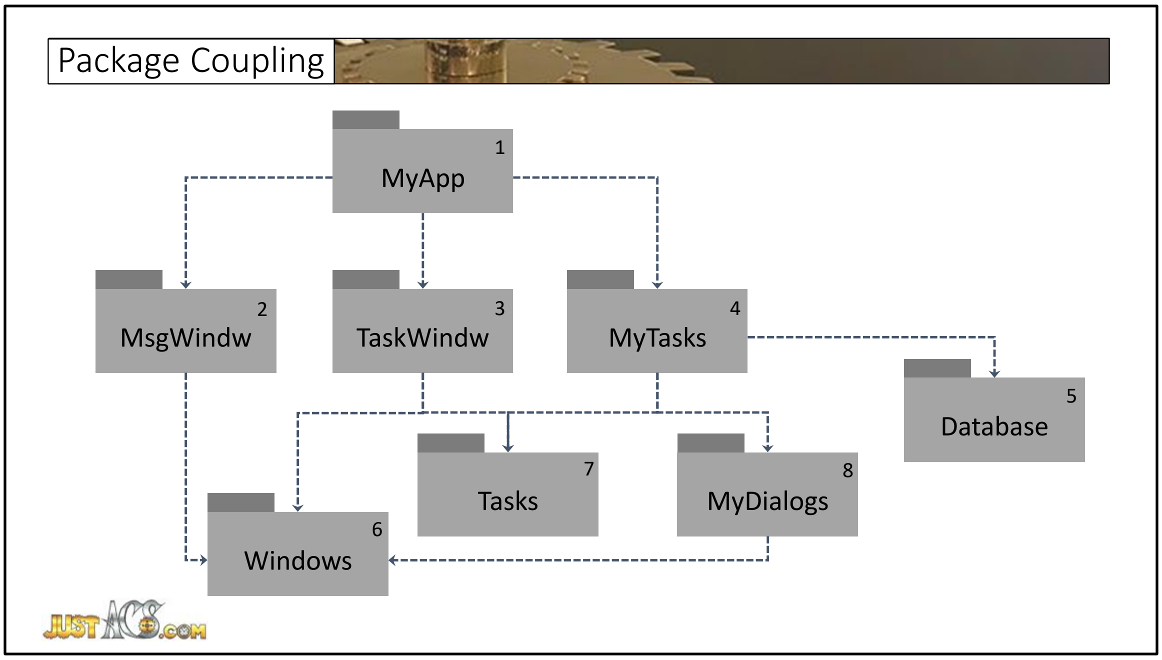


Package Coupling



The three principles of package coupling deal with the relationships between packages

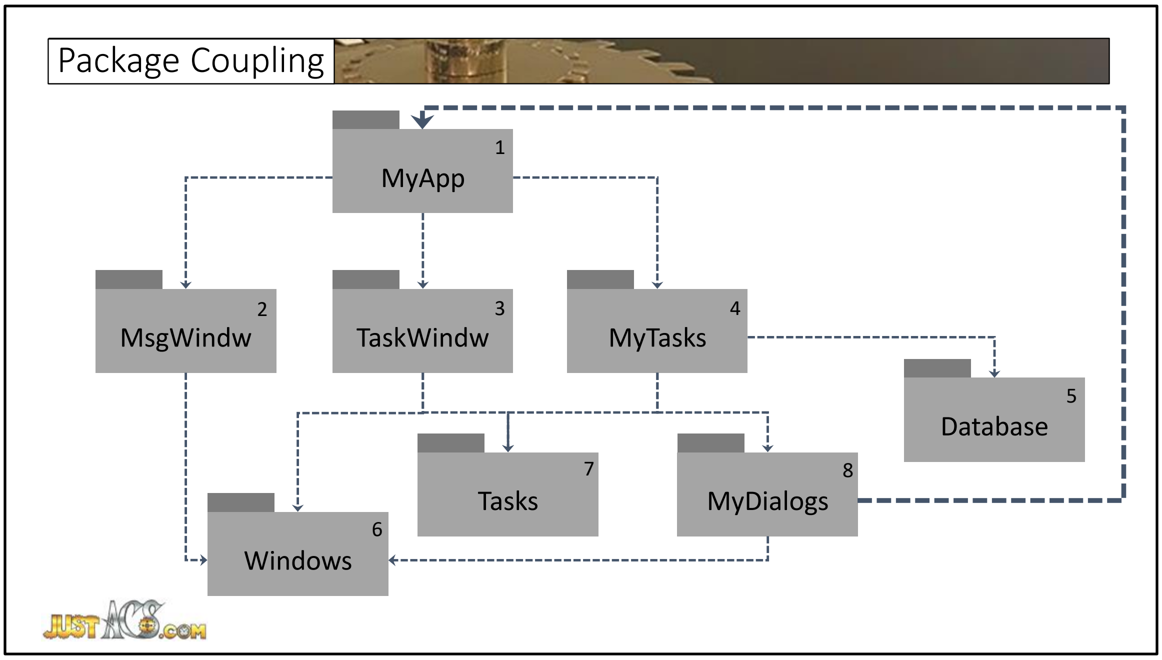
ACS



ACS

Acyclic-Dependencies Principle (ADP) - You stay late to finish up some work, but come in the next morning only to find that your code is broken--someone else stayed later and checked in changes that make your code very unhappy. Fixing your code breaks a third developer's code, and his fixes break yours again. This "morning-after syndrome" can be an unending nightmare of changes breaking other code, which in turn must be changed, which in turn...

Consider this hypothetical application. The dashed lines show dependencies; MyTasks, for example, depends on Windows, Tasks, and MyDialogs. All looks well. If you follow the dependencies, you cannot start at one package and wind up back at that package.

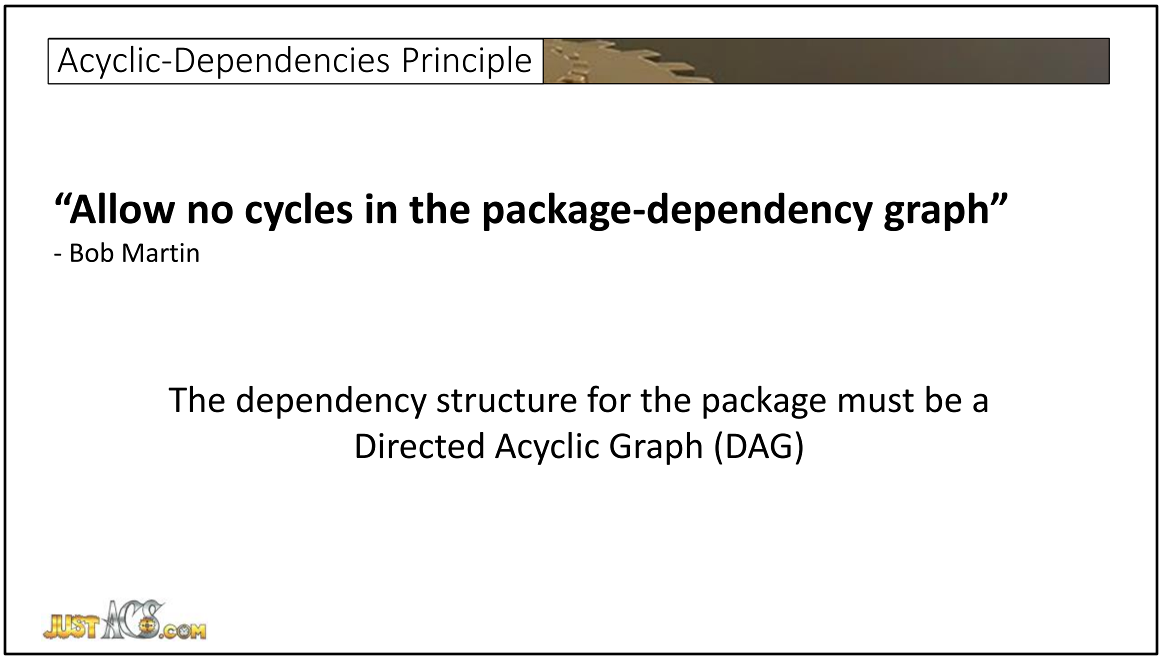

Package Coupling

ACS

Now, imagine a change to the software such that MyDialogs now depends on MyApp. This has the effect of making MyTasks dependent on every other package in the system. The effect is to require the developers to release MyApp, MyTasks, and MyDialogs at the same time. We are basically back to the monolithic build artifact from the start of this presentation, completely undermining the package design.

Cycles begin to make all packages dependent upon all other packages in the system. They dramatically increase build times to the point of pain (something one of us lives with daily on the C++ project he's working).

Dependency cycles among modules are a very serious problem, slowing development and wrecking deadlines. Note that mutual dependencies among *classes* are usually a problem if the classes are split across modules/packages/assemblies.

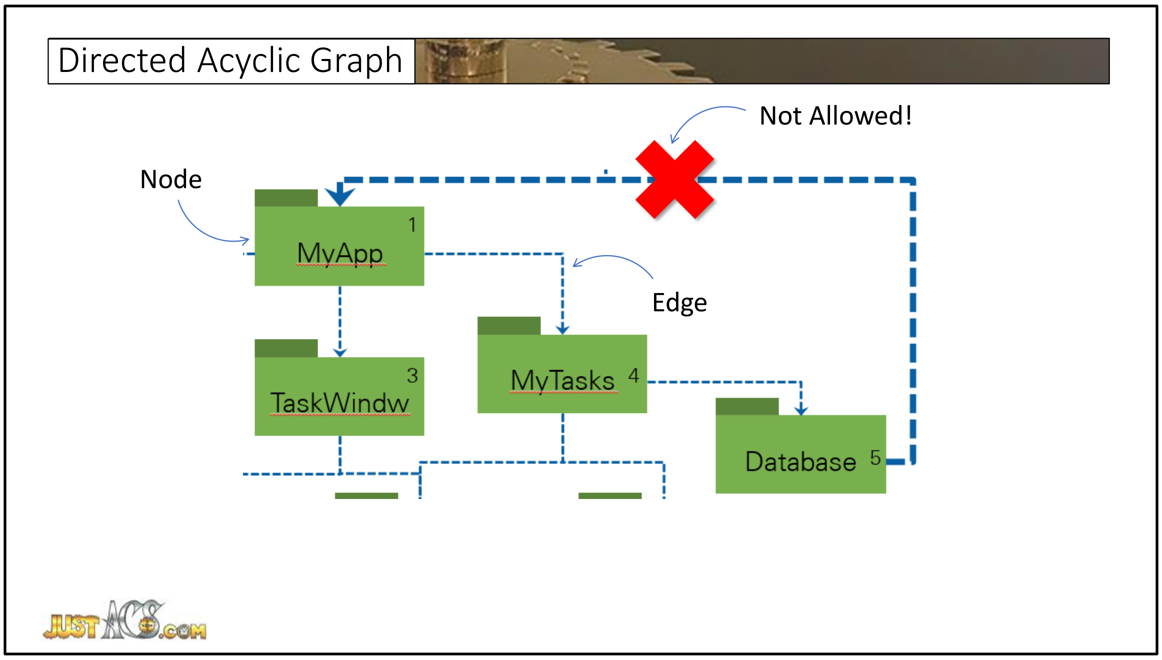


Acyclic-Dependencies Principle

“Allow no cycles in the package-dependency graph”
- Bob Martin

The dependency structure for the package must be a Directed Acyclic Graph (DAG)

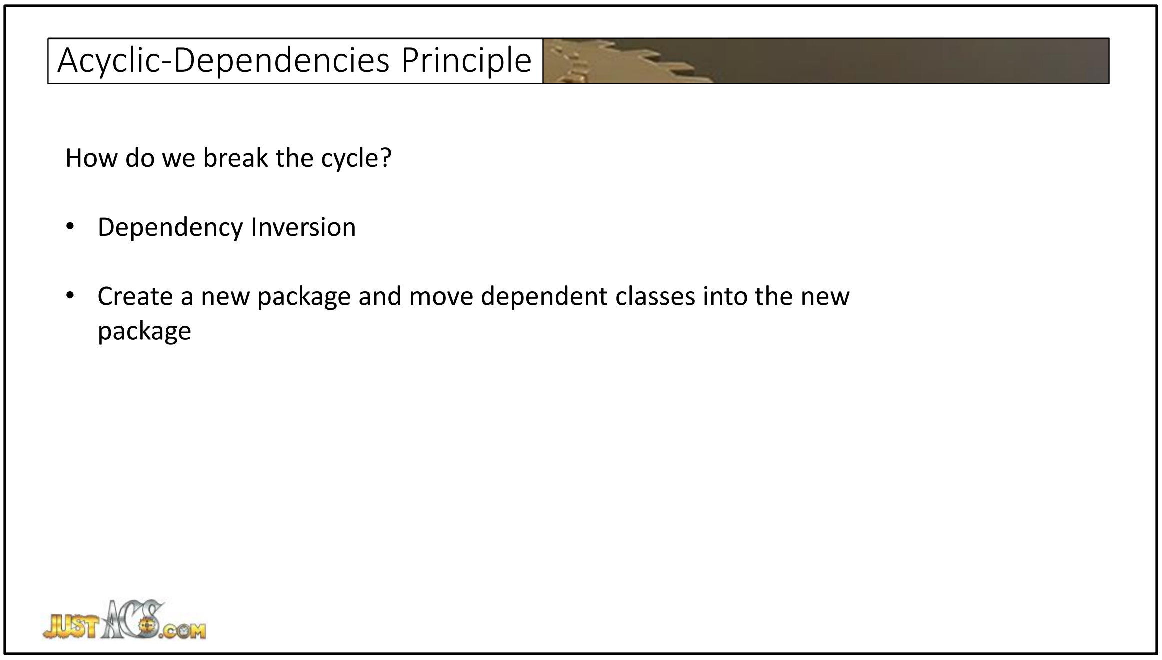
ACS



Directed Acyclic Graph

ACS

A package diagram is an example of a Directed Graph. Directed graphs have nodes (the packages) and directed edges (the dependency arrows). We require that this graph be acyclic – meaning there are no cycles in the graph.



Acyclic-Dependencies Principle

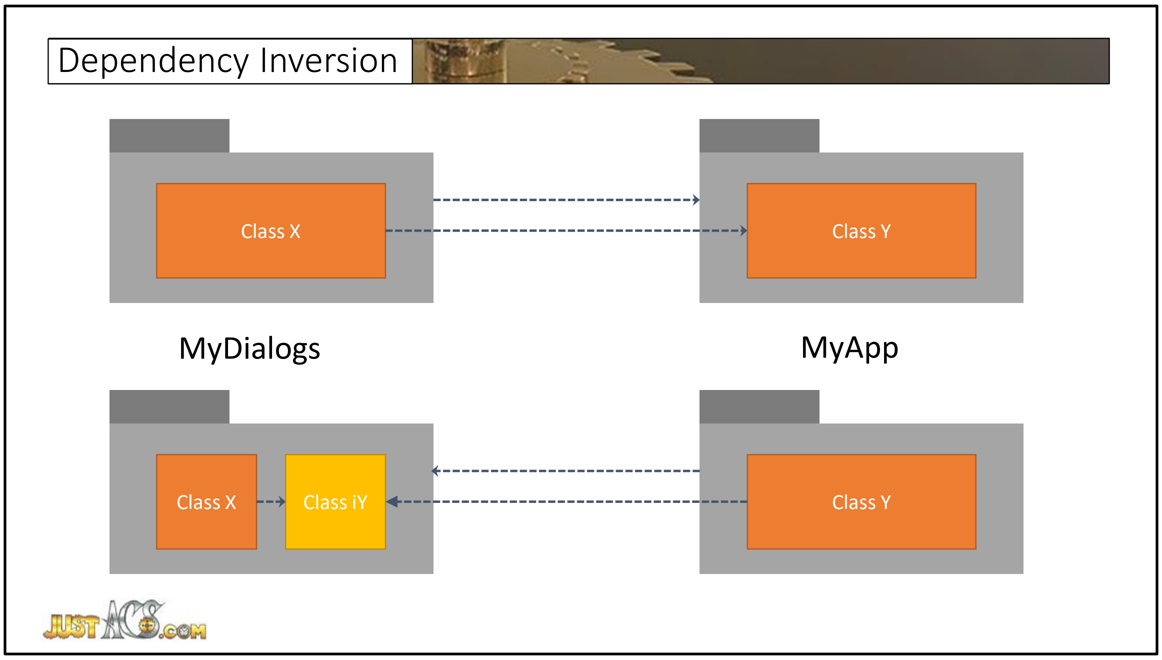
How do we break the cycle?

- Dependency Inversion
- Create a new package and move dependent classes into the new package

JUSTACS.COM

ACS

If we find we have a cycle, we can break it in two ways.

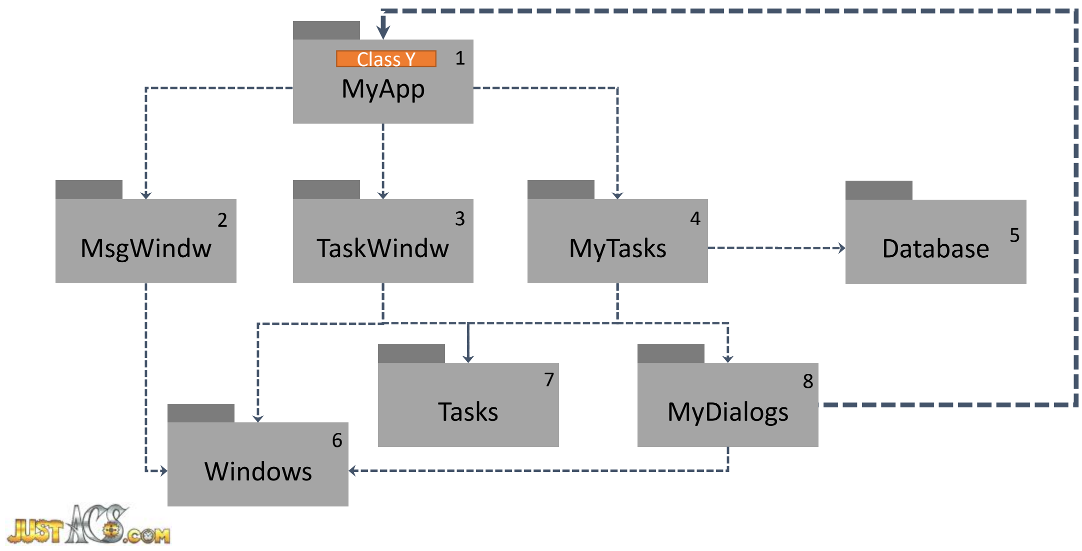


Dependency Inversion

ACS

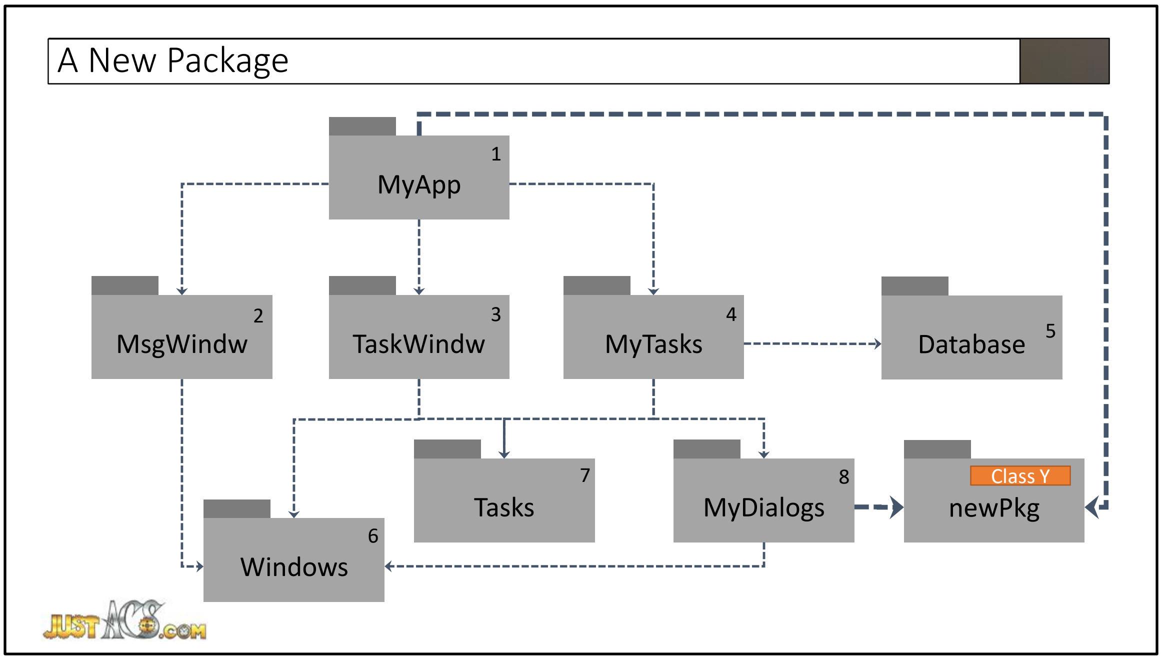
We can implement dependency inversion. Imagine Class X depends on Class Y. We can change the class hierarchy so that Class Y inherits from an abstract parent, and Class X depends on that abstract parent. We then move Class iY into the package that contains X. This flips the dependency, which breaks the cycle.

A New Package



ACS

The second option is to move the dependency into a new package. Here we see the original package design, where MyDialogs depends on a Class Y, in MyApps.

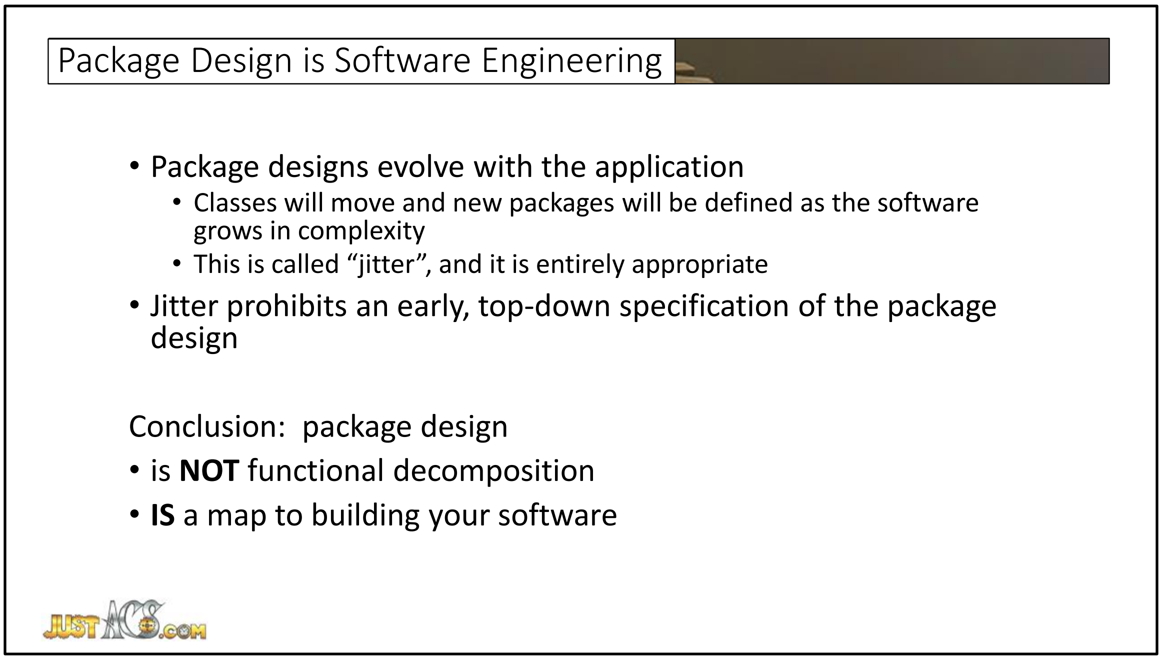


A New Package

ACS

If we move Class Y into a new package, MyApp and MyDialogs will now depend on the new package, which breaks the cycle.

You can expect to use both of these techniques often, over the life of your project.



Package Design is Software Engineering

- Package designs evolve with the application
  - Classes will move and new packages will be defined as the software grows in complexity
  - This is called “jitter”, and it is entirely appropriate
- Jitter prohibits an early, top-down specification of the package design

Conclusion: package design

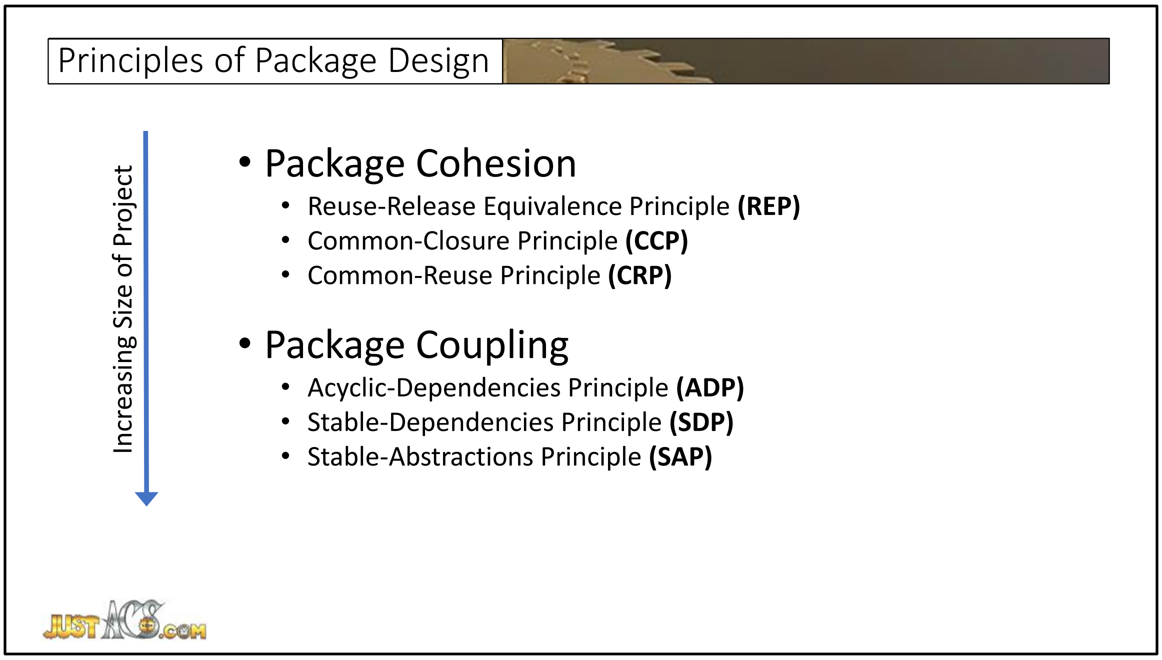
- is **NOT** functional decomposition
- **IS** a map to building your software

ACS

Packages are the basic unit of work, and your package design is a map for how to build and distribute your software.

Since package design is about how you work on software, not about how the software works, it is entirely a software engineering task, akin to maintaining the source code control repository for your project, and closely tied to your build services and deployment strategy.

TODO: Side comment about testability?



Principles of Package Design

Increasing Size of Project

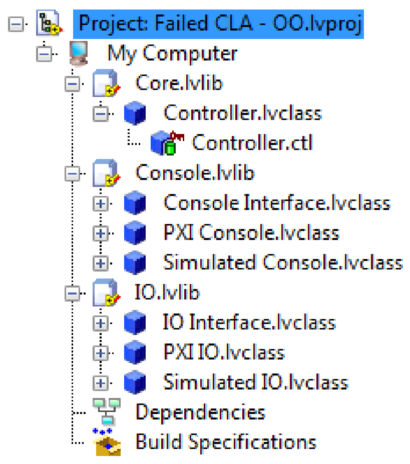
- Package Cohesion
  - Reuse-Release Equivalence Principle **(REP)**
  - Common-Closure Principle **(CCP)**
  - Common-Reuse Principle **(CRP)**
- Package Coupling
  - Acyclic-Dependencies Principle **(ADP)**
  - Stable-Dependencies Principle **(SDP)**
  - Stable-Abstractions Principle **(SAP)**

ACS

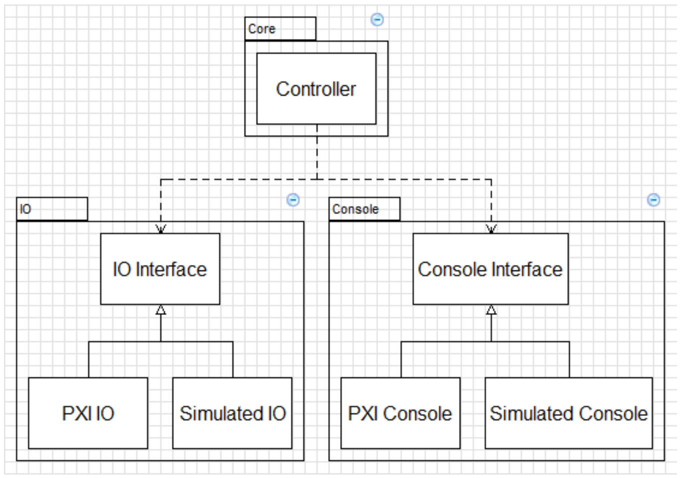
Revisiting our list of package design principles, we can discern an order of precedence.

In the early stages of development, you have little code, and no real need for a build map. Packages start to become important as you begin to need to manage dependencies and localize changes, so you define some initial packages and use the CCP to allocate code to them. The CRP becomes important as you begin to realize reuse opportunities, and you begin to apply the ADP when you get enough packages to be concerned about cycles.

Stability



Project: Failed CLA - OO.lvproj
- My Computer
  - Core.lvlib
    - Controller.lvclass
      - Controller.ctl
  - Console.lvlib
    - Console Interface.lvclass
    - PXI Console.lvclass
    - Simulated Console.lvclass
  - IO.lvlib
    - IO Interface.lvclass
    - PXI IO.lvclass
    - Simulated IO.lvclass
  - Dependencies
  - Build Specifications



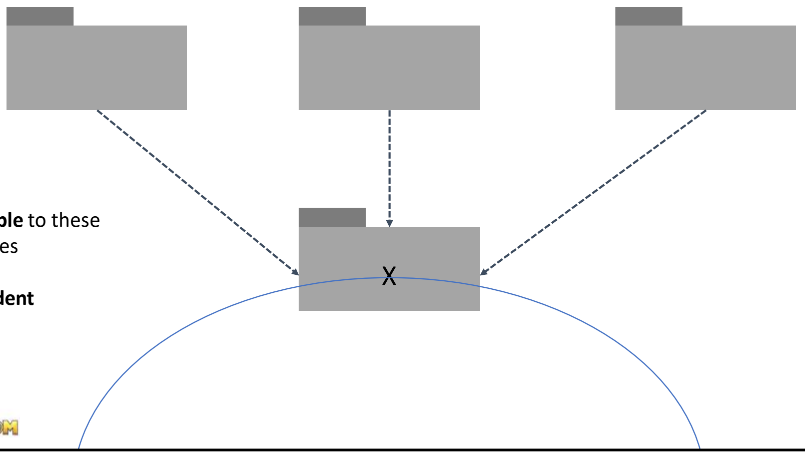
JON

Let’s return to our failed CLA exam. Here, we can see the project view and a UML diagram for the exam solution.

Changing a package early in the build map usually requires changing the downstream packages. Looking at the CLA example, a change in in the IO or Console packages will likely lead to changes in the Core package as well. We see this borne out in real applications, where changing IO hardware often leads to undesirable changes in the rest of the system.

If possible, we would like to minimize the frequency with which the Core package changes. This leads us to the concept of stability.

A Stable Package



X is **responsible** to these three packages

X is **independent**

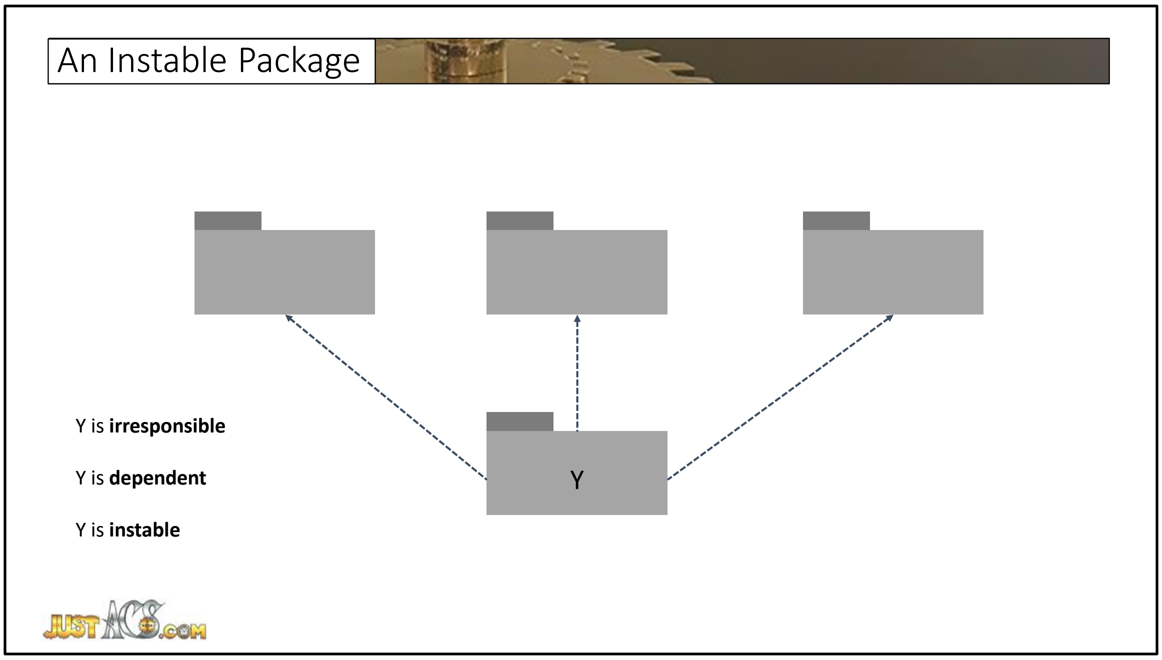
X is **stable**

JON

Three different packages depend on X. It is therefore very difficult to change X, because changes to X will likely require changes to the other three packages. Note that this is true regardless of the quality or complexity of the code in X.

X is a stable package, and not for any reason related to the contents of X. X is stable because changing it will *generate* a lot of additional work in its dependent packages.

TODO: add an animation showing the proliferation of changes, from X to dependent packages



An Instable Package

Y is **irresponsible**

Y is **dependent**

Y is **instable**

JON

Y, on the other hand, is instable. Matters of code complexity aside, it is relatively easy to change Y, because nothing depends on it.

Stable vs. Instable

An Instable Package

Direction of Dependency

Which would you rather modify?

A Stable Package

JON

It is important to note that, when we talk about the stability of a package, we are not talking about the quality of the code in the package. We are instead describing how much work (technical debt) we will incur if we change the package in some way

The game Jenga provides an excellent metaphor for package stability. (Giant Jenga is a much better metaphor than Tiny Twister.)

Here we see a game of Jenga in process. (Describe game play, if needed.) Imagine that each block in the game is a package in our software. Each block is pretty much the same, and they are all of comparable quality.

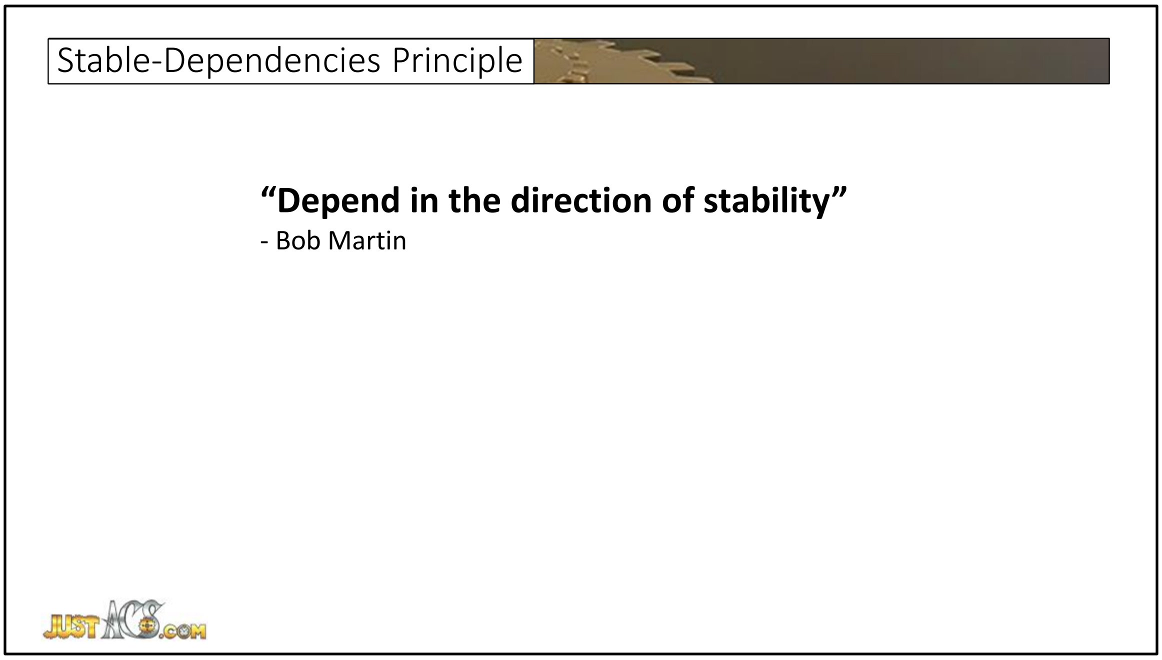
[Build] The arrow of dependency points down, meaning blocks higher in the stack depend on blocks lower in the stack.

[Build] This is an instable package. No one depends on it. You can take it off the stack, do what you will to it, and replace it.

[Build] This is a stable package.

[Build] Which package would you rather work on?

This is the essence of our next principle.



Stable-Dependencies Principle

“Depend in the direction of stability”
- Bob Martin

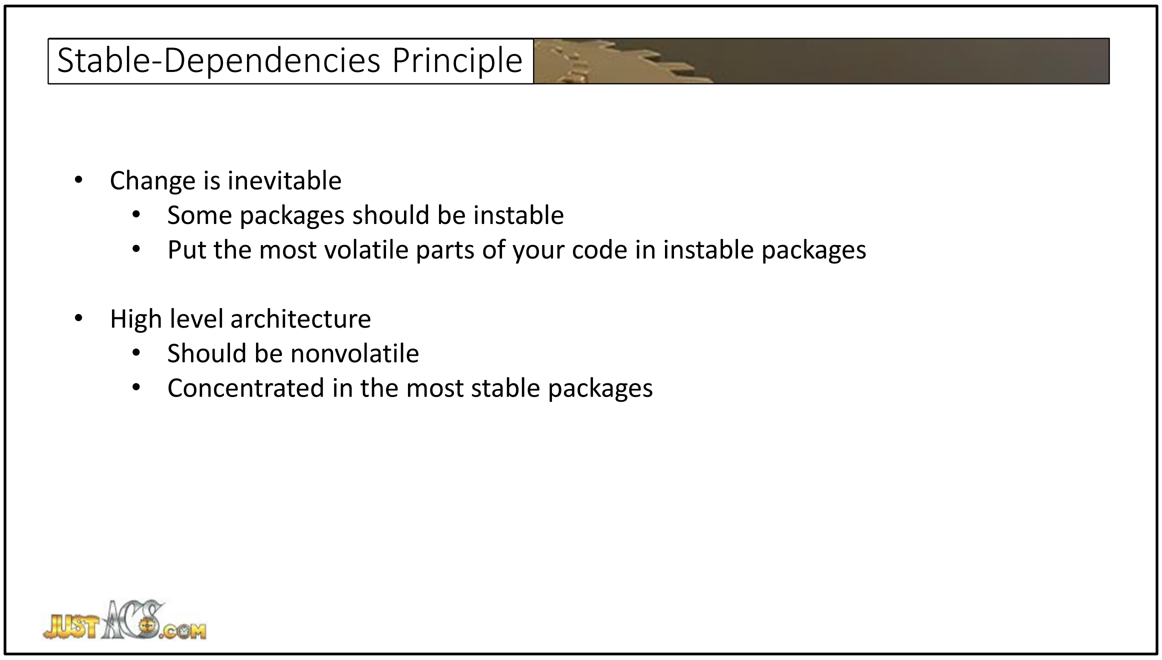
JON

Stable-Dependencies Principle (SDP) is based on the understanding that code has volatility, and that changes ripple through a system along dependency lines. From the point-of-view of volatility, dependency is transitive.

If the majority of the system's classes are dependent upon a volatile bit of code, then we can expect a high frequency of system breakage and rippling changes. If, instead, a system depends on nonvolatile code, then breakage should not ripple through the system very often at all.

Some modules should change frequently, and should be volatile. Other modules are essential core abstractions and should not change very often.

Therefore the package dependency graph should flow from instable packages (packages that are easy to change) to stable or "responsible" packages (packages that are hard to change). In a sense, this is the dependency inversion principle applied to packages: You should depend only upon things that are unlikely to change.



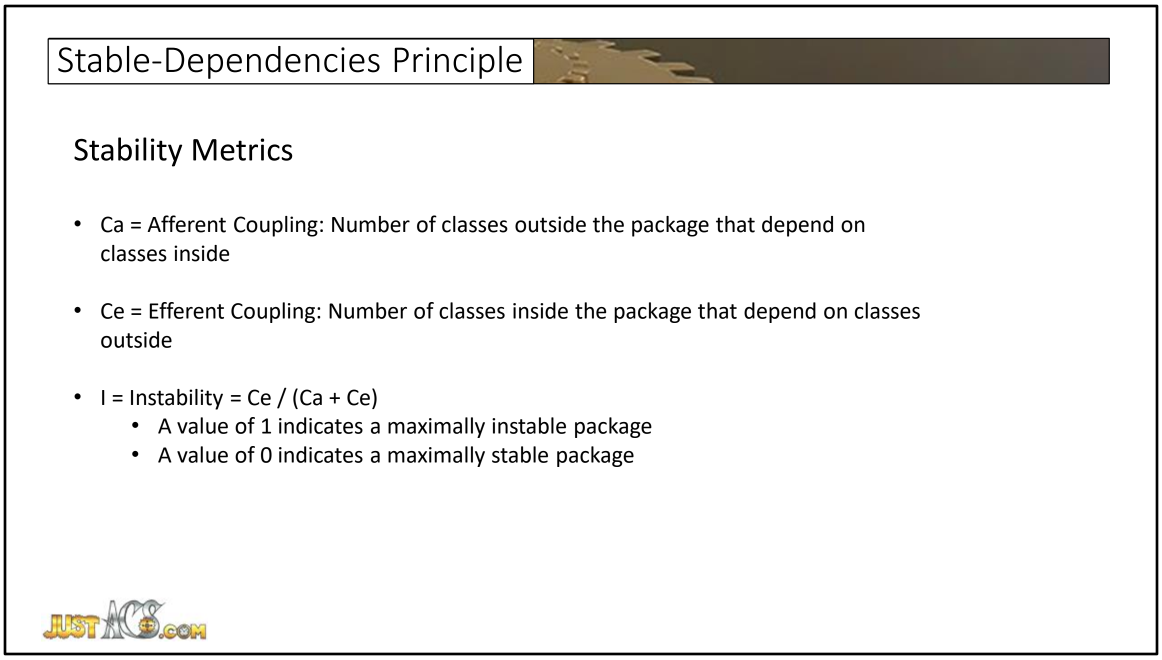
Stable-Dependencies Principle

- Change is inevitable
  - Some packages should be instable
  - Put the most volatile parts of your code in instable packages
- High level architecture
  - Should be nonvolatile
  - Concentrated in the most stable packages

JON

Sustainable software must lend itself to change. Certain parts of our code base are and should be volatile. The Stable-Dependencies Principle drives us to group those portions of our code into instable packages – i.e. packages that have few or no dependents. In contrast, the parts of the code base that should not change should be concentrated in the most stable packages – i.e. the packages with lots of dependents.

Designs need to be flexible to be useful over time. But if the high level architecture is in highly stable packages that are very hard to change, how can we adapt and extend our software to meet changing needs?



Stable-Dependencies Principle

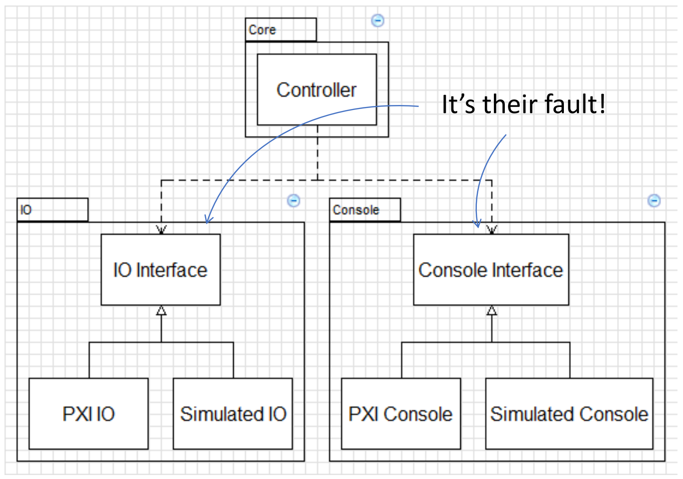
Stability Metrics

- Ca = Afferent Coupling: Number of classes outside the package that depend on classes inside
- Ce = Efferent Coupling: Number of classes inside the package that depend on classes outside
- I = Instability = Ce / (Ca + Ce)
    - A value of 1 indicates a maximally instable package
    - A value of 0 indicates a maximally stable package

JON

We can define a measure of instability, based on the number of dependencies into and out of the package.

Instability



Core Metrics

Ca = 0
Ce = 1
I = 1

Console Metrics

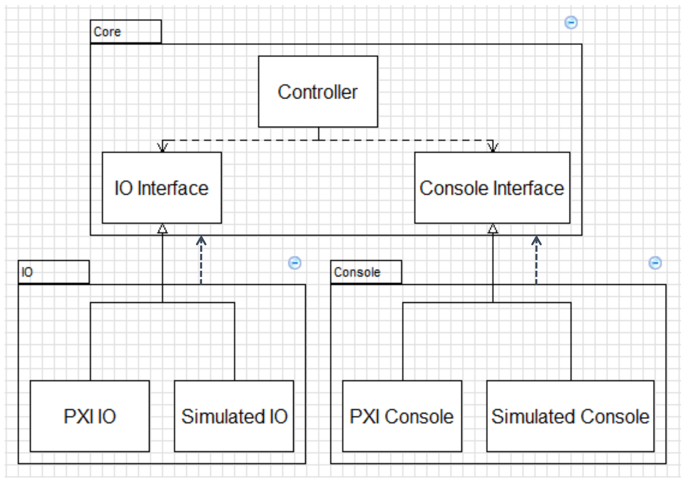
Ca = 1
Ce = 0
I = 0

JON

The metrics for the CLA example don't look very good, which matches our intuition. **[Build]:** It's due entirely to IO Interface and Console Interface.

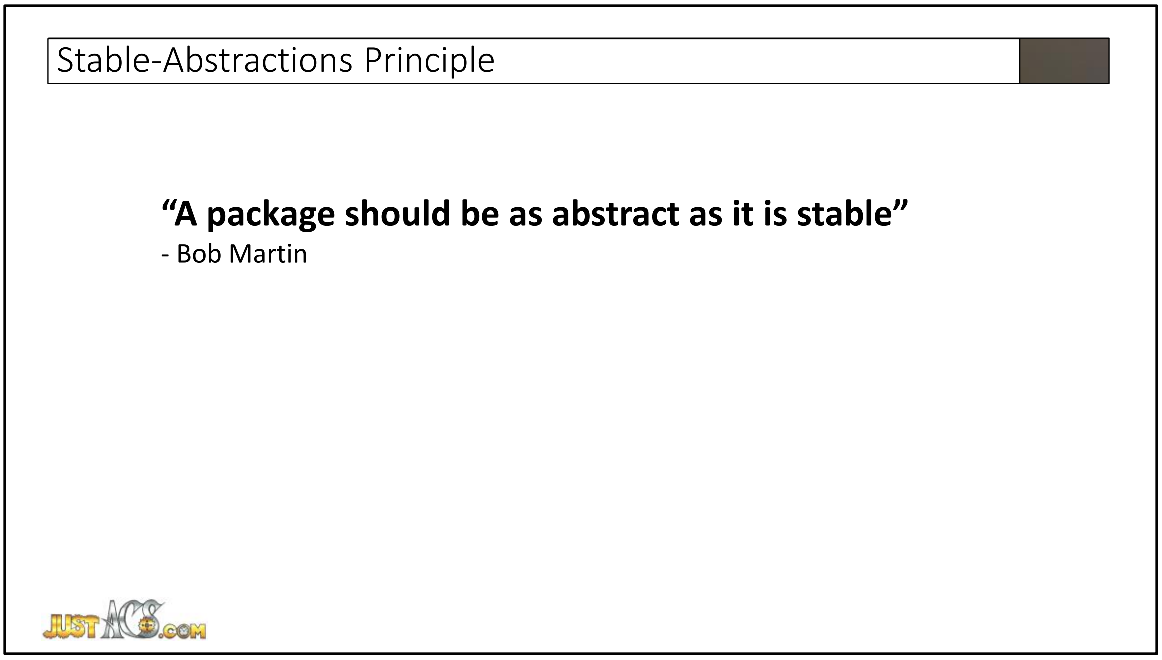
Which leads us to our last principle

Stable-Abstractions Principle



A stable package should be abstract so that its stability does not prevent it from being extended.

JON



Stable-Abstractions Principle

“A package should be as abstract as it is stable”
- Bob Martin

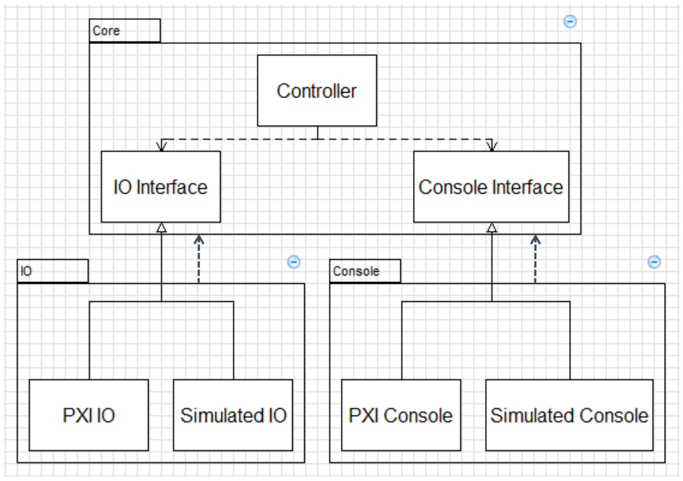
JON

Stable-Abstractions Principle (SAP) - Per Uncle Bob, this principle sets up a relationship between stability and abstraction. A stable package should be as abstract as possible, thus ensuring that its "stability does not prevent it from being extended." In contrast, an instable (not responsible) package should contain lots of concrete classes whose details can be easily changed.

This further reflects the Stable Dependencies principle, with the added observation that the reason for interfaces is to give safe, stable dependencies to clients of an abstraction. This drives us to the understanding that we should depend on the parts of the system built specifically to provide freedom from volatile bits (implementations).

In dynamic languages, SAP is a harder metric to automate. Clearly the principle still applies. There will be classes whose purpose is to provide a stable interface and other classes that provide easily-changeable behaviors. The only problem is that not having declared interfaces makes it harder to automate the process of assessing abstractness.

Stable-Abstractions Principle

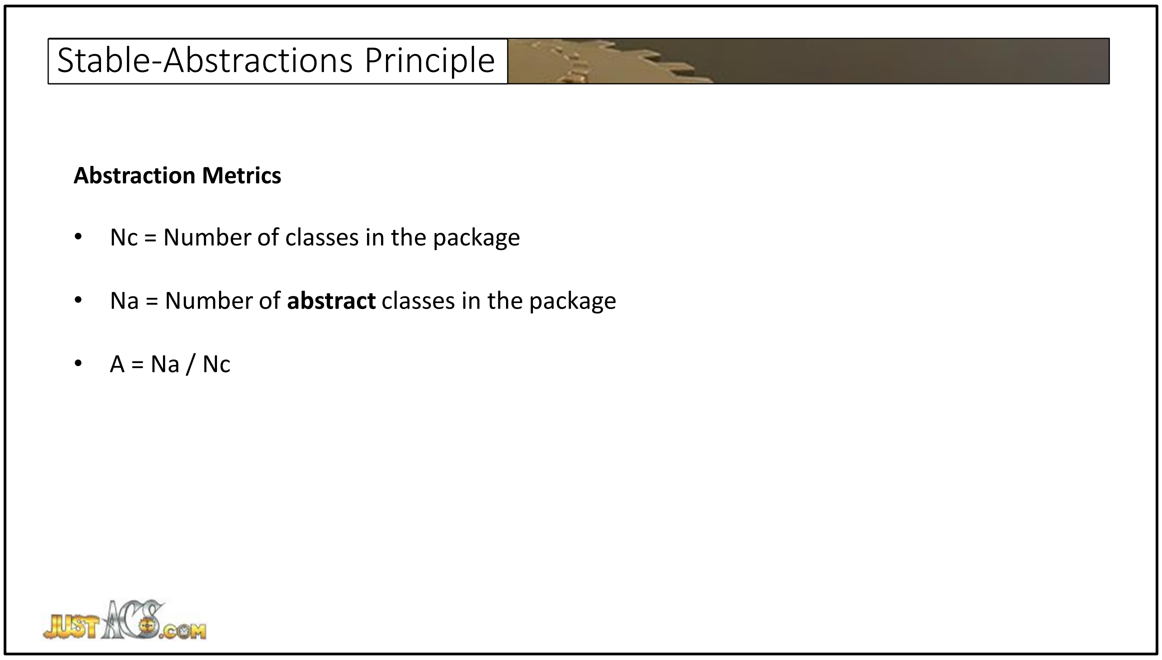


- **SDP:** dependencies run in the direction of stability
- **SAP:** stability implies abstraction

- **Dependencies run in the direction of abstraction**

JON

May be worth showing stability metrics here again.



Stable-Abstractions Principle

Abstraction Metrics

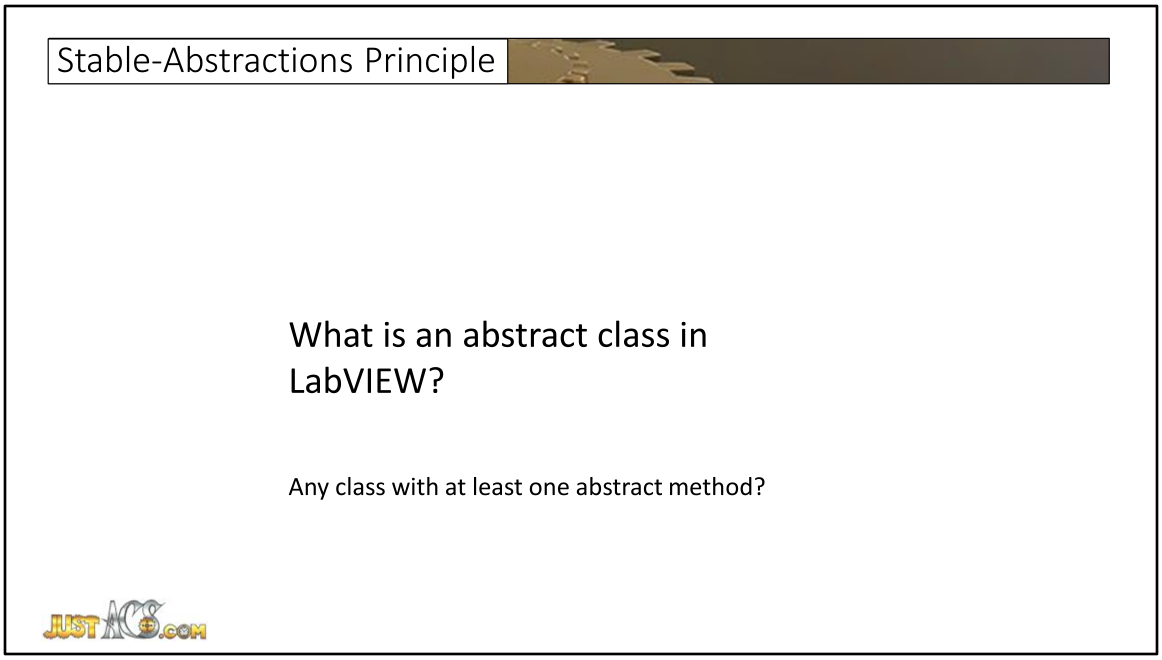
- Nc = Number of classes in the package
- Na = Number of **abstract** classes in the package
- A = Na / Nc

JON

Stable-Abstractions Principle (SAP) - Per Uncle Bob, this principle sets up a relationship between stability and abstraction. A stable package should be as abstract as possible, thus ensuring that its "stability does not prevent it from being extended." In contrast, an instable (not responsible) package should contain lots of concrete classes whose details can be easily changed.

This further reflects the Stable Dependencies principle, with the added observation that the reason for interfaces is to give safe, stable dependencies to clients of an abstraction. This drives us to the understanding that we should depend on the parts of the system built specifically to provide freedom from volatile bits (implementations).

In dynamic languages, SAP is a harder metric to automate. Clearly the principle still applies. There will be classes whose purpose is to provide a stable interface and other classes that provide easily-changeable behaviors. The only problem is that not having declared interfaces makes it harder to automate the process of assessing abstractness.



Stable-Abstractions Principle

What is an abstract class in LabVIEW?

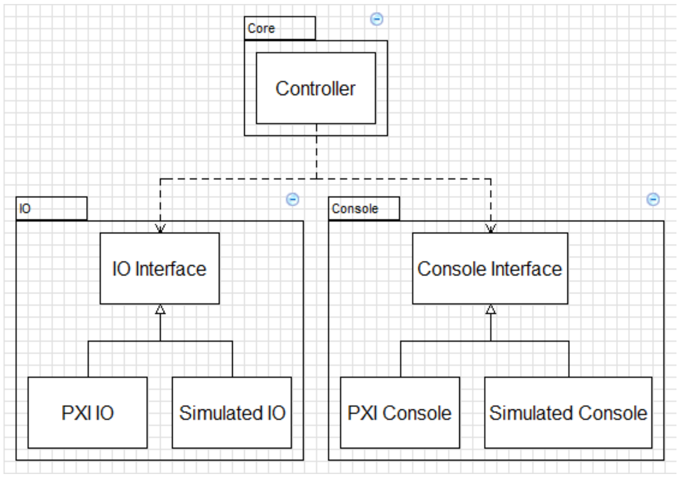
Any class with at least one abstract method?

JON

This is an open question.

Let’s assume that a class with at least one abstract method is an abstract class.

Abstraction



Core Metrics

Na = 0
Nc = 1
A = 0

Console Metrics

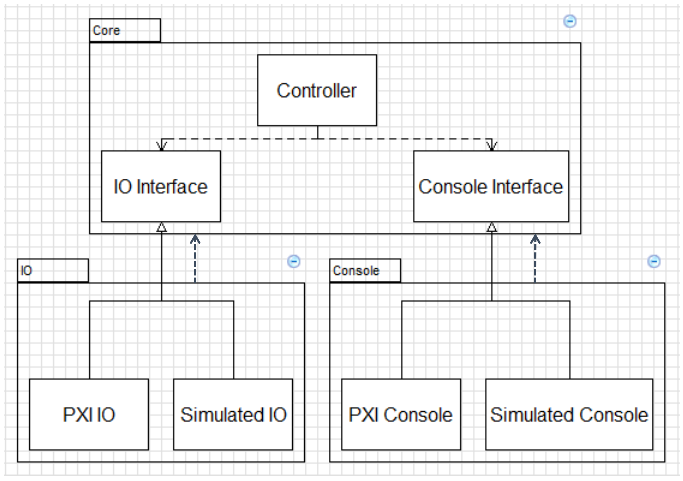
Na = 1
Nc = 2
A = 0.5

JON

Here, we have applied the abstraction metric to our design

Again, this is bad. But the two metrics taken together suggest a solution.

A Better Solution



Core Metrics

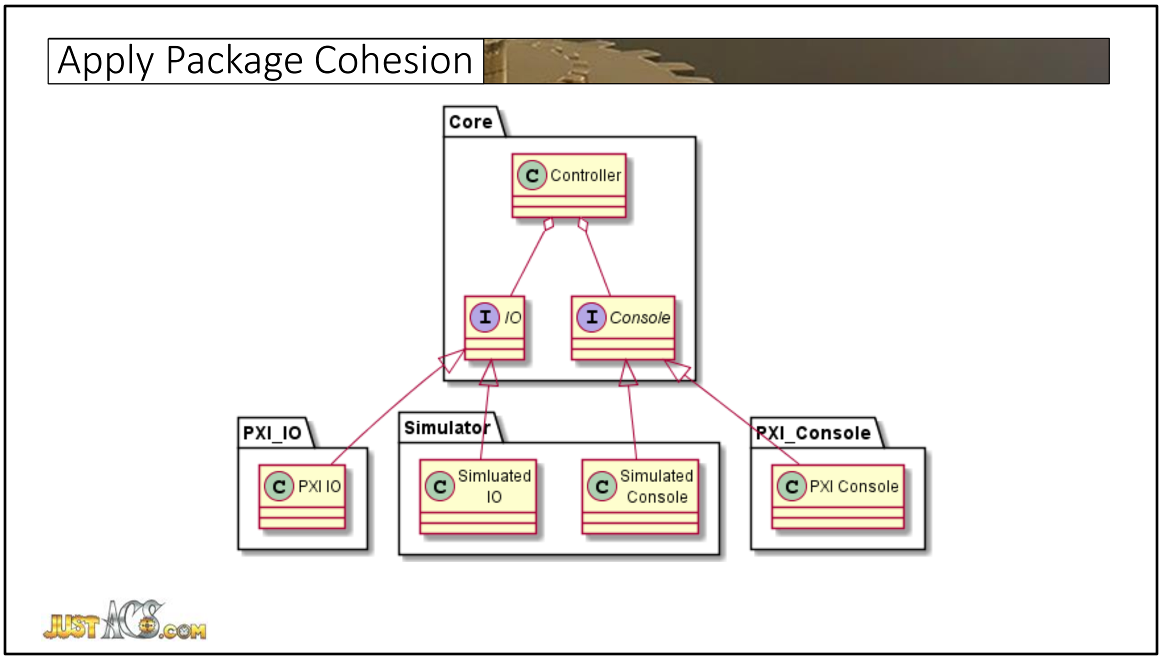
I = 0
A = .667

Console Metrics

I = 1
A = 0

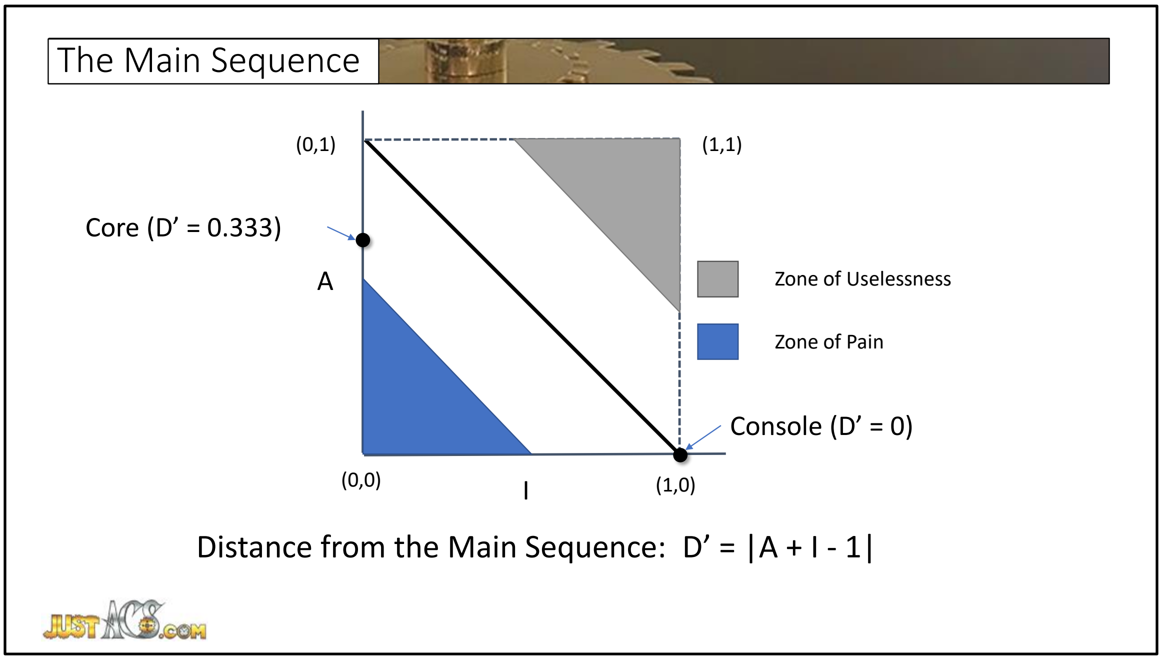
JON

Our metrics improve significantly if we move the interface classes, which are abstract, into the Core package. This makes sense; the interface classes define how the IO and Console talk to the controller, so they are likely to change if the controller changes



Apply Package Cohesion

If we go back and apply the rules for package cohesion, we can get close to an optimal arrangement for this set of classes. Common Reuse argues for grouping the simulator and PXI code in separate packages. Common Closure argues for breaking up the hardware package even further, as our hypothetical end user may well wish to have IO and console hardware vary independently of each other.



The Main Sequence

Distance from the Main Sequence: D' = |A + I - 1|

ACS

It is a useful exercise to create a scatter plot for your packages, using A and I as the axes. Maximally stable and abstract packages end up in the upper left corner (0,1) and maximally instable and concrete packages end up in the lower right (1,0). Not all packages can or should be at the two maxima.

However, it is pretty clear that we would like to avoid the other two corners. Packages at (0,0) are both concrete and stable. Such packages cannot be extended, only modified, and the modifications will likely drive changes in other packages. The area around (0,0) is therefore called the Zone of Pain.

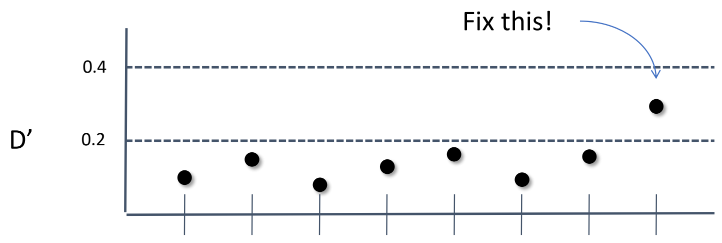
Now, life is pain, and anyone who tells you differently is trying to sell you something. You will certainly encounter packages that live in the Zone of Pain. Database schemas are one example; they are volatile and highly responsible (i.e. they have lots of dependents), and changing them is often painful. Some utility packages, however, may by both stable and concrete, but change so rarely as to not really be a concern.

The other corner contains packages that are highly abstract, but completely irresponsible (i.e. no one depends on them). That's a pretty useless thing. It's worth asking yourself why such packages are in your code base.

There is a middle zone on our scatter plot that defines a sort of "main sequence" – packages on the main sequence have a good balance of abstraction and stability that makes them both flexible and useful. Plotting the main sequence gives you a useful snapshot of the overall health of your package set. The distance from the optimal main sequence, D', is a useful metric for the overall health of any given package.

[Build] As an example, here are where our Core and Console packages fall on the main sequence.

Tracking D’



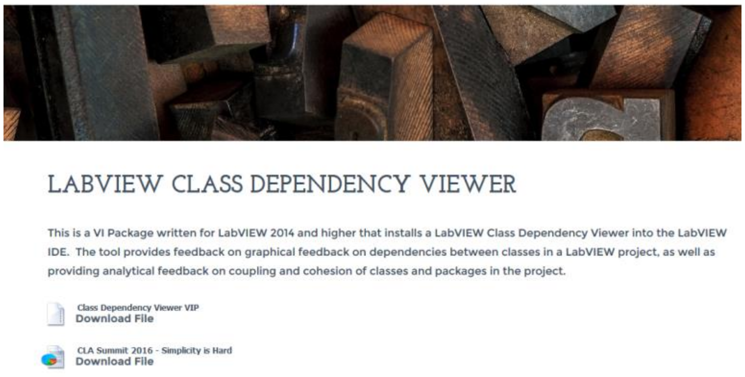
Releases

Distance from the Main Sequence: D’ = |A + I - 1|

ACS

You can also track D’ over time for individual packages. You can set control limits; an excursion would result in a change to the package design.

The Package Metrics Tool



LABVIEW CLASS DEPENDENCY VIEWER

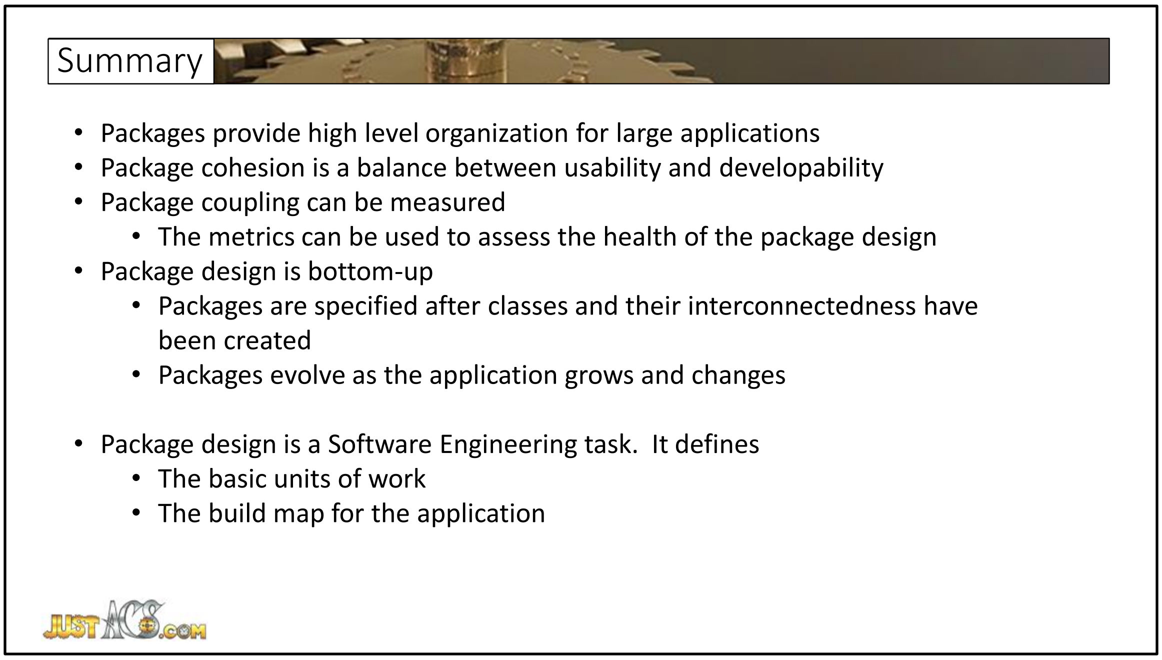
This is a VI Package written for LabVIEW 2014 and higher that installs a LabVIEW Class Dependency Viewer into the LabVIEW IDE. The tool provides feedback on graphical feedback on dependencies between classes in a LabVIEW project, as well as providing analytical feedback on coupling and cohesion of classes and packages in the project.

Class Dependency Viewer VIP
Download File

CLA Summit 2016 - Simplicity is Hard
Download File

http://www.labviewcraftsmen.com/tools.html

It’s part of the LabVIEW Class Dependency Viewer



Summary

- Packages provide high level organization for large applications
- Package cohesion is a balance between usability and developability
- Package coupling can be measured
  - The metrics can be used to assess the health of the package design
- Package design is bottom-up
  - Packages are specified after classes and their interconnectedness have been created
  - Packages evolve as the application grows and changes

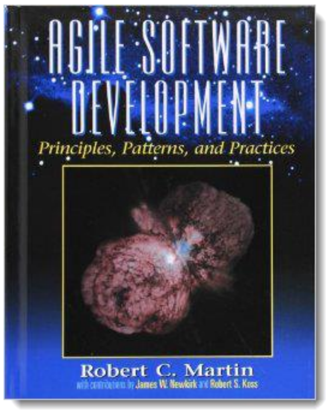
- Package design is a Software Engineering task. It defines
  - The basic units of work
  - The build map for the application

JUSTACS.COM

ACS

TODO: update this a bit.

Further Reading


AGILE SOFTWARE DEVELOPMENT
Principles, Patterns, and Practices
Robert C. Martin
with contributions by James W. Newkirk and Robert S. Koss

ACS

Further reading.

Contact Us!

- Allen C Smith
  - Linkedin/in/allencsmith
  - www.justacs.com

- Jon McBee
  - info@composedsystems.com