**Component Principles**

Table of Contents

[Component Cohesion 3](#_Toc83392811)

[The Reuse/Release Equivalence Principle 3](#_Toc83392812)

[The Common Closure Principle 3](#_Toc83392813)

[The Common Reuse Principle 4](#_Toc83392814)

[Component Coupling 5](#_Toc83392815)

[The Acyclic Dependencies Principle 5](#_Toc83392816)

[The Stable Dependencies Principle 6](#_Toc83392817)

[The Stable Abstractions Principle 8](#_Toc83392818)

Large software systems are built out of smaller **components**. The components are the units of deployment, the **smallest units** that can be deployed as a part of the system. In Java, these are .jar files. In Ruby, these are .gem files. In .Net, these are .dll files. Well-designed components always retain the ability to be **independently deployable**, and therefore, **independently developable**.

## Component Cohesion

A single component could contain **multiple classes**. These classes should all be related to each other and working to achieve the same goal, i.e. the component should be **cohesive**.

To achieve component cohesion, we can follow three principles:

1. The Reuse/Release Equivalence Principls (REP)
2. The Common Closure Principle (CCP)
3. The Common Reuse Principle (CRP)

### The Reuse/Release Equivalence Principle

The **Reuse/Release Equivalence Principle** (REP) states that classes and modules that are **grouped together** into a component should be **releasable together**. The releases should follow a **release process** and should maintain **release numbers**. Otherwise, people who wish to reuse the software component cannot and will not do so.

### The Common Closure Principle

The **Common Closure Principle** (CCP) states that classes that **change** for the **same reasons** and at the **same times** should be part of the **same component**. Otherwise, they should be in different components.

Similar to SRP, CCP essentially says that a component should not have multiple reasons to change.

### The Common Reuse Principle

The **Common Reuse Principle** (CRP) states that we should not force users of a component to depend on things they do not need. Essentially, classes and modules that tend to be **reused together** should be in the **same component**, while those that are not tightly bound to each other should be in different components.

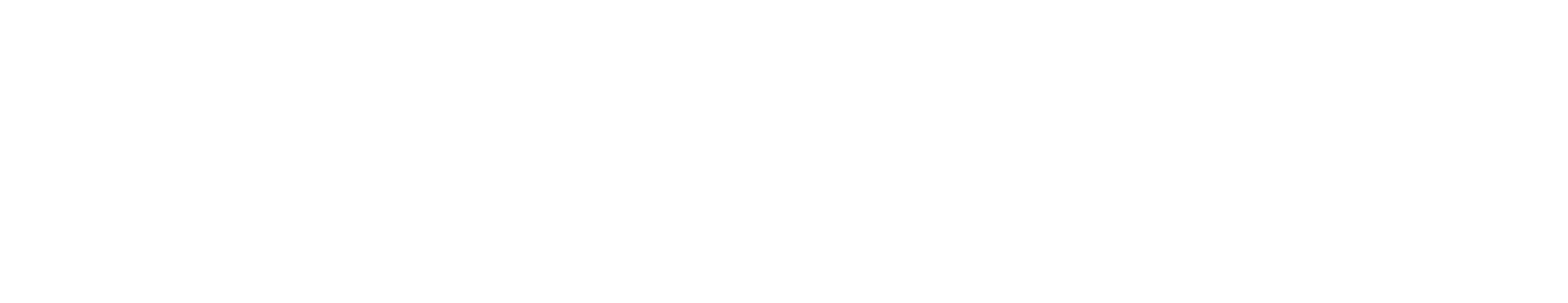
## Component Coupling

The **coupling** between components refers to how **dependent** they are on each other. There are three principles we will be looking into related to component coupling:

1. The Acyclic Dependencies Principle
2. The Stable Dependencies Principle
3. The Stable Abstractions Principle

### The Acyclic Dependencies Principle

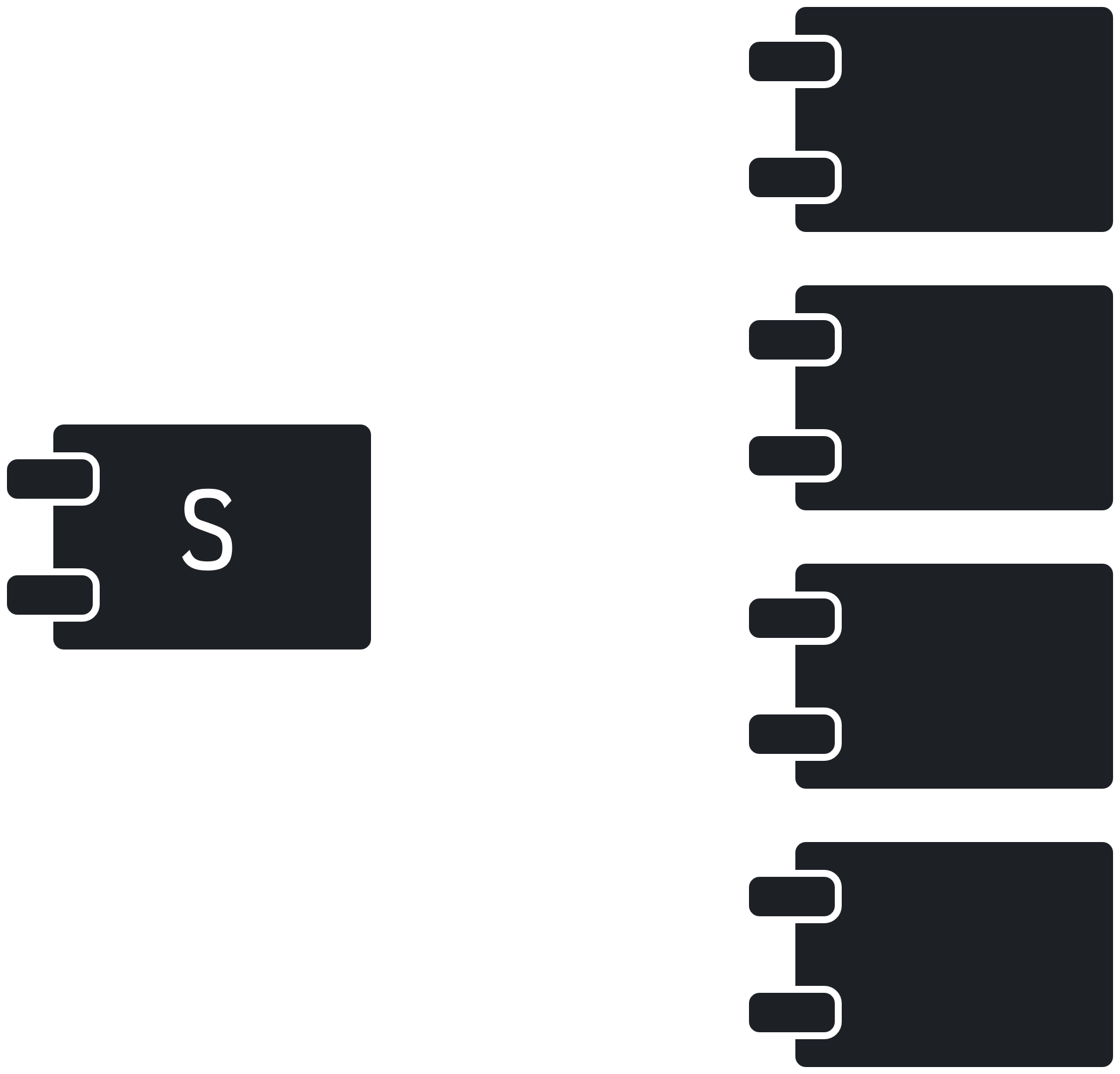
The **Acyclic Dependencies Principle** (ADP) states that we should design our components such that there are **no cycles** in the dependency graph for the components.



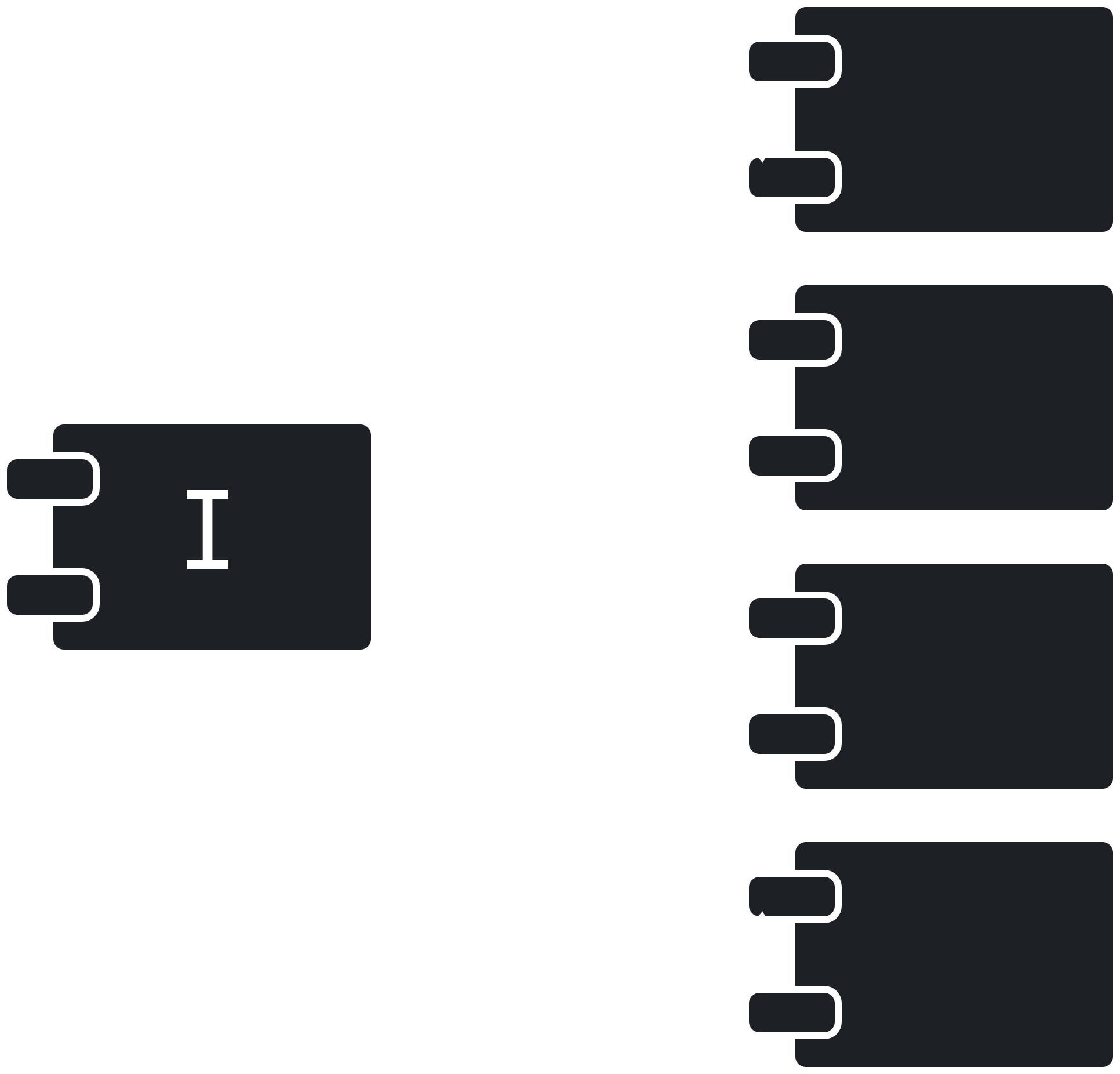
The dependency graph above shows a violation of ADP. We can fix this situation by using **interfaces**.

### The Stable Dependencies Principle

**Stability** is related to the amount of **work required** to make a **change**. **Stable** means **hard** to change. **Instable** means **easy** to change.



In the diagram above, the component S has four other components that are dependent on it. S is responsible to those components. Thus, it is very stable, since changing S would force us to change all four components. Notice that we do not relate stability to the complexity of the component.



Conversely, in the diagram above, the component I is independent and thus will have no issues changing. It is instable.

The **Stable Dependency Principle** (SDP) states that components should **depend** in the **direction of stability**. This means that less stable components should depend on more stable ones.

#### Calculating Stability

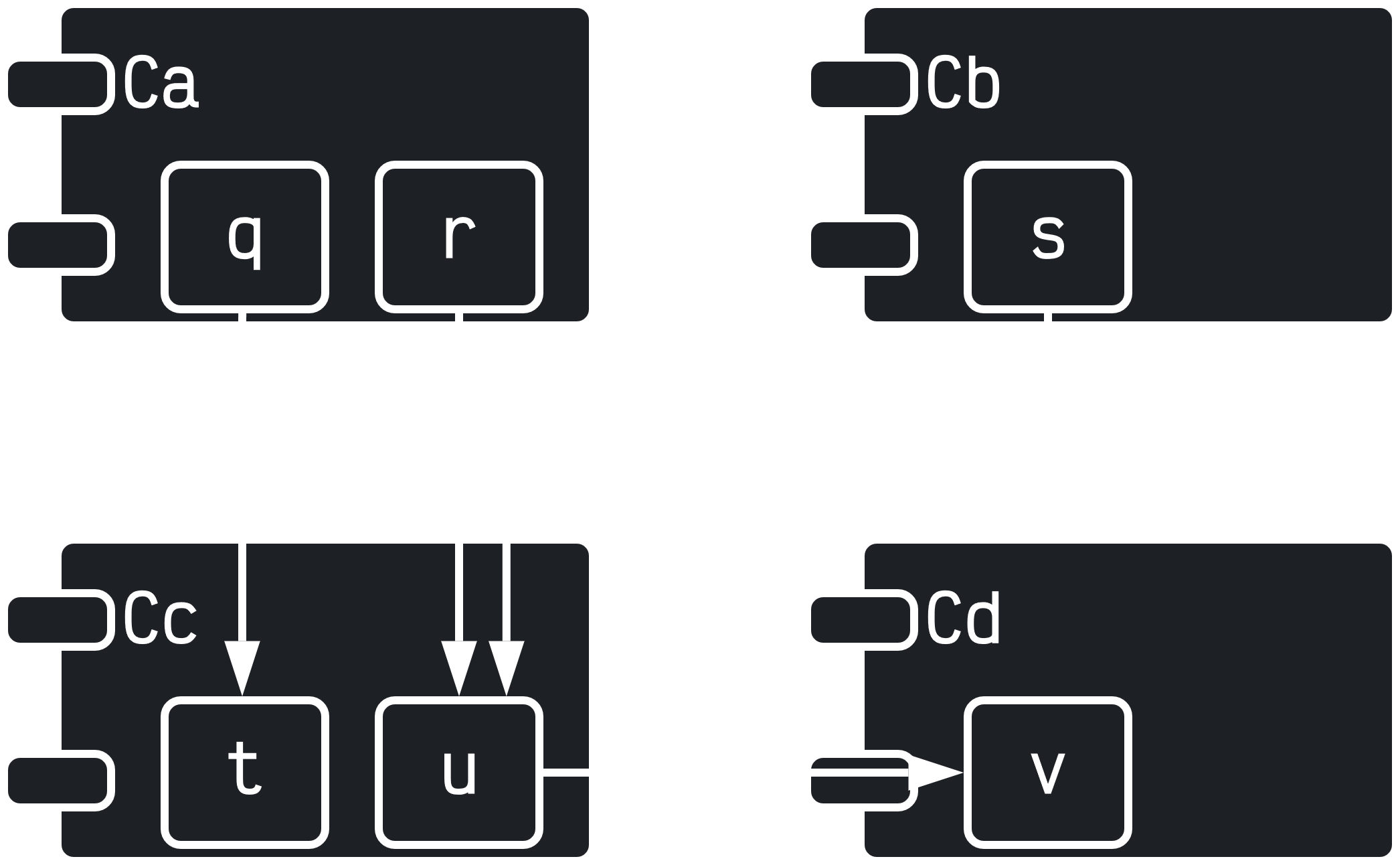
To **calculate stability**, we need to count the number of **dependencies** entering and leaving a component.

The number of **classes** **outside** the component that depend on classes **inside** the component is called the **Incoming Dependencies**, .

The number classes **inside** the component that depend on classes **outside** the component is called the **Outgoing Dependencies**, .

**Instability** is calculated as

This metric has a range from to , with being maximum **instability** and being maximum **stability**.



For the component in the diagram above,

### The Stable Abstractions Principle

The **Stable Abstraction Principle** (SAP) sets up a relationship between **stability** and **abstraction**. The more **stable** a component is, the more **abstract** it should be. If a component is to be stable, it needs to have **interfaces** and **abstract classes** so that it is extensible.

#### Measuring Abstraction

**Abstraction** is simple the ratio of **interfaces** and **abstract** classes in a component to the total number of **classes** in the component.

The metric ranges from to , with indicating that the component consists solely of abstract classes and indicating the component has no abstract classes.