# ArchUnit User Guide

Version 0.14.1

## Table of Contents

1. Introduction
    1.1. Module Overview
2. Installation
    2.1. JUnit 4
    2.2. JUnit 5
    2.3. Other Test Frameworks
    2.4. Maven Plugin
3. Getting Started
    3.1. Importing Classes
    3.2. Asserting (Architectural) Constraints
    3.3. Using JUnit 4 or JUnit 5
    3.4. Using JUnit support with Kotlin
4. What to Check
    4.1. Package Dependency Checks
    4.2. Class Dependency Checks
    4.3. Class and Package Containment Checks
    4.4. Inheritance Checks
    4.5. Annotation Checks
    4.6. Layer Checks
    4.7. Cycle Checks
5. Ideas and Concepts
    5.1. Core
    5.2. Lang
    5.3. Library
6. The Core API
    6.1. Import
    6.2. Domain
7. The Lang API
    7.1. Composing Class Rules
    7.2. Composing Member Rules
    7.3. Creating Custom Rules
    7.4. Predefined Predicates and Conditions

7.5. Rules with Custom Concepts
7.6. Controlling the Rule Text
7.7. Ignoring Violations
8. The Library API
8.1. Architectures
8.2. Slices
8.3. General Coding Rules
8.4. PlantUML Component Diagrams as rules
8.5. Freezing Arch Rules
9. JUnit Support
9.1. JUnit 4 & 5 Support
10. Advanced Configuration
10.1. Overriding configuration
10.2. Configuring the Resolution Behavior
10.3. MD5 Sums of Classes

# § 1. Introduction

ArchUnit (https://archunit.org) is a free, simple and extensible library for checking the architecture of your Java code. That is, ArchUnit can check dependencies between packages and classes, layers and slices, check for cyclic dependencies and more. It does so by analyzing given Java bytecode, importing all classes into a Java code structure. ArchUnit's main focus is to automatically test architecture and coding rules, using any plain Java unit testing framework.

## 1.1. Module Overview

ArchUnit consists of the following production modules: `archunit`, `archunit-junit4` as well as `archunit-junit5-api`, `archunit-junit5-engine` and `archunit-junit5-engine-api`. Also relevant for end users is the `archunit-example` module.

### 1.1.1. Module archunit

This module contains the actual ArchUnit core infrastructure required to write architecture tests: The `ClassFileImporter`, the domain objects, as well as the rule syntax infrastructure.

### 1.1.2. Module archunit-junit4

This module contains the infrastructure to integrate with JUnit 4, in particular the `ArchUnitRunner` to cache imported classes.

### 1.1.3. Modules archunit-junit5-*

These modules contain the infrastructure to integrate with JUnit 5 and contain the respective infrastructure to cache imported classes between test runs. `archunit-junit5-api` contains the user API to write tests with ArchUnit's JUnit 5 support, `archunit-junit5-engine` contains the runtime engine to run those tests. `archunit-junit5-engine-api` contains API code for tools that want more detailed control over running ArchUnit JUnit 5 tests, in particular a `FieldSelector` which can be used to instruct the `ArchUnitTestEngine` to run a specific rule field (compare JUnit 4 & 5 Support).

### 1.1.4. Module archunit-example

This module contains example architecture rules and sample code that violates these rules. Look here to get inspiration on how to set up rules for your project, or at ArchUnit-Examples (https://github.com/TNG/ArchUnit-Examples) for the last released version.

# 2. Installation

To use ArchUnit, it is sufficient to include the respective JAR files in the classpath. Most commonly, this is done by adding the dependency to your dependency management tool, which is illustrated for Maven and Gradle below. Alternatively you can obtain the necessary JAR files directly from Maven Central (https://search.maven.org/#search%7Cga%7C1%7Cg%3A%22com.tngtech.archunit%22).

## 2.1. JUnit 4

To use ArchUnit in combination with JUnit 4, include the following dependency from Maven Central:

*pom.xml*

```xml
<dependency>
    <groupId>com.tngtech.archunit</groupId>
    <artifactId>archunit-junit4</artifactId>
    <version>0.14.1</version>
    <scope>test</scope>
</dependency>
```

*build.gradle*

```
dependencies {
    testImplementation 'com.tngtech.archunit:archunit-junit4:0.14.1'
}
```

## 2.2. JUnit 5

ArchUnit's JUnit 5 artifacts follow the pattern of JUnit Jupiter. There is one artifact containing the API, i.e. the compile time dependencies to write tests. Then there is another artifact containing the actual `TestEngine` used at runtime. Just like JUnit Jupiter ArchUnit offers one convenience artifact transitively including both API and engine with the correct scope, which in turn can be added as a test compile dependency. Thus to include ArchUnit's JUnit 5 support, simply add the following dependency from Maven Central:

*pom.xml*

```xml
<dependency>
    <groupId>com.tngtech.archunit</groupId>
    <artifactId>archunit-junit5</artifactId>
    <version>0.14.1</version>
    <scope>test</scope>
</dependency>
```

*build.gradle*

```
dependencies {
    testImplementation 'com.tngtech.archunit:archunit-junit5:0.14.1'
}
```

## 2.3. Other Test Frameworks

ArchUnit works with any test framework that executes Java code. To use ArchUnit in such a context, include the core ArchUnit dependency from Maven Central:

*pom.xml*

```xml
<dependency>
    <groupId>com.tngtech.archunit</groupId>
    <artifactId>archunit</artifactId>
    <version>0.14.1</version>
    <scope>test</scope>
</dependency>
```

*build.gradle*

```
dependencies {
    testImplementation 'com.tngtech.archunit:archunit:0.14.1'
}
```

### 2.4. Maven Plugin

There exists a Maven plugin by Société Générale to run ArchUnit rules straight from Maven. For more information visit their GitHub repo: https://github.com/societe-generale/arch-unit-maven-plugin

## 3. Getting Started

ArchUnit tests are written the same way as any Java unit test and can be written with any Java unit testing framework. To really understand the ideas behind ArchUnit, one should consult Ideas and Concepts. The following will outline a "technical" getting started.

### 3.1. Importing Classes

At its core ArchUnit provides infrastructure to import Java bytecode into Java code structures. This can be done using the `ClassFileImporter`

```java
JavaClasses classes = new ClassFileImporter().importPackages("com.mycompany.myapp");
```

The `ClassFileImporter` offers many ways to import classes. Some ways depend on the current project's classpath, like `importPackages(..)`. However there are other ways that do not, for example:

```java
JavaClasses classes = new ClassFileImporter().importPath("/some/path");
```

The returned object of type `JavaClasses` represents a collection of elements of type `JavaClass`, where `JavaClass` in turn represents a single imported class file. You can in fact access most properties of the imported class via the public API:

```java
JavaClass clazz = classes.get(Object.class);
System.out.print(clazz.getSimpleName()); // returns 'Object'
```

### 3.2. Asserting (Architectural) Constraints

To express architectural rules, like 'Services should only be accessed by Controllers', ArchUnit offers an abstract DSL-like fluent API, which can in turn be evaluated against imported classes. To specify a rule, use the class `ArchRuleDefinition` as entry point:

```java
import static com.tngtech.archunit.lang.syntax.ArchRuleDefinition.classes;

// ...

ArchRule myRule = classes()
    .that().resideInAPackage("..service..")
    .should().onlyBeAccessed().byAnyPackage("..controller..", "..service..");
```

The two dots represent any number of packages (compare AspectJ Pointcuts). The returned object of type `ArchRule` can now be evaluated against a set of imported classes:

```java
myRule.check(importedClasses);
```

Thus the complete example could look like

```java
@Test
public void Services_should_only_be_accessed_by_Controllers() {
    JavaClasses importedClasses = new ClassFileImporter().importPackages("com.mycompany.myap

    ArchRule myRule = classes()
        .that().resideInAPackage("..service..")
        .should().onlyBeAccessed().byAnyPackage("..controller..", "..service..");

    myRule.check(importedClasses);
}
```

## 3.3. Using JUnit 4 or JUnit 5

While ArchUnit can be used with any unit testing framework, it provides extended support for writing tests with JUnit 4 and JUnit 5. The main advantage is automatic caching of imported classes between tests (of the same imported classes), as well as reduction of boilerplate code.

To use the JUnit support, declare ArchUnit’s `ArchUnitRunner` (only JUnit 4), declare the classes to import via `@AnalyzeClasses` and add the respective rules as fields:

```java
@RunWith(ArchUnitRunner.class) // Remove this line for JUnit 5!!
@AnalyzeClasses(packages = "com.mycompany.myapp")
public class MyArchitectureTest {

    @ArchTest
    public static final ArchRule myRule = classes()
        .that().resideInAPackage("..service..")
        .should().onlyBeAccessed().byAnyPackage("..controller..", "..service..");

}
```

The JUnit test support will automatically import (or reuse) the specified classes and evaluate any rule annotated with `@ArchTest` against those classes.

For further information on how to use the JUnit support refer to JUnit Support.

## 3.4. Using JUnit support with Kotlin

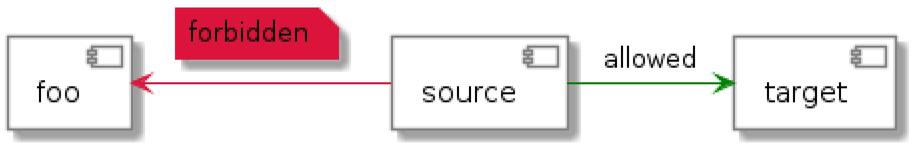
Using the JUnit support with Kotlin is quite similar to Java:

```kotlin
@RunWith(ArchUnitRunner::class) // Remove this line for JUnit 5!!
@AnalyzeClasses(packagesOf = [MyArchitectureTest::class])
class MyArchitectureTest {
    @ArchTest
    val rule_as_field = ArchRuleDefinition.noClasses().should()...

    @ArchTest
    fun rule_as_method(importedClasses: JavaClasses) {
        val rule = ArchRuleDefinition.noClasses().should()...
        rule.check(importedClasses)
    }
}
```

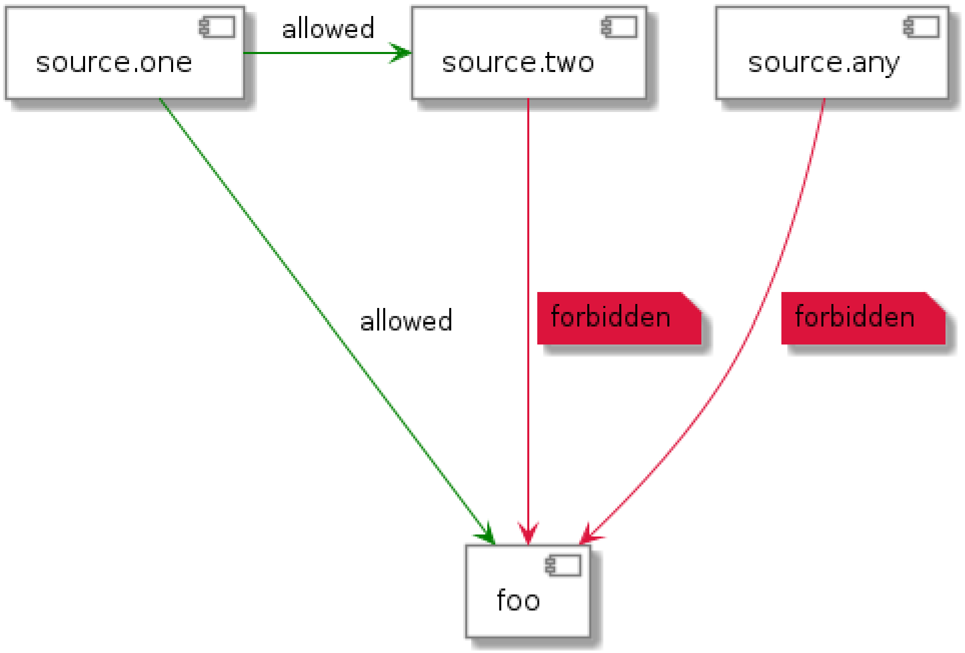
# 4. What to Check

The following section illustrates some typical checks you could do with ArchUnit.

## 4.1. Package Dependency Checks

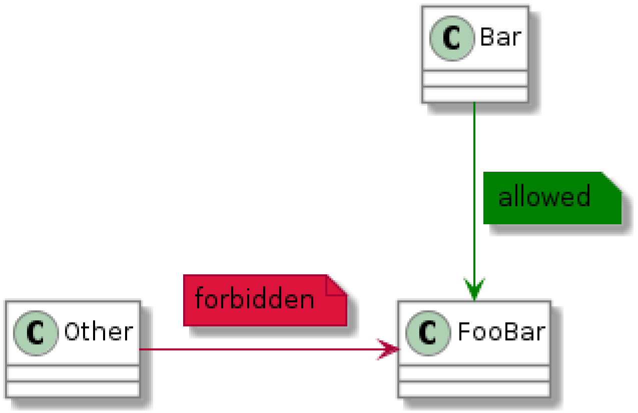


```java
noClasses().that().resideInAPackage("..source..")
    .should().dependOnClassesThat().resideInAPackage("..foo..")
```



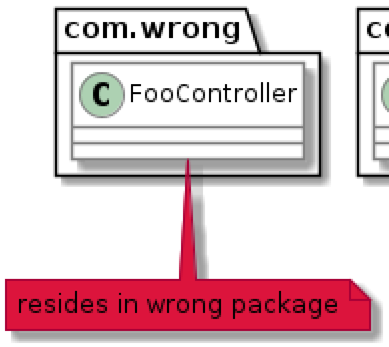
```java
classes().that().resideInAPackage("..foo..")
    .should().onlyHaveDependentClassesThat().resideInAnyPackage("..source.one..", "..foo..")
```

## 4.2. Class Dependency Checks



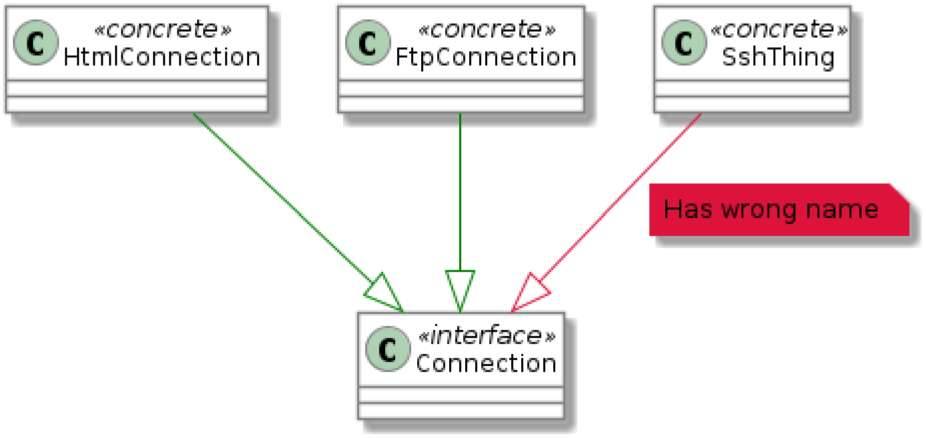
```java
classes().that().haveNameMatching(".*Bar")
    .should().onlyBeAccessed().byClassesThat().haveSimpleName("Bar")
```

## 4.3. Class and Package Containment Checks

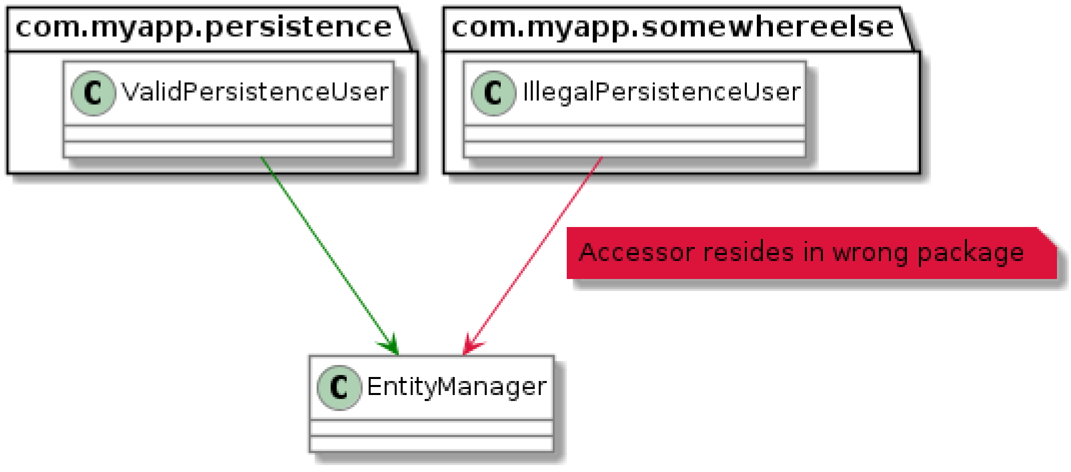


```java
classes().that().haveSimpleNameStartingWith("Foo")
    .should().resideInAPackage("com.foo")
```

## 4.4. Inheritance Checks

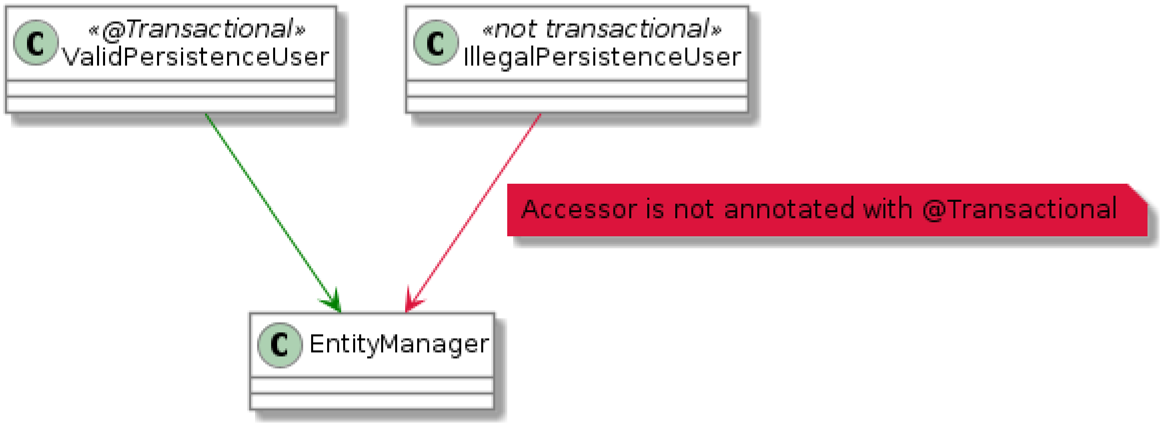


```java
classes().that().implement(Connection.class)
    .should().haveSimpleNameEndingWith("Connection")
```



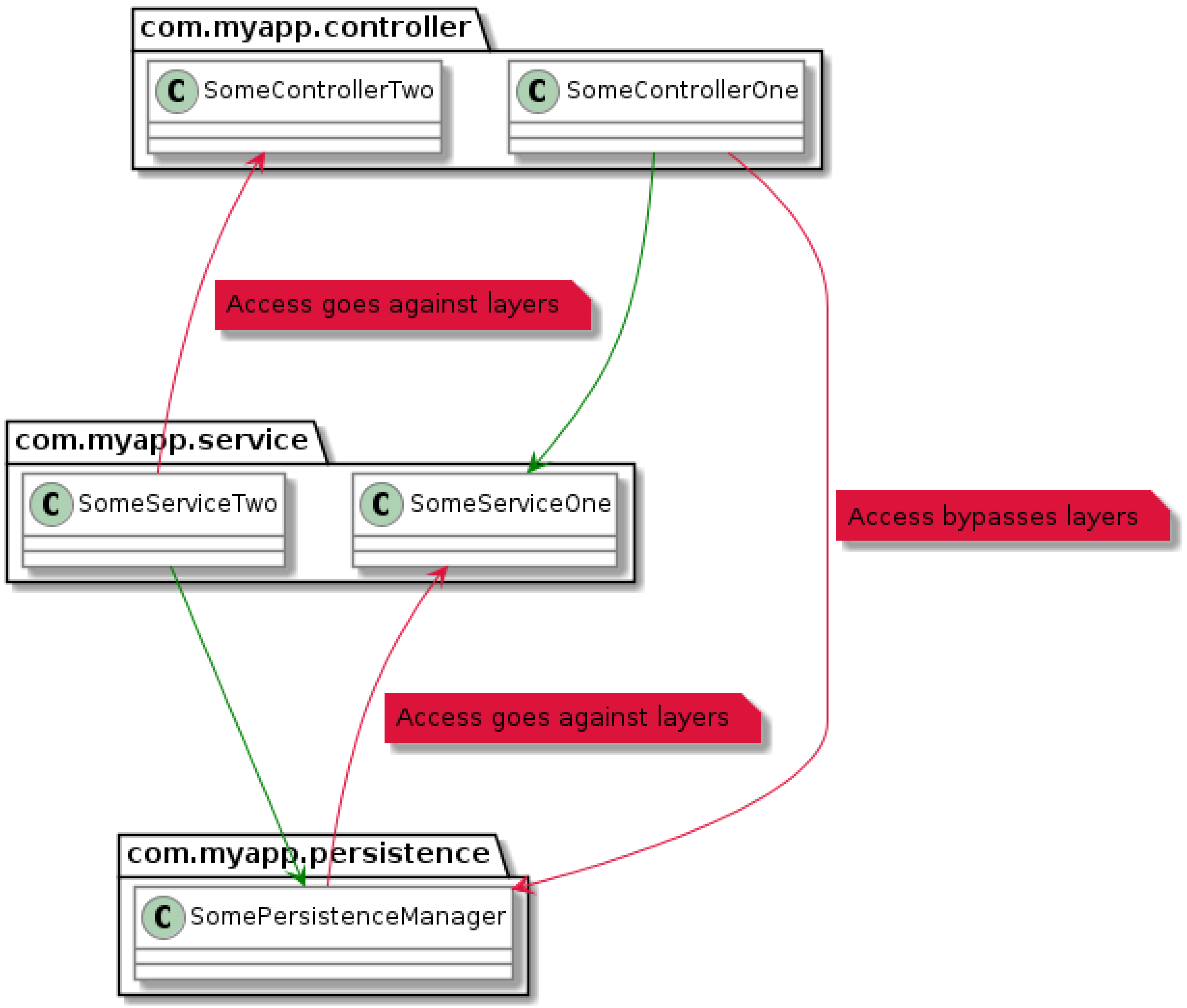
```java
classes().that().areAssignableTo(EntityManager.class)
    .should().onlyBeAccessed().byAnyPackage("..persistence..")
```

## 4.5. Annotation Checks



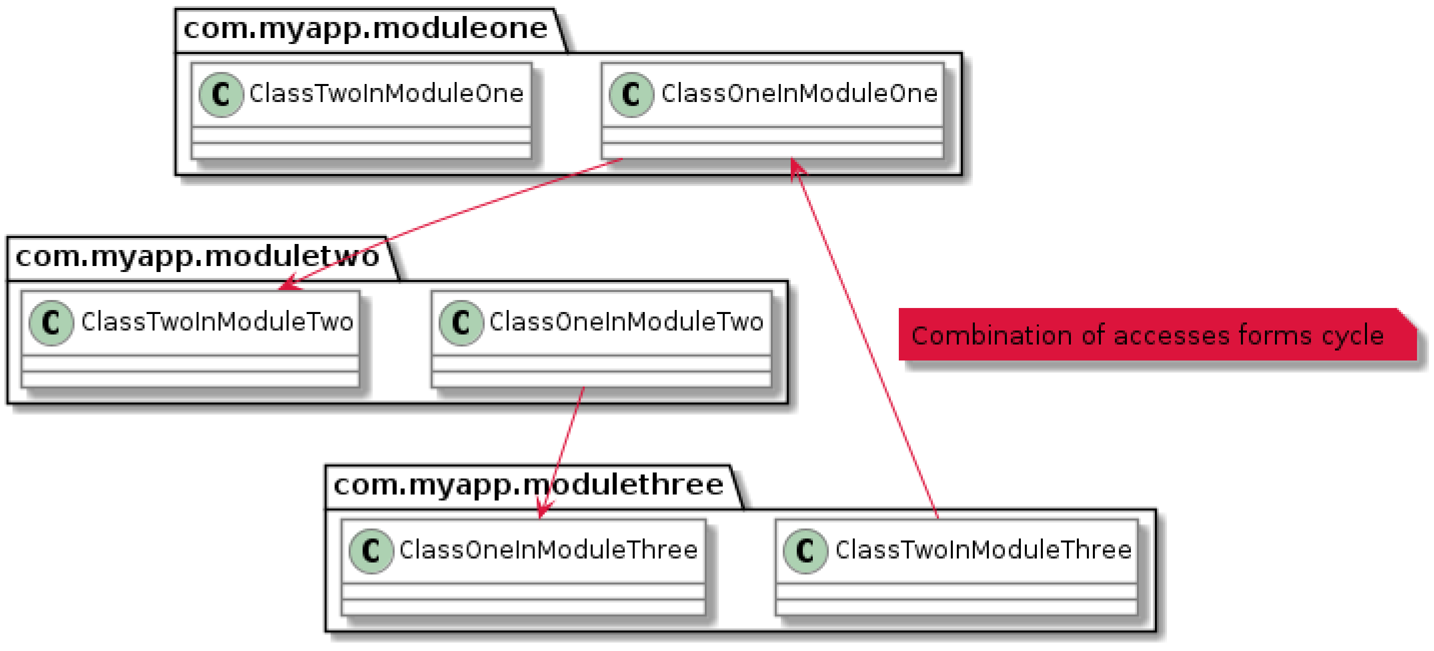
```java
classes().that().areAssignableTo(EntityManager.class)
    .should().onlyBeAccessed().byClassesThat().areAnnotatedWith(Transactional.class)
```

## 4.6. Layer Checks



```java
layeredArchitecture()
    .layer("Controller").definedBy("..controller..")
    .layer("Service").definedBy("..service..")
    .layer("Persistence").definedBy("..persistence..")

    .whereLayer("Controller").mayNotBeAccessedByAnyLayer()
    .whereLayer("Service").mayOnlyBeAccessedByLayers("Controller")
    .whereLayer("Persistence").mayOnlyBeAccessedByLayers("Service")
```

## 4.7. Cycle Checks



```java
slices().matching("com.myapp.(*)..").should().beFreeOfCycles()
```

## 5. Ideas and Concepts

ArchUnit is divided into different layers, where the most important ones are the "Core" layer, the "Lang" layer and the "Library" layer. In short the Core layer deals with the basic infrastructure, i.e. how to import byte code into Java objects. The Lang layer contains the rule syntax to specify architecture rules in a succinct way. The Library layer contains more complex predefined rules, like a layered architecture with several layers. The following section will explain these layers in more detail.

### 5.1. Core

Much of ArchUnit's core API resembles the Java Reflection API. There are classes like `JavaMethod`, `JavaField`, and more, and the public API consists of methods like `getName()`, `getMethods()`, `getRawType()` or `getRawParameterTypes()`. Additionally ArchUnit extends this API for concepts needed to talk about dependencies between code, like `JavaMethodCall`, `JavaConstructorCall` or `JavaFieldAccess`. For example, it is possible to programmatically iterate over `javaClass.getAccessesFromSelf()` and react to the imported accesses between this Java class and other Java classes.

To import compiled Java class files, ArchUnit provides the `ClassFileImporter`, which can for example be used to import packages from the classpath:

```java
JavaClasses classes = new ClassFileImporter().importPackages("com.mycompany.myapp");
```

For more information refer to The Core API.

## 5.2. Lang

The Core API is quite powerful and offers a lot of information about the static structure of a Java program. However, tests directly using the Core API lack expressiveness, in particular with respect to architectural rules.

For this reason ArchUnit provides the Lang API, which offers a powerful syntax to express rules in an abstract way. Most parts of the Lang API are composed as fluent APIs, i.e. an IDE can provide valuable suggestions on the possibilities the syntax offers.

An example for a specified architecture rule would be:

```java
ArchRule rule =
    classes().that().resideInAPackage("..service..")
        .should().onlyBeAccessed().byAnyPackage("..controller..", "..service..");
```

Once a rule is composed, imported Java classes can be checked against it:

```java
JavaClasses importedClasses = new ClassFileImporter().importPackage("com.myapp");
ArchRule rule = // define the rule
rule.check(importedClasses);
```

The syntax ArchUnit provides is fully extensible and can thus be adjusted to almost any specific need. For further information, please refer to The Lang API.

## 5.3. Library

The Library API offers predefined complex rules for typical architectural goals. For example a succinct definition of a layered architecture via package definitions. Or rules to slice the code base in a certain way, for example in different areas of the domain, and enforce these slices to be acyclic or independent of each other. More detailed information is provided in The Library API.

# 6. The Core API

The Core API is itself divided into the domain objects and the actual import.

## 6.1. Import

As mentioned in Ideas and Concepts the backbone of the infrastructure is the `ClassFileImporter`, which provides various ways to import Java classes. One way is to import packages from the classpath, or the complete classpath via

```java
JavaClasses classes = new ClassFileImporter().importClasspath();
```

However, the import process is completely independent of the classpath, so it would be well possible to import any path from the file system:

```java
JavaClasses classes = new ClassFileImporter().importPath("/some/path/to/classes");
```

The `ClassFileImporter` offers several other methods to import classes, for example locations can be specified as URLs or as JAR files.

Furthermore specific locations can be filtered out, if they are contained in the source of classes, but should not be imported. A typical use case would be to ignore test classes, when the classpath is imported. This can be achieved by specifying `ImportOptions`:

```java
ImportOption ignoreTests = new ImportOption() {
    @Override
    public boolean includes(Location location) {
        return !location.contains("/test/"); // ignore any URI to sources that contains '/te
    }
};

JavaClasses classes = new ClassFileImporter().withImportOption(ignoreTests).importClasspath(
```

A `Location` is principally an URI, i.e. ArchUnit considers sources as File or JAR URIs

- `file:///home/dev/my/project/target/classes/some/Thing.class`
- `jar:file:///home/dev/.m2/repository/some/things.jar!/some/Thing.class`

For the two common cases to skip importing JAR files and to skip importing test files (for typical setups, like a Maven or Gradle build), there already exist predefined `ImportOptions`:

```java
new ClassFileImporter()
    .withImportOption(ImportOption.Predefined.DO_NOT_INCLUDE_JARS)
    .withImportOption(ImportOption.Predefined.DO_NOT_INCLUDE_TESTS)
    .importClasspath();
```

### 6.1.1. Dealing with Missing Classes

While importing the requested classes (e.g. `target/classes` or `target/test-classes`) it can happen that a class within the scope of the import has a reference to a class outside of the scope of the import. This will naturally happen, if the classes of the JDK are not imported, since then for example any dependency on `Object.class` will be unresolved within the import.

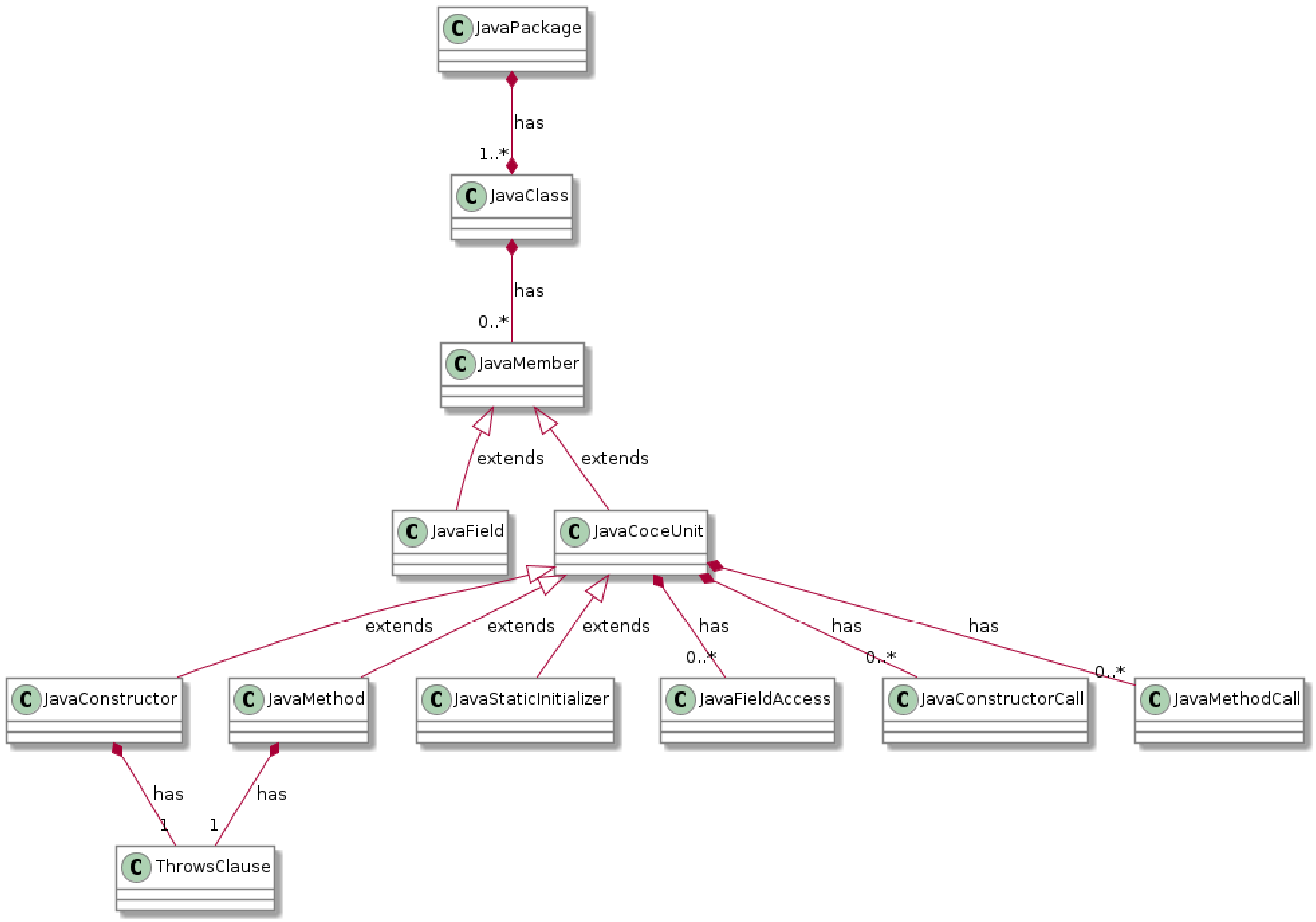
At this point ArchUnit needs to decide how to treat these classes that are missing from the import. By default, ArchUnit searches within the classpath for missing classes and if found imports them. This obviously has the advantage that information about those classes (which interfaces they implement, how they are annotated) is present during rule evaluation.

On the downside this additional lookup from the classpath will cost some performance and in some cases might not make sense (e.g. if information about classes not present in the original import is known to be unnecessary for evaluating rules). Thus ArchUnit can be configured to create stubs instead, i.e. a `JavaClass` that has all the known information, like the fully qualified name or the method called. However, this stub might naturally lack some information, like superclasses, annotations or other details that cannot be determined without importing the bytecode of this class. This behavior will also happen, if ArchUnit fails to determine the location of a missing class from the classpath.

To find out, how to configure the default behavior, refer to Configuring the Resolution Behavior.

## 6.2. Domain

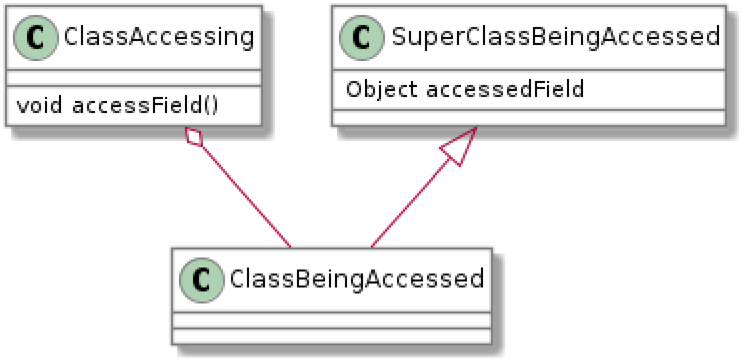
The domain objects represent Java code, thus the naming should be pretty straight forward. Most commonly, the `ClassFileImporter` imports instances of type `JavaClass`. A rough overview looks like this:



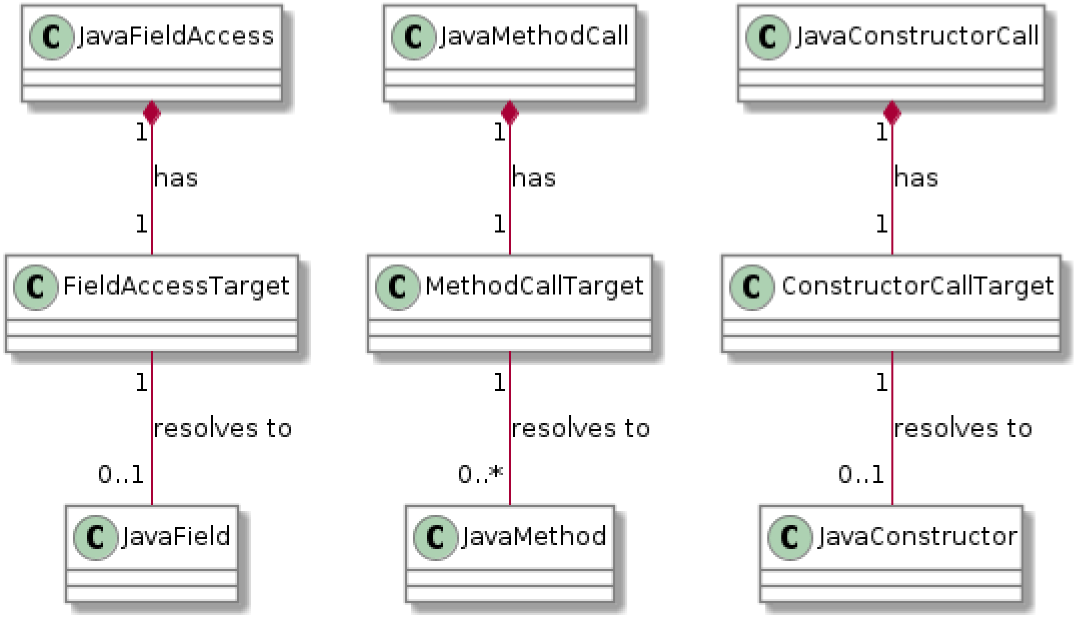
Most objects resemble the Java Reflection API, including inheritance relations. Thus a `JavaClass` has `JavaMembers`, which can in turn be either `JavaField`, `JavaMethod`, `JavaConstructor` (or `JavaStaticInitializer`). While not present within the reflection API, it makes sense to introduce an expression for anything that can access other code, which ArchUnit calls 'code unit', and is in fact either a method, a constructor (including the class initializer) or a static initializer of a class (e.g. a `static { … }` block, a static field assignment, etc.).

Furthermore one of the most interesting features of ArchUnit that exceeds the Java Reflection API, is the concept of accesses to another class. On the lowest level accesses can only take place from a code unit (as mentioned, any block of executable code) to either a field (`JavaFieldAccess`), a method (`JavaMethodCall`) or constructor (`JavaConstructorCall`).

ArchUnit imports the whole graph of classes and their relationship to each other. While checking the accesses **from** a class is pretty isolated (the bytecode offers all this information), checking accesses **to** a class requires the whole graph to be built first. To distinguish which sort of access is referred to, methods will always clearly state **fromSelf** and **toSelf**. For example, every `JavaField` allows to call `JavaField#getAccessesToSelf()` to retrieve all code units within the graph that access this specific field. The resolution process through inheritance is not completely straight forward. Consider for example



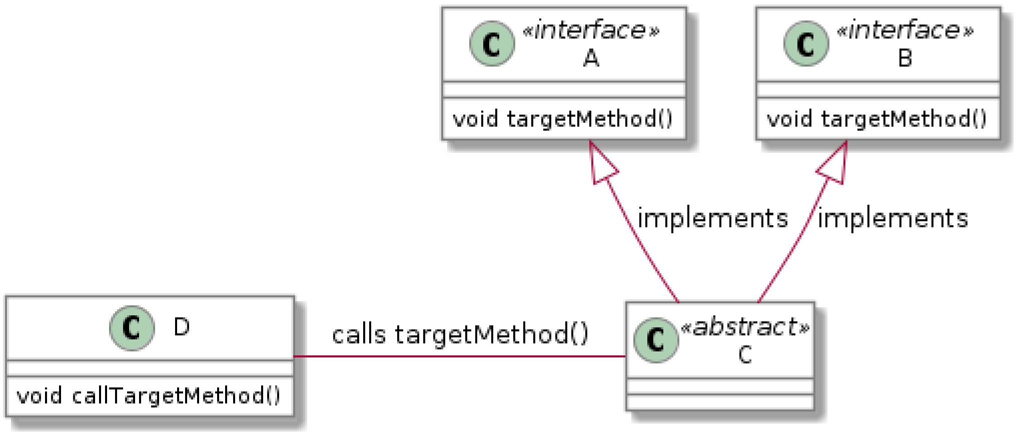
The bytecode will record a field access from `ClassAccessing.accessField()` to `ClassBeingAccessed.accessedField`. However, there is no such field, since the field is actually declared in the superclass. This is the reason why a `JavaFieldAccess` has no `JavaField` as its target, but a `FieldAccessTarget`. In other words, ArchUnit models the situation, as it is found within the bytecode, and an access target is not an actual member within another class. If a member is queried for `accessesToSelf()` though, ArchUnit will resolve the necessary targets and determine, which member is represented by which target. The situation looks roughly like



Two things might seem strange at the first look.

First, why can a target resolve to zero matching members? The reason is that the set of classes that was imported does not need to have all classes involved within this resolution process. Consider the above example, if `SuperClassBeingAccessed` would not be imported, ArchUnit would have no way of knowing where the actual targeted field resides. Thus in this case the resolution would return zero elements.

Second, why can there be more than one resolved methods for method calls? The reason for this is that a call target might indeed match several methods in those cases, for example:



While this situation will always be resolved in a specified way for a real program, ArchUnit cannot do the same. Instead, the resolution will report all candidates that match a specific access target, so in the above example, the call target `C.targetMethod()` would in fact resolve to two `JavaMethods`, namely `A.targetMethod()` and `B.targetMethod()`. Likewise a check of either `A.targetMethod.getCallsToSelf()` or `B.targetMethod.getCallsToSelf()` would return the same call from `D.callTargetMethod()` to `C.targetMethod()`.

### 6.2.1. Domain Objects, Reflection and the Classpath

ArchUnit tries to offer a lot of information from the bytecode. For example, a `JavaClass` provides details like if it is an enum or an interface, modifiers like `public` or `abstract`, but also the source, where this class was imported from (namely the URI mentioned in the first section). However, if information if missing, and the classpath is correct, ArchUnit offers some convenience to rely on the reflection API for extended details. For this reason, most `Java*` objects offer a method `reflect()`, which will in fact try to resolve the respective object from the Reflection API. For example:

```java
JavaClasses classes = new ClassFileImporter().importClasspath(new ImportOptions());

// ArchUnit's java.lang.String
JavaClass javaClass = classes.get(String.class);
// Reflection API's java.lang.String
Class<?> stringClass = javaClass.reflect();

// ArchUnit's public int java.lang.String.length()
JavaMethod javaMethod = javaClass.getMethod("length");
// Reflection API's public int java.lang.String.length()
Method lengthMethod = javaMethod.reflect();
```

However, this will throw an `Exception`, if the respective classes are missing on the classpath (e.g. because they were just imported from some file path).

This restriction also applies to handling annotations in a more convenient way. Consider the following annotation:

```java
@interface CustomAnnotation {
    String value();
}
```

If you need to access this annotation without it being on the classpath, you must rely on

```java
JavaAnnotation<?> annotation = javaClass.getAnnotationOfType("some.pkg.CustomAnnotation");
// result is untyped, since it might not be on the classpath (e.g. enums)
Object value = annotation.get("value");
```

So there is neither type safety nor automatic refactoring support. If this annotation is on the classpath, however, this can be written way more naturally:

```java
CustomAnnotation annotation = javaClass.getAnnotationOfType(CustomAnnotation.class);
String value = annotation.value();
```

ArchUnit's own rule APIs (compare The Lang API) never rely on the classpath though. Thus the evaluation of default rules and syntax combinations, described in the next section, does not depend on whether the classes were imported from the classpath or some JAR / folder.

# 7. The Lang API

## 7.1. Composing Class Rules

The Core API is pretty powerful with regard to all the details from the bytecode that it provides to tests. However, tests written this way lack conciseness and fail to convey the architectural concept that they should assert. Consider:

JAVA

```java
Set<JavaClass> services = new HashSet<>();
for (JavaClass clazz : classes) {
    // choose those classes with FQN with infix '.service.'
    if (clazz.getName().contains(".service.")) {
        services.add(clazz);
    }
}

for (JavaClass service : services) {
    for (JavaAccess<?> access : service.getAccessesFromSelf()) {
        String targetName = access.getTargetOwner().getName();

        // fail if the target FQN has the infix ".controller."
        if (targetName.contains(".controller.")) {
            String message = String.format(
                    "Service %s accesses Controller %s in line %d",
                    service.getName(), targetName, access.getLineNumber());
            Assert.fail(message);
        }
    }
}
```

What we want to express, is the rule "*no classes that reside in a package 'service' should access classes that reside in a package 'controller'*". Nevertheless, it's hard to read through that code and distill that information. And the same process has to be done every time someone needs to understand the semantics of this rule.

To solve this shortcoming, ArchUnit offers a high level API to express architectural concepts in a concise way. In fact, we can write code that is almost equivalent to the prose rule text mentioned before:

JAVA

```java
ArchRule rule = ArchRuleDefinition.noClasses()
    .that().resideInAPackage("..service..")
    .should().accessClassesThat().resideInAPackage("..controller..");

rule.check(importedClasses);
```

The only difference to colloquial language is the ".." in the package notation, which refers to any number of packages. Thus "..service.." just expresses "*any package that contains some sub-package 'service'*", e.g. `com.myapp.service.any`. If this test fails, it will report an `AssertionError` with the following message:

BASH

```bash
java.lang.AssertionError: Architecture Violation [Priority: MEDIUM] -
Rule 'no classes that reside in a package '..service..'
should access classes that reside in a package '..controller..'' was violated (1 times):
Method <some.pkg.service.SomeService.callController()>
calls method <some.pkg.controller.SomeController.execute()>
in (SomeService.java:14)
```

So as a benefit, the assertion error contains the full rule text out of the box and reports all violations including the exact class and line number. The rule API also allows to combine predicates and conditions:

```java
noClasses()
    .that().resideInAPackage("..service..")
    .or().resideInAPackage("..persistence..")
    .should().accessClassesThat().resideInAPackage("..controller..")
    .orShould().accessClassesThat().resideInAPackage("..ui..")

rule.check(importedClasses);
```

## 7.2. Composing Member Rules

In addition to a predefined API to write rules about Java classes and their relations, there is an extended API to define rules for members of Java classes. This might be relevant, for example, if methods in a certain context need to be annotated with a specific annotation, or return types implementing a certain interface. The entry point is again `ArchRuleDefinition`, e.g.

```java
ArchRule rule = ArchRuleDefinition.methods()
    .that().arePublic()
    .and().areDeclaredInClassesThat().resideInAPackage("..controller..")
    .should().beAnnotatedWith(Secured.class);

rule.check(importedClasses);
```

Besides `methods()`, `ArchRuleDefinition` offers the methods `members()`, `fields()`, `codeUnits()`, `constructors()` – and the corresponding negations `noMembers()`, `noFields()`, `noMethods()`, etc.

## 7.3. Creating Custom Rules

In fact, most architectural rules take the form

```
classes that ${PREDICATE} should ${CONDITION}
```

In other words, we always want to limit imported classes to a relevant subset, and then evaluate some condition to see that all those classes satisfy it. ArchUnit's API allows you to do just that, by exposing the concepts of `DescribedPredicate` and `ArchCondition`. So the rule above is just an application of this generic API:

```java
DescribedPredicate<JavaClass> resideInAPackageService = // define the predicate
ArchCondition<JavaClass> accessClassesThatResideInAPackageController = // define the conditi

noClasses().that(resideInAPackageService)
    .should(accessClassesThatResideInAPackageController);
```

Thus, if the predefined API does not allow to express some concept, it is possible to extend it in any custom way. For example:

```java
DescribedPredicate<JavaClass> haveAFieldAnnotatedWithPayload =
    new DescribedPredicate<JavaClass>("have a field annotated with @Payload"){
        @Override
        public boolean apply(JavaClass input) {
            boolean someFieldAnnotatedWithPayload = // iterate fields and check for @Payload
            return someFieldAnnotatedWithPayload;
        }
    };

ArchCondition<JavaClass> onlyBeAccessedBySecuredMethods =
    new ArchCondition<JavaClass>("only be accessed by @Secured methods") {
        @Override
        public void check(JavaClass item, ConditionEvents events) {
            for (JavaMethodCall call : item.getMethodCallsToSelf()) {
                if (!call.getOrigin().isAnnotatedWith(Secured.class)) {
                    String message = String.format(
                        "Method %s is not @Secured", call.getOrigin().getFullName());
                    events.add(SimpleConditionEvent.violated(call, message));
                }
            }
        }
    };

classes().that(haveAFieldAnnotatedWithPayload).should(onlyBeAccessedBySecuredMethods);
```

If the rule fails, the error message will be built from the supplied descriptions. In the example above, it would be

```
classes that have a field annotated with @Payload should only be accessed by @Secured method
```

## 7.4. Predefined Predicates and Conditions

Custom predicates and conditions like in the last section can often be composed from predefined elements. ArchUnit's basic convention for predicates is that they are defined in an inner class `Predicates` within the type they target. For example, one can find the predicate to check for the simple name of a `JavaClass` as

```java
JavaClass.Predicates.simpleName(String)
```

Predicates can be joined using the methods `predicate.or(other)` and `predicate.and(other)`. So for example a predicate testing for a class with simple name "Foo" that is serializable could be created the following way:

```java
import static com.tngtech.archunit.core.domain.JavaClass.Predicates.assignableTo;
import static com.tngtech.archunit.core.domain.JavaClass.Predicates.simpleName;

DescribedPredicate<JavaClass> serializableNamedFoo =
    simpleName("Foo").and(assignableTo(Serializable.class));
```

Note that for some properties, there exist interfaces with predicates defined for them. For example the property to have a name is represented by the interface `HasName`; consequently the predicate to check the name of a `JavaClass` is the same as the predicate to check the name of a `JavaMethod`, and resides within

```java
HasName.Predicates.name(String)
```

This can at times lead to problems with the type system, if predicates are supposed to be joined. Since the `or(..)` method accepts a type of `DescribedPredicate<? super T>`, where `T` is the type of the first predicate. For example:

```java
// Does not compile, because type(..) targets a subtype of HasName
HasName.Predicates.name("").and(JavaClass.Predicates.type(Serializable.class))

// Does compile, because name(..) targets a supertype of JavaClass
JavaClass.Predicates.type(Serializable.class).and(HasName.Predicates.name(""))

// Does compile, because the compiler now sees name(..) as a predicate for JavaClass
DescribedPredicate<JavaClass> name = HasName.Predicates.name("").forSubType();
name.and(JavaClass.Predicates.type(Serializable.class));
```

This behavior is somewhat tedious, but unfortunately it is a shortcoming of the Java type system that cannot be circumvented in a satisfying way.

Just like predicates, there exist predefined conditions that can be combined in a similar way. Since `ArchCondition` is a less generic concept, all predefined conditions can be found within `ArchConditions`. Examples:

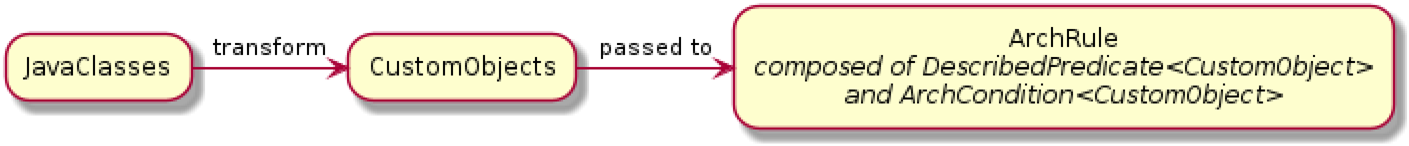
```java
ArchCondition<JavaClass> callEquals =
    ArchConditions.callMethod(Object.class, "equals", Object.class);
ArchCondition<JavaClass> callHashCode =
    ArchConditions.callMethod(Object.class, "hashCode");

ArchCondition<JavaClass> callEqualsOrHashCode = callEquals.or(callHashCode);
```

## 7.5. Rules with Custom Concepts

Earlier we stated that most architectural rules take the form

```
classes that ${PREDICATE} should ${CONDITION}
```

However, we do not always talk about classes, if we express architectural concepts. We might have custom language, we might talk about modules, about slices, or on the other hand more detailed about fields, methods or constructors. A generic API will never be able to support every imaginable concept out of the box. Thus ArchUnit's rule API has at its foundation a more generic API that controls the types of objects that our concept targets.


JavaClasses
transform
CustomObjects
passed to
ArchRule
composed of DescribedPredicate<CustomObject>
and ArchCondition<CustomObject>

To achieve this, any rule definition is based on a `ClassesTransformer` that defines how `JavaClasses` are to be transformed to the desired rule input. In many cases, like the ones mentioned in the sections above, this is the identity transformation, passing classes on to the rule as they are. However, one can supply any custom transformation to express a rule about a different type of input object. For example:

```java
ClassesTransformer<JavaPackage> packages = new AbstractClassesTransformer<JavaPackage>("pack
    @Override
    public Iterable<JavaPackage> doTransform(JavaClasses classes) {
        Set<JavaPackage> result = new HashSet<>();
        classes.getDefaultPackage().accept(alwaysTrue(), new PackageVisitor() {
            @Override
            public void visit(JavaPackage javaPackage) {
                result.add(javaPackage);
            }
        });
        return result;
    }
};

all(packages).that(containACoreClass()).should(...);
```

Of course these transformers can represent any custom concept desired:

```java
// how we map classes to business modules
ClassesTransformer<BusinessModule> businessModules = ...

// filter business module dealing with orders
DescribedPredicate<BusinessModule> dealWithOrders = ...

// check that the actual business module is independent of payment
ArchCondition<BusinessModule> beIndependentOfPayment = ...

all(businessModules).that(dealWithOrders).should(beIndependentOfPayment);
```

## 7.6. Controlling the Rule Text

If the rule is straight forward, the rule text that is created automatically should be sufficient in many cases. However, for rules that are not common knowledge, it is good practice to document the reason for this rule. This can be done in the following way:

```java
classes().that(haveAFieldAnnotatedWithPayload).should(onlyBeAccessedBySecuredMethods)
    .because("@Secured methods will be intercepted, checking for increased privileges " +
        "and obfuscating sensitive auditing information");
```

Nevertheless, the generated rule text might sometimes not convey the real intention concisely enough, e.g. if multiple predicates or conditions are joined. It is possible to completely overwrite the rule description in those cases:

```java
classes().that(haveAFieldAnnotatedWithPayload).should(onlyBeAccessedBySecuredMethods)
    .as("Payload may only be accessed in a secure way");
```

## 7.7. Ignoring Violations

In legacy projects there might be too many violations to fix at once. Nevertheless, that code should be covered completely by architecture tests to ensure that no further violations will be added to the existing code. One approach to ignore existing violations is to tailor the `that(..)` clause of the rules in question to ignore certain violations. A more generic approach is to ignore violations based on simple regex matches. For this one can put a file named `archunit_ignore_patterns.txt` in the root of the classpath. Every line will be interpreted as a regular expression and checked against reported violations. Violations with a message matching the pattern will be ignored. If no violations are left, the check will pass.

For example, suppose the class `some.pkg.LegacyService` violates a lot of different rules. It is possible to add *archunit_ignore_patterns.txt*

```bash
.*some\.pkg\.LegacyService.*
```

All violations mentioning `some.pkg.LegacyService` will consequently be ignored, and rules that are only violated by such violations will report success instead of failure.

It is possible to add comments to ignore patterns by prefixing the line with a '#':

*archunit_ignore_patterns.txt*

```bash
# There are many known violations where LegacyService is involved; we'll ignore them all
.*some\.pkg\.LegacyService.*
```

# 8. The Library API

The Library API offers a growing collection of predefined rules, which offer a more concise API for more complex but common patterns, like a layered architecture or checks for cycles between slices (compare What to Check).

## 8.1. Architectures

The entrance point for checks of common architectural styles is:

```java
com.tngtech.archunit.library.Architectures
```

At the moment this only provides a convenient check for a layered architecture and onion architecture. But in the future it might be extended for styles like a pipes and filters, separation of business logic and technical infrastructure, etc.

### 8.1.1. Layered Architecture

In layered architectures, we define different layers and how those interact with each other. An example setup for a simple 3-tier architecture can be found in Layer Checks.

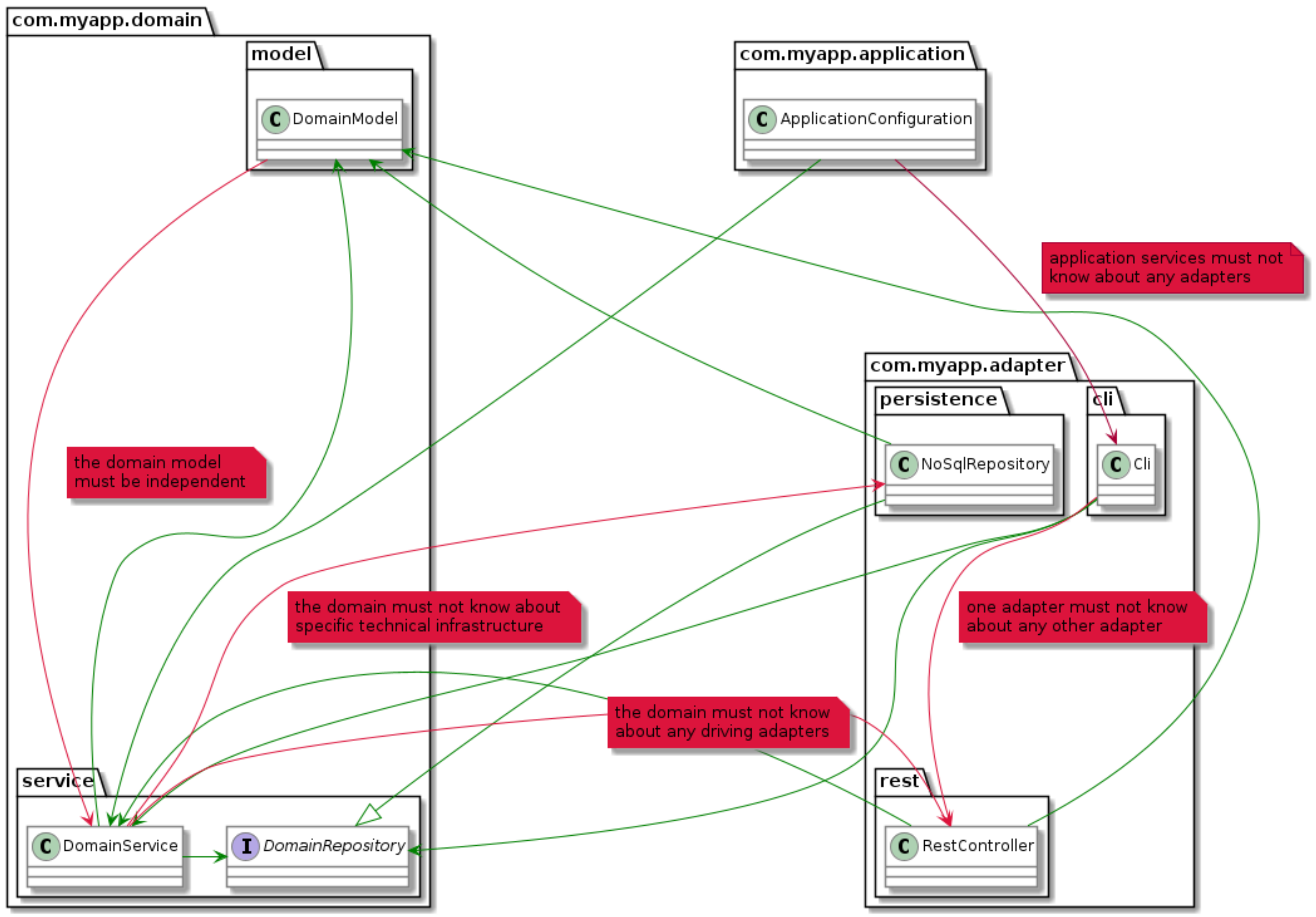
### 8.1.2. Onion Architecture

In an "Onion Architecture" (also known as "Hexagonal Architecture" or "Ports and Adapters"), we can define domain packages and adapter packages as follows.

```java
onionArchitecture()
        .domainModels("com.myapp.domain.model..")
        .domainServices("com.myapp.domain.service..")
        .applicationServices("com.myapp.application..")
        .adapter("cli", "com.myapp.adapter.cli..")
        .adapter("persistence", "com.myapp.adapter.persistence..")
        .adapter("rest", "com.myapp.adapter.rest..");
```

The semantic follows the descriptions in https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/. More precisely, the following holds:

- The `domain` package is the core of the application. It consists of two parts.
  1. The `domainModels` packages contain the domain entities.
  2. The packages in `domainServices` contains services that use the entities in the `domainModel` packages.
- The `applicationServices` packages contain services and configuration to run the application and use cases. It can use the items of the `domain` package but there must not be any dependency from the `domain` to the `application` packages.
- The `adapter` package contains logic to connect to external systems and/or infrastructure. No adapter may depend on another adapter. Adapters can use both the items of the `domain` as well as the `application` packages. Vice versa, neither the `domain` nor the `application` packages must contain dependencies on any `adapter` package.


com.myapp.domain
model
DomainModel
com.myapp.application
ApplicationConfiguration
application services must not know about any adapters
com.myapp.adapter
persistence
cli
NoSqlRepository
Cli
the domain model must be independent
the domain must not know about specific technical infrastructure
one adapter must not know about any other adapter
the domain must not know about any driving adapters
service
rest
DomainService
DomainRepository
RestController

## 8.2. Slices

Currently there are two "slice" rules offered by the Library API. These are basically rules that slice the code by packages, and contain assertions on those slices. The entrance point is:

```java
com.tngtech.archunit.library.dependencies.SlicesRuleDefinition
```

The API is based on the idea to sort classes into slices according to one or several package infixes, and then write assertions against those slices. At the moment this is for example:

```java
// sort classes by the first package after 'myapp'
// then check those slices for cyclic dependencies
SlicesRuleDefinition.slices().matching("..myapp.(*)..").should().beFreeOfCycles()

// checks all subpackages of 'myapp' for cycles
SlicesRuleDefinition.slices().matching("..myapp.(**)").should().notDependOnEachOther()

// sort classes by packages between 'myapp' and 'service'
// then check those slices for not having any dependencies on each other
SlicesRuleDefinition.slices().matching("..myapp.(**).service..").should().notDependOnEachOth
```

If this constraint is too rigid, e.g. in legacy applications where the package structure is rather inconsistent, it is possible to further customize the slice creation. This can be done by specifying a mapping of `JavaClass` to `SliceIdentifier` where classes with the same `SliceIdentifier` will be sorted into the same slice. Consider this example:

```java
SliceAssignment legacyPackageStructure = new SliceAssignment() {
    // this will specify which classes belong together in the same slice
    @Override
    public SliceIdentifier getIdentifierOf(JavaClass javaClass) {
        if (javaClass.getPackageName().startsWith("com.oldapp")) {
            return SliceIdentifier.of("Legacy");
        }
        if (javaClass.getName().contains(".esb.")) {
            return SliceIdentifier.of("ESB");
        }
        // ... further custom mappings

        // if the class does not match anything, we ignore it
        return SliceIdentifier.ignore();
    }

    // this will be part of the rule description if the test fails
    @Override
    public String getDescription() {
        return "legacy package structure";
    }
};

SlicesRuleDefinition.slices().assignedFrom(legacyPackageStructure).should().beFreeOfCycles()
```

### 8.2.1. Configurations

There are two configuration parameters to adjust the behavior of the cycle detection. They can be configured via `archunit.properties` (compare Advanced Configuration).

*archunit.properties*

```
# This will limit the maximum number of cycles to detect and thus required CPU and heap.
# default is 100
cycles.maxNumberToDetect=50

# This will limit the maximum number of dependencies to report per cycle edge.
# Note that ArchUnit will regardless always analyze all dependencies to detect cycles,
# so this purely affects how many dependencies will be printed in the report.
# Also note that this number will quickly affect the required heap since it scales with numb
# of edges and number of cycles
# default is 20
cycles.maxNumberOfDependenciesPerEdge=5
```

## 8.3. General Coding Rules

The Library API also offers a small set of coding rules that might be useful in various projects. Those can be found within

```java
com.tngtech.archunit.library.GeneralCodingRules
```

These for example contain rules not to use `java.util.logging`, not to write to `System.out` (but use logging instead) or not to throw generic exceptions.

## 8.4. PlantUML Component Diagrams as rules

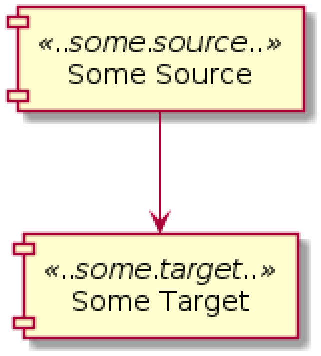
The Library API offers a feature that supports PlantUML (http://plantuml.com/component-diagram) diagrams. This feature is located in

```java
com.tngtech.archunit.library.plantuml
```

ArchUnit can derive rules straight from PlantUML diagrams and check to make sure that all imported `JavaClasses` abide by the dependencies of the diagram. The respective rule can be created in the following way:

```java
URL myDiagram = getClass().getResource("my-diagram.puml");

classes().should(adhereToPlantUmlDiagram(myDiagram, consideringAllDependencies()));
```

Diagrams supported have to be component diagrams and associate classes to components via stereotypes. The way this works is to use the respective package identifiers (compare `ArchConditions.onlyHaveDependenciesInAnyPackage(..)`) as stereotypes:



```
@startuml
[Some Source] <<..some.source..>>
[Some Target] <<..some.target..>> as target

[Some Source] --> target
@enduml
```

Consider this diagram applied as a rule via `adhereToPlantUmlDiagram(..)`, then for example a class `some.target.Target` accessing `some.source.Source` would be reported as a violation.

### 8.4.1. Configurations

There are different ways to deal with dependencies of imported classes not covered by the diagram at all. The behavior of the PlantUML API can be configured by supplying a respective `Configuration`:

```java
// considers all dependencies possible (including java.lang, java.util, ...)
classes().should(adhereToPlantUmlDiagram(
        mydiagram, consideringAllDependencies())

// considers only dependencies specified in the PlantUML diagram
// (so any unknown depedency will be ignored)
classes().should(adhereToPlantUmlDiagram(
        mydiagram, consideringOnlyDependenciesInDiagram())

// considers only dependencies in any specified package
// (control the set of dependencies to consider, e.g. only com.myapp..)
classes().should(adhereToPlantUmlDiagram(
        mydiagram, consideringOnlyDependenciesInAnyPackage("..some.package.."))
```

It is possible to further customize which dependencies to ignore:

```java
// there are further ignore flavors available
classes().should(adhereToPlantUmlDiagram(mydiagram).ignoreDependencies(predicate))
```

A PlantUML diagram used with ArchUnit must abide by a certain set of rules:

1. Components must be declared in the bracket notation (i.e. `[Some Component]`)
2. Components must have at least one (possible multiple) stereotype(s). Each stereotype in the diagram must be unique and represent a valid package identifier (e.g. `<<..example..>>` where `..` represents an arbitrary number of packages; compare the core API)
3. Components may have an optional alias (e.g. `[Some Component] <<..example..>> as myalias`)
4. Dependencies must use arrows only consisting of dashes (e.g. `-->`)
5. Dependencies may go from left to right `-->` or right to left `<--`
6. Dependencies may consist of any number of dashes (e.g `->` or `----->`)
7. Dependencies may contain direction hints (e.g. `-up->`) or color directives (e.g. `-[#green]->`)

You can compare this diagram of ArchUnit-Examples (https://github.com/TNG/ArchUnit-Examples/blob/master/example-plain/src/test/resources/com/tngtech/archunit/exampletest/shopping_example.puml)
.

## 8.5. Freezing Arch Rules

When rules are introduced in grown projects, there are often hundreds or even thousands of violations, way too many to fix immediately. The only way to tackle such extensive violations is to establish an iterative approach, which prevents the code base from further deterioration.

`FreezingArchRule` can help in these scenarios by recording all existing violations to a `ViolationStore`. Consecutive runs will then only report new violations and ignore known violations. If violations are fixed, `FreezingArchRule` will automatically reduce the known stored violations to prevent any regression.

### 8.5.1. Usage

To freeze an arbitrary `ArchRule` just wrap it into a `FreezingArchRule`:

```java
ArchRule rule = FreezingArchRule.freeze(classes().should()./*complete ArchRule*/);
```

On the first run all violations of that rule will be stored as the current state. On consecutive runs only new violations will be reported. By default `FreezingArchRule` will ignore line numbers, i.e. if a violation is just shifted to a different line, it will still count as previously recorded and will not be reported.

### 8.5.2. Configuration

By default `FreezingArchRule` will use a simple `ViolationStore` based on plain text files. This is sufficient to add these files to any version control system to continuously track the progress. You can configure the location of the violation store within `archunit.properties` (compare Advanced Configuration):

*archunit.properties*

```
freeze.store.default.path=/some/path/in/a/vcs/repo
```

Furthermore, it is possible to configure

*archunit.properties*

```
# must be set to true to allow the creation of a new violation store
# default is false
freeze.store.default.allowStoreCreation=true

# can be set to false to forbid updates of the violations stored for frozen rules
# default is true
freeze.store.default.allowStoreUpdate=false
```

This can help in CI environments to prevent misconfiguration: For example, a CI build should probably never create a new the violation store, but operate on an existing one.

As mentioned in Overriding configuration, these properties can be passed as system properties as needed. For example to allow the creation of the violation store in a specific environment, it is possible to pass the system property via

```
-Darchunit.freeze.store.default.allowStoreCreation=true
```

### 8.5.3. Extension

`FreezingArchRule` provides two extension points to adjust the behavior to custom needs. The first one is the `ViolationStore`, i.e. the store violations will be recorded to. The second one is the `ViolationLineMatcher`, i.e. how `FreezingArchRule` will associate lines of stored violations with lines of actual violations. As mentioned, by default this is a line matcher that ignores the line numbers of violations within the same class.

#### Violation Store

As mentioned in Configuration, the default `ViolationStore` is a simple text based store. It can be exchanged though, for example to store violations in a database. To provide your own implementation, implement `com.tngtech.archunit.library.freeze.ViolationStore` and configure `FreezingArchRule` to use it. This can either be done programmatically:

```java
FreezingArchRule.freeze(rule).persistIn(customViolationStore);
```

Alternatively it can be configured via `archunit.properties` (compare Advanced Configuration):

```
freeze.store=fully.qualified.name.of.MyCustomViolationStore
```

You can supply properties to initialize the store by using the namespace `freeze.store`. For properties

```
freeze.store.propOne=valueOne
freeze.store.propTwo=valueTwo
```

the method `ViolationStore.initialize(props)` will be called with the properties

```
propOne=valueOne
propTwo=valueTwo
```

### Violation Line Matcher

The `ViolationLineMatcher` compares lines from occurred violations with lines from the store. The default implementation ignores line numbers and numbers of anonymous classes or lambda expressions, and counts lines as equivalent when all other details match. A custom `ViolationLineMatcher` can again either be defined programmatically:

```java
FreezingArchRule.freeze(rule).associateViolationLinesVia(customLineMatcher);
```

or via `archunit.properties`:

```
freeze.lineMatcher=fully.qualified.name.of.MyCustomLineMatcher
```

# 9. JUnit Support

At the moment ArchUnit offers extended support for writing tests with JUnit 4 and JUnit 5. This mainly tackles the problem of caching classes between test runs and to remove some boilerplate.

Consider a straight forward approach to write tests:

```java
@Test
public void rule1() {
    JavaClasses importedClasses = new ClassFileImporter().importClasspath();

    ArchRule rule = classes()...

    rule.check(importedClasses);
}

@Test
public void rule2() {
    JavaClasses importedClasses = new ClassFileImporter().importClasspath();

    ArchRule rule = classes()...

    rule.check(importedClasses);
}
```

For bigger projects, this will have a significant performance impact, since the import can take a noticeable amount of time. Also rules will always be checked against the imported classes, thus the explicit call of `check(importedClasses)` is bloat and error prone (i.e. it can be forgotten).

## 9.1. JUnit 4 & 5 Support

Make sure you follow the installation instructions at Installation, in particular to include the correct dependency for the respective JUnit support.

### 9.1.1. Writing tests

Tests look and behave very similar between JUnit 4 and 5. The only difference is, that with JUnit 4 it is necessary to add a specific `Runner` to take care of caching and checking rules, while JUnit 5 picks up the respective `TestEngine` transparently. A test typically looks the following way:

```java
@RunWith(ArchUnitRunner.class) // Remove this line for JUnit 5!!
@AnalyzeClasses(packages = "com.myapp")
public class ArchitectureTest {

    // ArchRules can just be declared as static fields and will be evaluated
    @ArchTest
    public static final ArchRule rule1 = classes().should()...

    @ArchTest
    public static final ArchRule rule2 = classes().should()...

    @ArchTest
    public static void rule3(JavaClasses classes) {
        // The runner also understands static methods with a single JavaClasses argument
        // reusing the cached classes
    }

}
```

The `JavaClass` cache will work in two ways. On the one hand it will cache the classes by test, so they can be reused by several rules declared within the same class. On the other hand, it will cache the classes by location, so a second test that wants to import classes from the same URLs will reuse the classes previously imported as well. Note that this second caching uses soft references, so the classes will be dropped from memory, if the heap runs low. For further information see Controlling the Cache.

### 9.1.2. Controlling the Import

Which classes will be imported can be controlled in a declarative way through `@AnalyzeClasses`. If no packages or locations are provided, the whole classpath will be imported. You can specify packages to import as strings:

```java
@AnalyzeClasses(packages = {"com.myapp.subone", "com.myapp.subone"})
```

To better support refactorings, packages can also be declared relative to classes, i.e. the packages these classes reside in will be imported:

```java
@AnalyzeClasses(packagesOf = {SubOneConfiguration.class, SubTwoConfiguration.class})
```

As a third option, locations can be specified freely by implementing a `LocationProvider`:

```java
public class MyLocationProvider implements LocationProvider {
    @Override
    public Set<Location> get(Class<?> testClass) {
        // Determine Locations (= URLs) to import
        // Can also consider the actual test class, e.g. to read some custom annotation
    }
}

@AnalyzeClasses(locations = MyLocationProvider.class)
```

Furthermore to choose specific classes beneath those locations, `ImportOptions` can be specified (compare The Core API). For example, to import the classpath, but only consider production code, and only consider code that is directly supplied and does not come from JARs:

```java
@AnalyzeClasses(importOptions = {DoNotIncludeTests.class, DoNotIncludeJars.class})
```

As explained in The Core API, you can write your own custom implementation of `ImportOption` and then supply the type to `@AnalyzeClasses`.

### 9.1.3. Controlling the Cache

By default all classes will be cached by location. This means that between different test class runs imported Java classes will be reused, if the exact combination of locations has already been imported.

If the heap runs low, and thus the garbage collector has to do a big sweep in one run, this can cause a noticeable delay. On the other hand, if it is known that no other test class will reuse the imported Java classes, it would make sense to deactivate this cache.

This can be achieved by configuring `CacheMode.PER_CLASS`, e.g.

```java
@AnalyzeClasses(packages = "com.myapp.special", cacheMode = CacheMode.PER_CLASS)
```

The Java classes imported during this test run will not be cached by location and just be reused within the same test class. After all tests of this class have been run, the imported Java classes will simply be dropped.

### 9.1.4. Ignoring Tests

It is possible to skip tests by annotating them with `@ArchIgnore`, for example:

```java
public class ArchitectureTest {

    // will run
    @ArchTest
    public static final ArchRule rule1 = classes().should()...

    // won't run
    @ArchIgnore
    @ArchTest
    public static final ArchRule rule2 = classes().should()...
}
```

### 9.1.5. Grouping Rules

Often a project might end up with different categories of rules, for example "service rules" and "persistence rules". It is possible to write one class for each set of rules, and then refer to those sets from another test:

```java
public class ServiceRules {
    @ArchTest
    public static final ArchRule ruleOne = ...

    // further rules
}

public class PersistenceRules {
    @ArchTest
    public static final ArchRule ruleOne = ...

    // further rules
}

@RunWith(ArchUnitRunner.class) // Remove this line for JUnit 5!!
@AnalyzeClasses
public class ArchitectureTest {

    @ArchTest
    public static final ArchRules serviceRules = ArchRules.in(ServiceRules.class);

    @ArchTest
    public static final ArchRules persistenceRules = ArchRules.in(PersistenceRules.class);

}
```

The runner will evaluate all rules within `ServiceRules` and `PersistenceRules` against the classes declared at `ArchitectureTest`. This also allows an easy reuse of a rule library in different projects or modules.

# 10. Advanced Configuration

Some behavior of ArchUnit can be centrally configured by adding a file `archunit.properties` to the root of the classpath (e.g. under `src/test/resources`). This section will outline some global configuration options.

## 10.1. Overriding configuration

ArchUnit will use exactly the `archunit.properties` file returned by the context `ClassLoader` from the classpath root, via the standard Java resource loading mechanism.

It is possible to override any property from `archunit.properties`, by passing a system property to the respective JVM process executing ArchUnit:

```
-Darchunit.propertyName=propertyValue
```

E.g. to override the property `resolveMissingDependenciesFromClassPath` described in the next section, it would be possible to pass:

```
-Darchunit.resolveMissingDependenciesFromClassPath=false
```

## 10.2. Configuring the Resolution Behavior

As mentioned in Dealing with Missing Classes, it might be preferable to configure a different import behavior if dealing with missing classes wastes too much performance. One way that can be chosen out of the box is to never resolve any missing class from the classpath:

*archunit.properties*

```
resolveMissingDependenciesFromClassPath=false
```

If you want to resolve just some classes from the classpath (e.g. to import missing classes from your own organization but avoid the performance impact of importing classes from 3rd party packages), it is possible to configure only specific packages to be resolved from the classpath:

*archunit.properties*

```
classResolver=com.tngtech.archunit.core.importer.resolvers.SelectedClassResolverFromClasspat
classResolver.args=some.pkg.one,some.pkg.two
```

This configuration would only resolve the packages `some.pkg.one` and `some.pkg.two` from the classpath, and stub all other missing classes.

The last example also demonstrates, how the behavior can be customized freely, for example if classes are imported from a different source and are not on the classpath:

First Supply a custom implementation of

```java
com.tngtech.archunit.core.importer.resolvers.ClassResolver
```

Then configure it

*archunit.properties*

```
classResolver=some.pkg.MyCustomClassResolver
```

If the resolver needs some further arguments, create a public constructor with one `List<String>` argument, and supply the concrete arguments as

*archunit.properties*

```
classResolver.args=myArgOne,myArgTwo
```

For further details, compare the sources of `SelectedClassResolverFromClasspath`.

## 10.3. MD5 Sums of Classes

Sometimes it can be valuable to record the MD5 sums of classes being imported to track unexpected behavior. Since this has a performance impact, it is disabled by default, but it can be activated the following way:

*archunit.properties*

```
enableMd5InClassSources=true
```

If this feature is enabled, the MD5 sum can be queried as

```java
javaClass.getSource().get().getMd5sum()
```