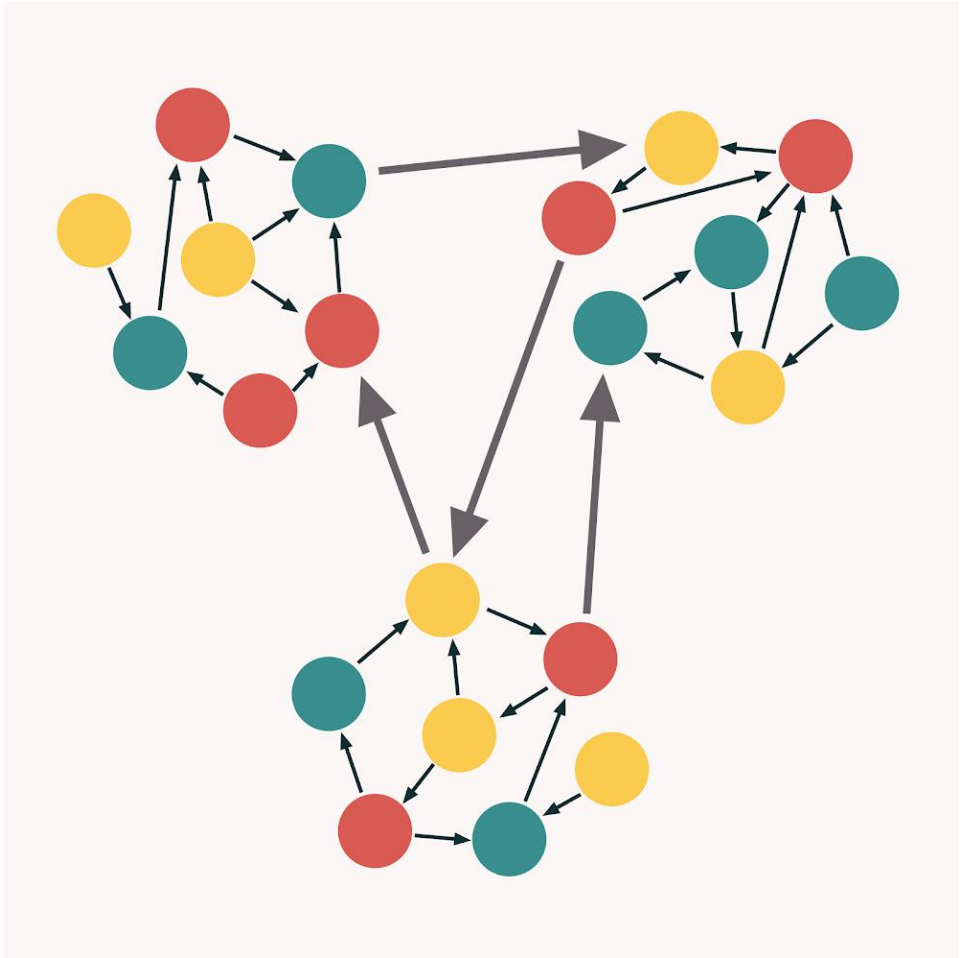
Principles of component(Module) design

Principles for Package and Component Design

"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".

When we design a package or component, we need a guide to make sure that the packages or Components are in **High Cohesion and Low Coupling**.

Principles for Package and Component Design (Con.)



Now Let's talk about Components and Packages!

What's a component

- A relocatable library that contains code
- Ready to run
- Jar dll

What's indie a component?

- Functions
- Classes are the first layer of cohesion for functions
- Data dare unimportant that's why they are private

Cohesion

IN PHYSICS

- The sticking **together** of particles of the **same substance**.

In human society

- Communities.

Similar things come together.

Cohesion （Cont.）

In software development, cohesion refers to the degree to which the elements within a module, component, or package are **closely related** and focused on a **single responsibility** or a specific set of related tasks.

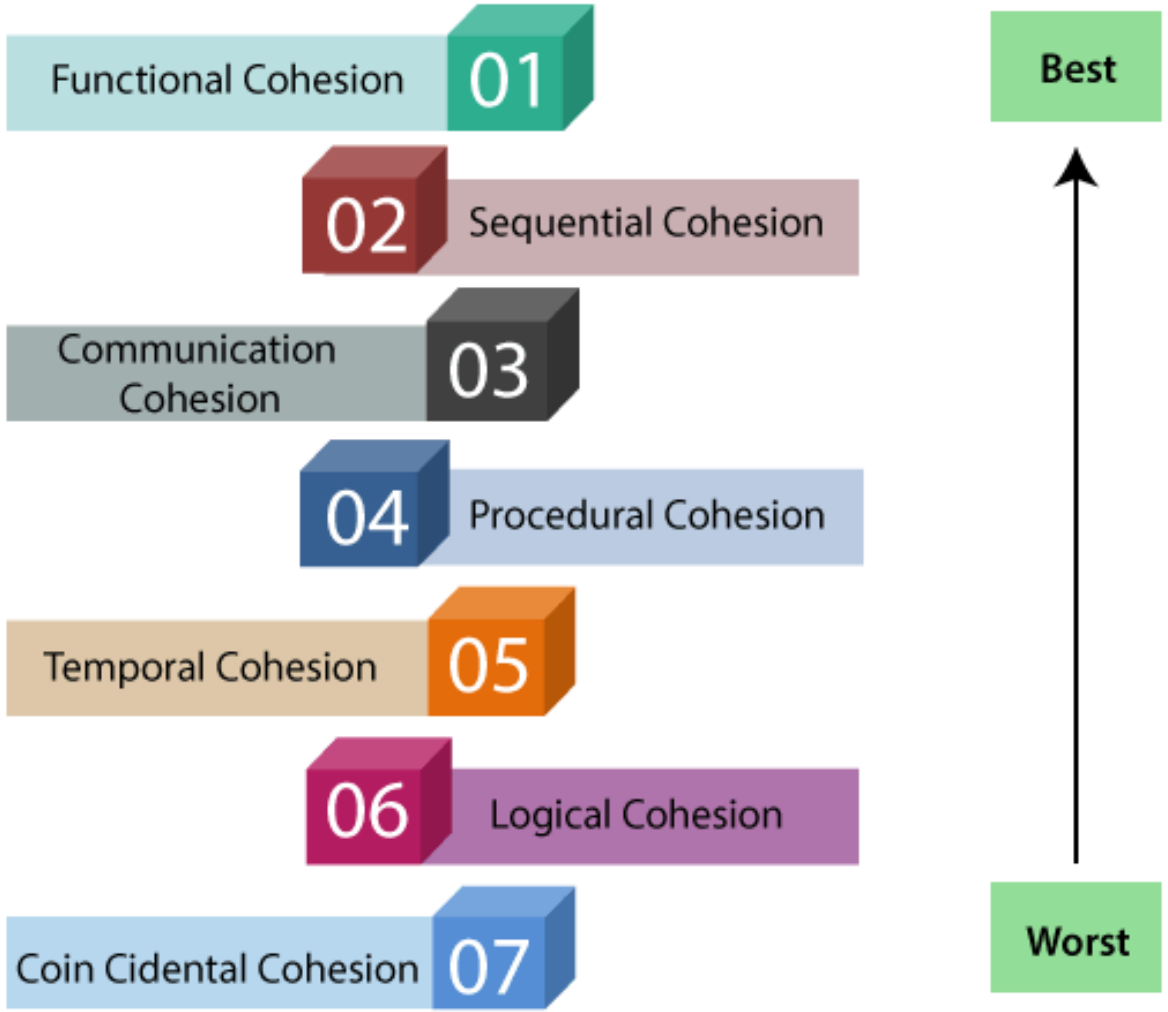
(In java): it could be used to measure the closeness of functions, and properties in a class, or many classes in a package.

Cohesion (Cont.）

We always want HIGH Cohesion,

High cohesion refers to a **desirable** property of software design where each module, class, or component in a system is focused on a single, well-defined responsibility or functionality.

Cohesion - Type of Cohesions



Cohesion - Type of Cohesions (Cont.)

- **Functional cohesion:** Elements perform a single, well-defined task; highest and most desirable cohesion.
- **Sequential cohesion:** Elements arranged in sequence, output of one serves as input for the next.
- **Communicational cohesion:** Elements operate on the same input/output data, sharing a common data context.
- **Procedural cohesion:** Different tasks executed in a specific sequence, elements not inherently related.

Cohesion - Type of Cohesions (Cont.)

- **Temporal cohesion:** Elements related by execution time, sharing a common time of execution.
- **Logical cohesion:** Elements related by common theme or category, performing different tasks.
- **Coincidental cohesion:** Elements arbitrarily grouped, no meaningful relationship; lowest and least desirable cohesion.

Cohesion - Benefits of high Cohesion

- **Enhanced readability:** Cohesive modules are focused, making them easier to understand for developers.
- **Improved reusability:** Highly cohesive modules perform specific tasks, increasing reusability in other applications.
- **Easier debugging:** Single-responsibility modules simplify debugging and maintenance.
- **Fewer errors:** High cohesion reduces error likelihood when making changes, minimizing unintended side effects.
- **Better testability:** Clear, self-contained responsibilities enable more focused and effective test cases.
- **Modularity and flexibility:** Cohesive modules promote adaptable systems, facilitating updates, extensions, or replacements.

How to archive high cohesion when creating the components?
Using Principles of Component Cohesion!

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

Release Reuse Equivalence Principle (REP)

"The granule of reuse is the granule of release"

This means that if a set of classes is intended to be reused together, they should be versioned and released together as a cohesive unit

Release Reuse Equivalence Principle (REP) Example

```
StringUtils (library)
│                          V 1.0
└──string
    │   StringFormatter.java
    │   StringValidator.java
```

```
StringUtils (library)                V 2.0
│
└──string
    │   StringFormatter.java
    │   StringValidator.java
    │   StringEncoder.java
```

Release Reuse Equivalence Principle (Cont.)

- The Use of module management tools (e.g., Maven, Leiningen, RVM) has been due to an increase in reusable components and component libraries, leading to the age of software reuse.
- **Release numbers** used to track Reusable software components through a release process .
- Developers may decide to continue using old releases based on the changes in the new releases, making release notifications and documentation crucial for informed decision-making.

Common Closure Principle (CCP)

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

So The focus of CCP is to minimize the impact of change!

Common Closure Principle (CCP) Example

```
fileprocessing
|
|___text
|   |   TextFileReader.java
|   |   TextFileWriter.java
|
|___csv
    |   CsvFileReader.java
    |   CsvFileWriter.java
```

Common Closure Principle (Cont.)

- The Common Closure Principle (CCP) is a restatement of the Single Responsibility Principle (SRP) for components, emphasizing that a component should not have multiple reasons to change.
- Maintainability is generally more important than reusability for most applications. It is preferable for changes to be confined to a single component rather than distributed across multiple components.
- When changes are limited to one component, only that component needs to be redeployed, simplifying the release, revalidation, and redeployment process.

Common Closure Principle (Cont.)

- The CCP advises that classes likely to **change for the same reasons** should be gathered **in one place**, while tightly bound classes should belong to the same component.
- The CCP is closely related to the Open Closed Principle (OCP), which states that classes should be closed for modification but open for extension. The CCP gathers classes closed to the same types of changes in the same component.
- Both the SRP and the CCP focus on separating methods or classes that change for different reasons, contributing to a more maintainable and organized codebase.

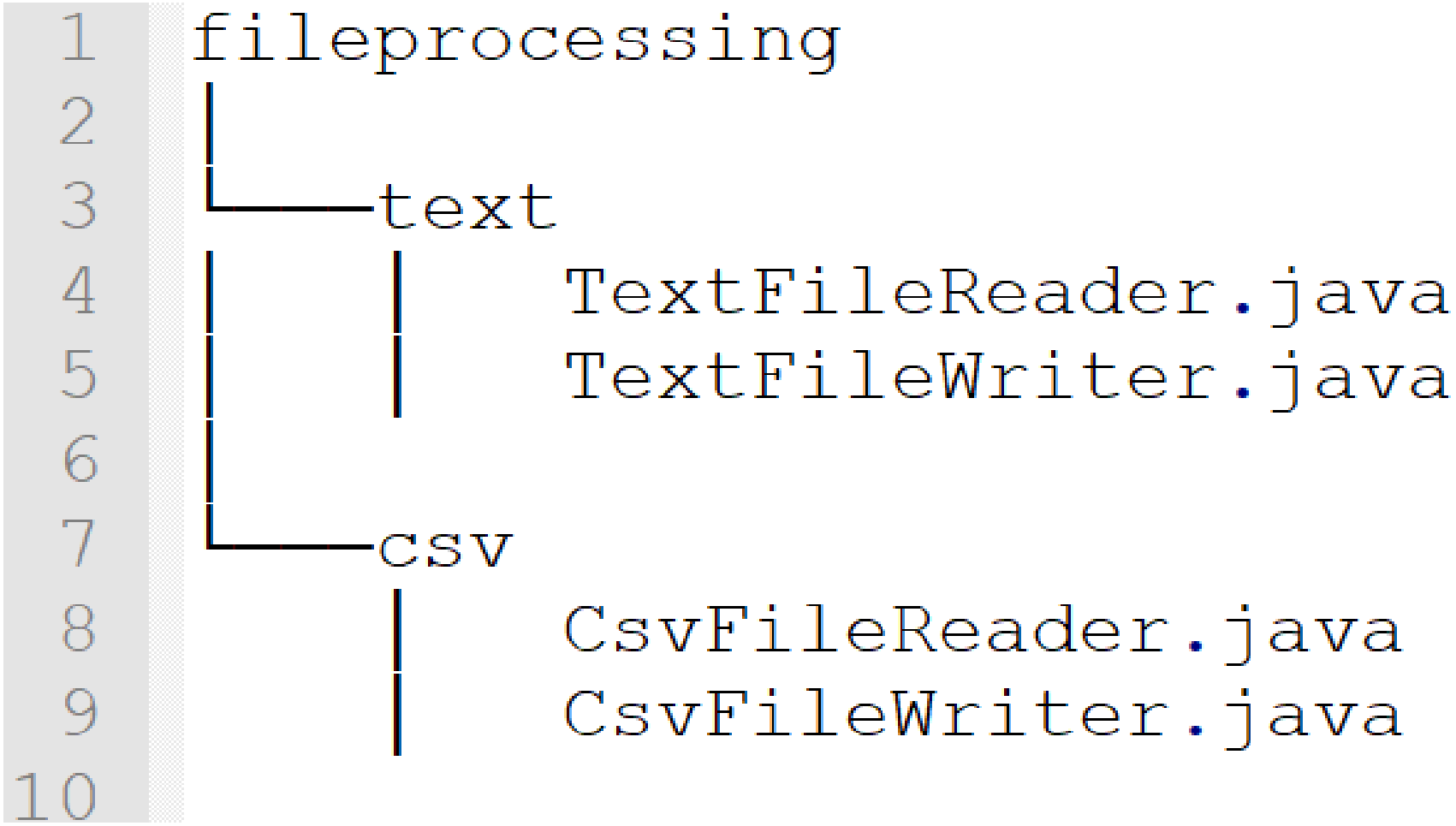
The Common Reuse Principle (Cont.)

"The classes in a component are reused together. If you reuse one of the classes in a component, you reuse them all."

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

The focus of CRP is to make it easy to reuse a package

The Common Reuse Principle (Cont.) why same example?



The Common Reuse Principle (Cont.)

- Classes and modules that are **tightly bound to each other with class relationships** and tend to be **reused together** should be placed in the same component, as they often collaborate with each other as part of a reusable abstraction.

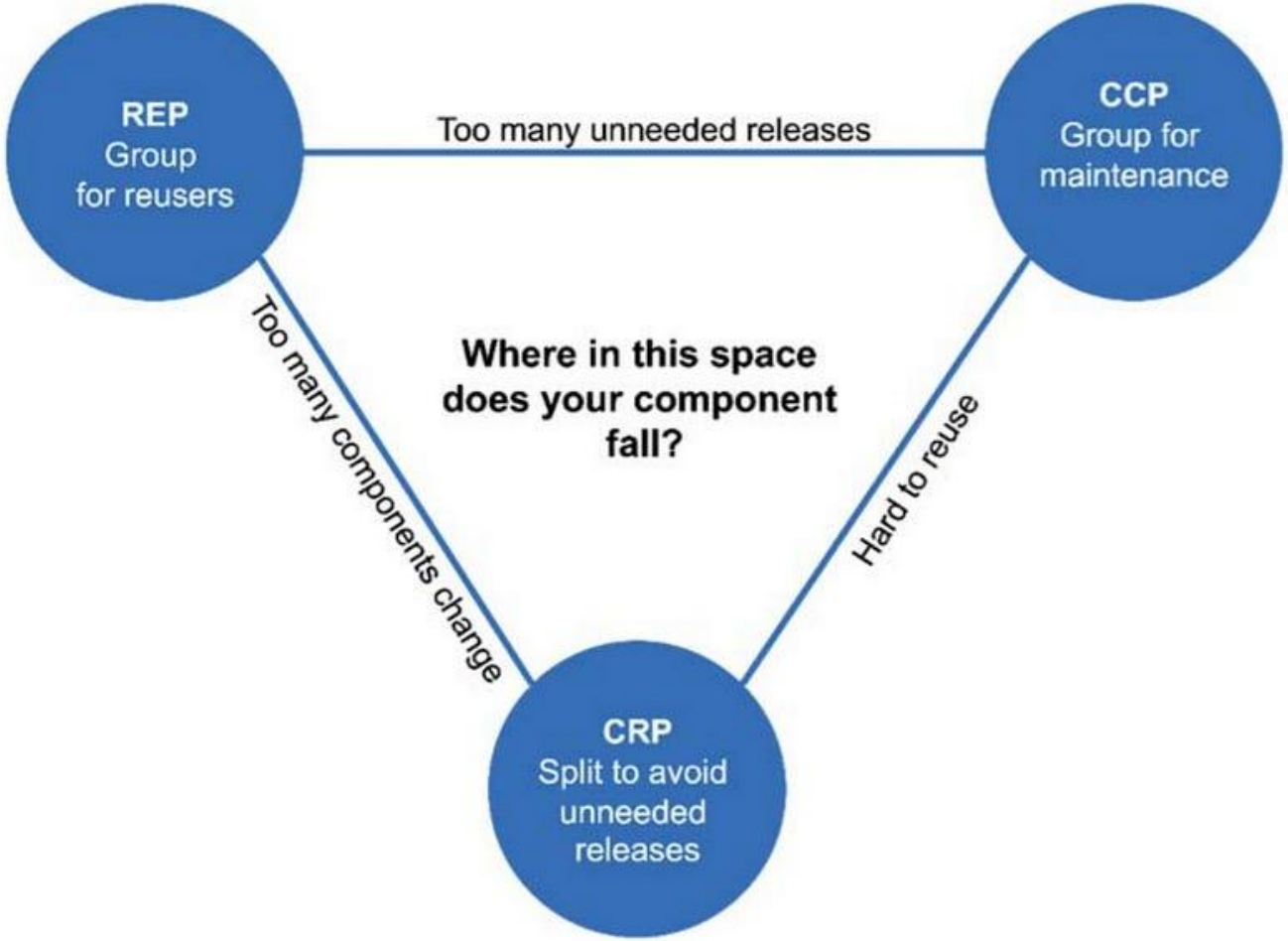
- Classes not tightly bound to each other should not be placed in the same component, helping to minimize recompilation, revalidation, and redeployment efforts.

The Common Reuse Principle (Cont.)

- In other words, Classes that are not tightly bound to each other should be separated.

- To minimize unnecessary redeployment and wasted effort, make sure that classes in a component are inseparable and that all classes within a component are depended upon.

Tension for component cohesion principles



Tension for component cohesion principles(Cont.)

- The three cohesion principles (REP, CCP, and CRP) tend to have opposing effects.
- REP tends to increase classes to the component
- CCP and CRP tends to reduce the classes from component.
- Good architects seek to resolve the tension between these principles to create a balanced component design.
- A good architect considers the current concerns of the development team and anticipates how these concerns may change over time.

Tension for component cohesion principles(Cont.)

- Early in a project's development, CCP and CRP is often more important than REP as the focus is on develop-ability rather than reuse.
- As a project matures , its position within the tension triangle shifts to more and more REP.
- The component structure of a project can vary with time and maturity, and is influenced by how the project is developed and used, rather than what the project actually does.

Coupling

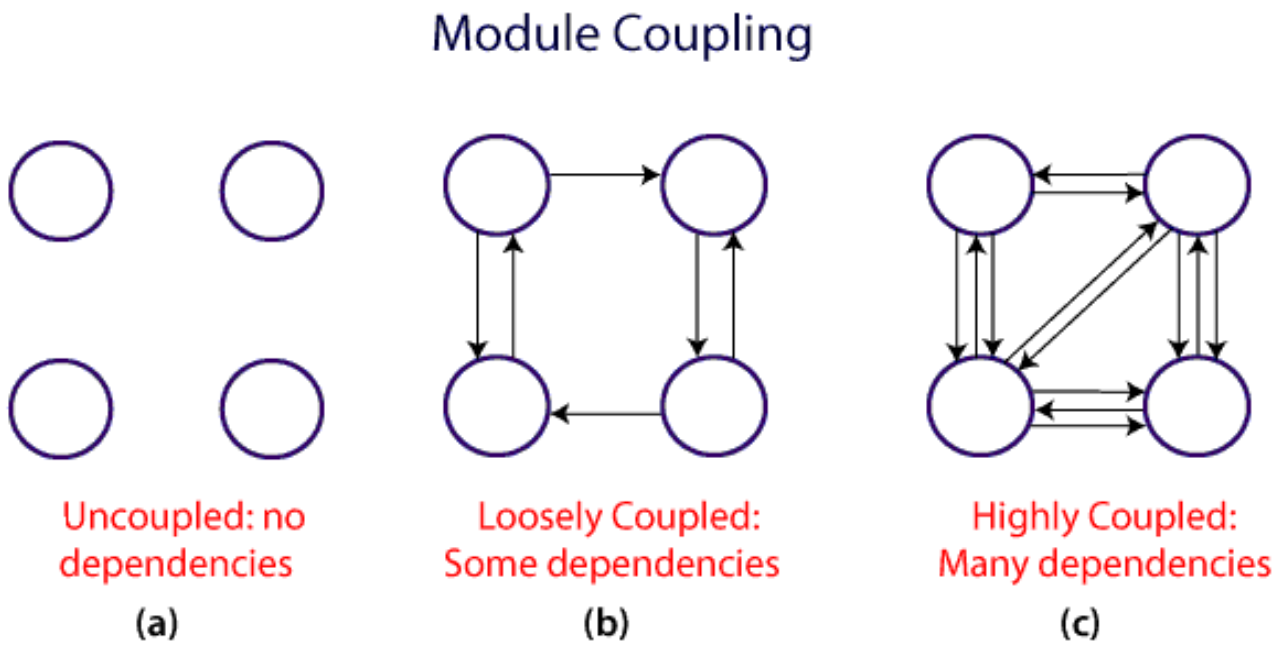
Degree of interdependence **between different modules or components** within a software system. It measures how closely connected they are and how much they rely on each other to function correctly.

(In java): it could be used to measure the interconnectedness of class to class from module to module.

Coupling(Cont.)

We always want LOW Coupling,

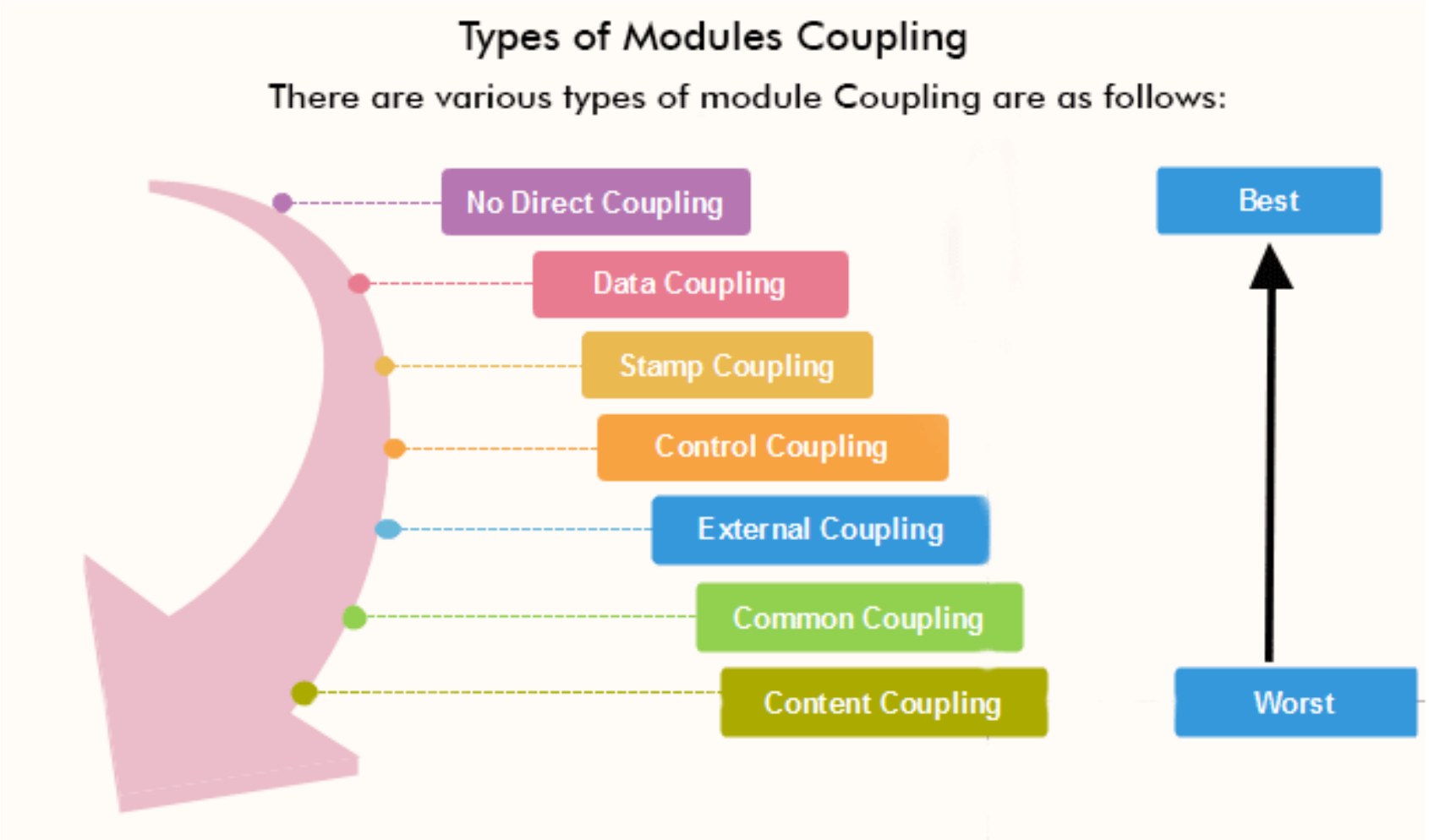
Low coupling is desirable because it makes the system more modular, easier to maintain, and less prone to errors.



Coupling(Cont.) High Coupling Cost

- **Maintainability:** High coupling complicates maintenance, as changes in one class may impact other related classes.
- **Testability:** Testing is harder due to dependencies, requiring complex mock objects or stubs for related classes.
- **Reusability:** Highly coupled code is difficult to reuse, as it depends on specific behavior and structure of other components.
- **Scalability:** Rigid, highly coupled systems are challenging to scale and may introduce bugs when adding or modifying features.
- **Understandability:** High coupling hinders code comprehension, making it harder for developers to quickly grasp the system.

Coupling - Types, from Best to the Worst



Coupling - Types, from Best to the Worst (Cont.)

- **No coupling:** Modules have no dependencies on one another; the best, but often unattainable, level of coupling. (DEMO Starts)
- **Data coupling:** Modules share data through simple parameters, but their behavior remains independent.
- **Stamp coupling:** Modules share data structures and communicate using composite data structures or objects.
- **Control coupling:** One module controls the flow or behavior of another by passing control information.

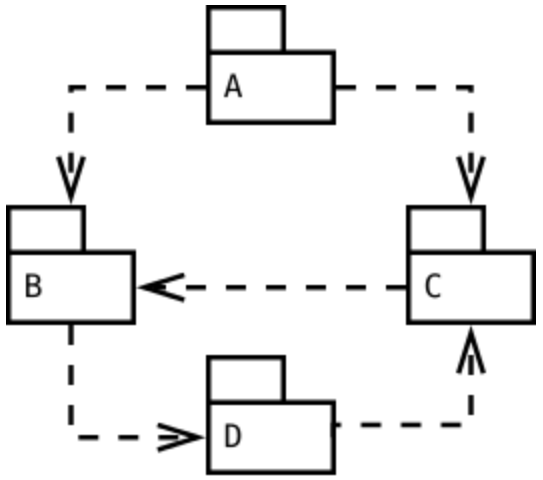
Coupling - Types, from Best to the Worst (Cont.)

- **External coupling:** Modules depend on external components, such as a shared system or environment.
- **Common coupling:** Modules share global data or state, which can lead to unwanted side effects and dependencies.
- **Content coupling:** One module directly accesses or modifies the internals of another, resulting in a high degree of interdependence and the worst form of coupling.

How to archive low coupling when creating the components?
Using Principles of Component Coupling!

- **ADP:** The Acyclic Dependencies Principle
- **SDP:** The Stable-Dependencies Principle
- **SAP:** The Stable-Abstractions Principle

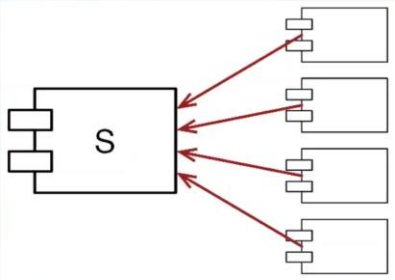
Acyclic Dependency Principle



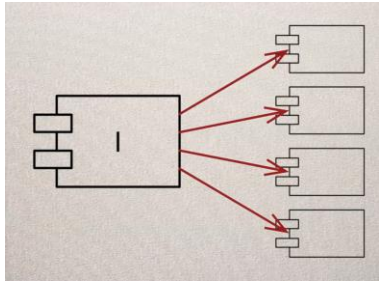
- **The dependency graph of packages should have no cycles**
- If this happens, you cannot swap or upgrade any component in the cycle
- Because we are not in control over when change will happen, or the severity of that change, we cannot allow this to happen

Stable Dependencies Principle

- **"The dependencies between software packages should be in the direction of the stability of the packages. That is, a given package should depend only on more stable packages."**
- Stability is not a boolean, it is an attribute or degree
  - Something that is hard to change is stable, something that is easy to change is unstable
  - If you are going to depend on something, you want it to be hard to change



These two diagrams show the principle in action. S would be more stable than I. S is not dependent on other packages, I is dependent on packages (that can change). Which is safer for you to depend on?



Stable Abstractions Principle

- **"A stable package should also be abstract, so that its stability does not prevent it from being extended."**
- An **unstable** package should be concrete, since its instability allows the concrete code within it to be easily changed.
- This is the open/closed principle applied to components
- By obeying the stable abstractions principle we achieve "closed" modules that are safe to depend on

Project Packaging Discussion