Bezpieczeństwo w inżynierii oprogramowania

Testy bezpieczeństwa i inne zagadnienia

Arkadiusz Chrobot

Katedra Systemów Informatycznych

11 stycznia 2025

Plan

1. Wstęp
2. Testy bezpieczeństwa
   - Testy jednostkowe
   - Testy integracyjne
3. Bezpieczeństwo łańcuchów dostaw
4. Wycofywanie oprogramowania z użycia

Wstęp

Wykład poświęcony będzie fazom występującym zazwyczaj w drugiej części cyklu życia oprogramowania. Pierwsza część będzie dotyczyła testowania, rozumianego jako testowanie dynamiczne i jego związków z bezpieczeństwem. W drugiej poruszone zostaną zagadnienia bezpieczeństwa artefaktów pozyskiwanych od stron trzecich i wycofywania oprogramowania z użycia.

Testy bezpieczeństwa

Testowanie bezpieczeństwa jest bardzo obszerną dziedziną obejmującą takie zagadnienia jak testowanie rozmyte (ang. *fuzzing*), SAST (ang. *Static Application Security Testing*), DAST (ang. *Dynamic Application Security Testing*) oraz testy penetracyjne. Wykład będzie jednak poświęcony testom bezpieczeństwa, jakie można przeprowadzić na poziomie jednostkowym i integracyjnym. Będzie on bazował na rekomendacjach pracowników firmy Google [1].

Testy jednostkowe

Definicje

Przyjmuje się, że testowanie jednostkowe (ang. *unit testing*) polega na przeprowadzaniu testów elementów kodu, które nie mają zewnętrznych zależności, są niezależne. Najczęściej tymi elementami są pojedyncze obiekty, ich metody lub funkcje. W rzeczywistym oprogramowaniu większość komponentów nie jest niezależna, więc w praktyce testowanie jednostkowe polega na wykonywaniu testów wyizolowanych, względnie małych komponentów kodu.
Ten rodzaj testów jest najczęściej automatyzowany przy pomocy odpowiedniej biblioteki lub platformy (ang. *framework*). Tradycyjnie testy te wykonywane są w środowisku pracy programistów, aby dostarczyć im informacji zwrotnej o jakości modyfikacji kodu, które wykonali, zanim trafią one do repozytorium. W potoku CI/CD są one wykonywane zanim zmiana (ang. *commit*) zostanie włączona (ang. *merge*) do głównej gałęzi kodu.

Testy jednostkowe

Tworzenie efektywnych testów jednostkowych

Testy jednostkowe powinny być szybkie, niezawodne i hermetyczne, tzn. dostarczać programistom informacji zwrotnej, na której mogą polegać, najszybciej jak to możliwe i w sposób powtarzalny. Jeśli ta ostatnia własność nie jest zachowana, to testy stają się *kruche* (ang. *flaky*) i zazwyczaj świadczą o tym, że testowany element oprogramowania nie jest w pełni niezależny.

Aby uzyskać efektywne testy jednostkowe bezpieczeństwa, należy postępować podobnie, jak w przypadku każdych innych rodzajów testów jednostkowych. Najpierw należy je opracować nie tylko dla *typowych* przypadków, ale również dla brzegowych (ang. *edge cases*). Następnie należy dodać testy, w których dane wejściowe są zniekształcone lub niebezpieczne i upewnić się, że testowana jednostka kodu je wykrywa. Ten sam kod powinien być sprawdzany dla różnych parametrów i danych środowiskowych, np. testowany dla kilku użytkowników z różnymi uprawnieniami.

Testy jednostkowe

Tworzenie efektywnych testów jednostkowych

Testy jednostkowe są zazwyczaj tworzone *po* utworzeniu zmian w kodzie produkcyjnym, aby sprawdzić ich poprawność. Problem z takim podejściem polega na tym, że te testy mogą kończyć się pomyślnie (tzn. dawać negatywne wyniki) nawet wtedy, gdy testowany kod jest nieaktywny lub wręcz usunięty. Aby uniknąć takich problemów zaleca się stosowanie metodologii (TDD) (ang. *Test-Driven Development*), polegającej (w uproszeniu) na pisaniu testów *przed* kodem produkcyjnym. Oznacza to, że początkowo wszystkie testy będą się kończyły niepomyślnie (tzn. będą dawały wynik pozytywny), ale wtedy wiadomo, że faktycznie weryfikują one zachowanie kodu produkcyjnego. Pozwala to także śledzić postęp prac. Ta metodologia także nie jest pozbawiona wad. Może np. zachęcać programistów do pisania kodu produkcyjnego, który spełnia testy, ale nie realizuje zadań określonych wymaganiami. Niemniej jednak, prawidłowo stosowana może przynieść korzyści.

Testy jednostkowe

Zastępowanie zależności w testach jednostkowych

Testowanie kodu jednostkowego, który np. wywołuje zewnętrzną usługę w środowisku produkcyjnym jest albo niemożliwe, albo co najmniej uciążliwe. W eliminacji tej trudności pomagają abstrakcje nazwane *dublerami testowymi*, choć w różnych opracowaniach stosowane są inne nazwy, jak *namiastki* (ang. *stub*), atrapy (ang. *mock*). Niestety, nie istnieje w tej kwestii ogólny standard nazewniczy, ale można posłużyć się ▸ klasyfikacją sporządzoną przez Gerarda Meszarosa, pracownika firmy Microsoft. Dublery testowe pozwalają sprawdzić, jak testowany komponent korzysta z usługi dostarczanej przez inny element oprogramowania, np. czy wywołuje ją z prawidłowymi argumentami i czy robi to odpowiednią liczbę razy. Dzięki nim testy stają się niezależne od zewnętrznych zależności, ale pojawia się ryzyko związane z niejawnymi założeniami odnośnie interakcji ze współpracującym komponentem, np. przyjęcie za pewnik, że zawsze będzie on dostępny lub że będzie przyjmował argumenty w ściśle ustalonej kolejności.

Testy integracyjne

Testy integracyjne sprawdzają interakcje testowanego kodu z prawdziwymi komponentami, nie z dublerami testowymi. W związku z tym weryfikują kompletne lub prawie kompletne ścieżki kodu. To powoduje, że wraz z rozwojem oprogramowania te testy stają się coraz bardziej skomplikowane, co z jednej strony pozwala zwiększyć przekonanie (ang. *confidence*), że system działa zgodnie z wymaganiami, ale z drugiej strony powoduje wydłużanie czasu realizacji tych testów i zwiększa ich kruchość. Problemem powiązanym bezpośrednio z bezpieczeństwem, jest przygotowanie testów integracyjnych w taki sposób, aby były one zgodne z zasadą najmniejszych uprawnień (wiedzy koniecznej). Przykładowo, kopiowanie do środowiska testowego bazy danych z środowiska produkcyjnego udostępnia ją dla osób, które być może nie powinny mieć do niej wglądu i może skutkować wyciekiem danych. Dlatego w takiej sytuacji zalecane jest przygotowanie specjalnej wersji tej bazy, która nie zawiera wrażliwych danych.

Biblioteka ArchUnit

Biblioteka ArchUnit, trochę wbrew nazwie, służy do przeprowadzania specjalnego rodzaju testów integracyjnych — weryfikacji architektury. Ponieważ architektura decyduje o spełnieniu przez oprogramowanie wymagań niefunkcjonalnych, a jednym z nich jest bezpieczeństwo, to jej testy są szczególnie istotne dla weryfikacji zabezpieczeń. Biblioteka ArchUnit pozwala sprawdzić poprawność zależności między pakietami, klasami, warstwami i fragmentami (ang. *slices*), a także wykryć zależności cykliczne, które są w architekturze niepożądane [3]. Dokonuje ona weryfikacji kodu aplikacji importując jej kod bajtowy i odwzorowując go na strukturę kodu źródłowego. Import ten wykonywany jest przez następujący fragment kodu:

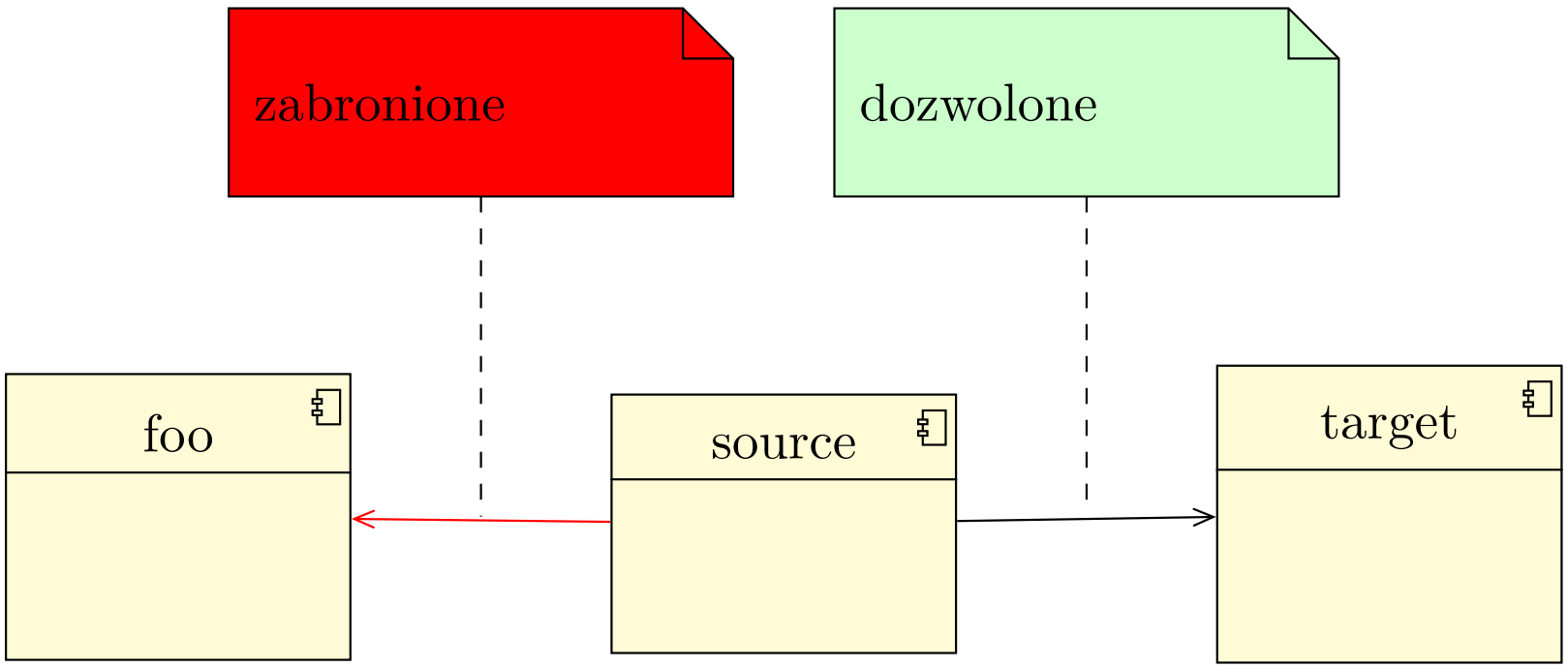
```
JavaClasses classes = new
↪  ClassFileImporter().importPackages("com.myapp");
```

Metoda `importPackages` może jako argument przyjmować także ścieżkę do katalogu z klasami, a zwraca referencję do kolekcji obiektów reprezentujących klasy.

Biblioteka ArchUnit

ArchUnit nie implementuje funkcji tworzących środowisko dla testów i umożliwiających ich przeprowadzenie. Współpracuje ona w tym zakresie z innymi bibliotekami, takimi jak JUnit. Zawiera jednak kod umożliwiający opisanie reguł dotyczących zależności, jakie uznawane są za prawidłowe w testowanym oprogramowaniu. Programista piszący testy może tworzyć tworzyć te zasady samodzielnie lub skorzystać z gotowych bibliotek, które zawierają reguły weryfikacji dla np. architektury wielowarstwowej, heksagonalnej lub typu VSA.
Kolejne slajdy zawierają przykłady reguł testujących niektóre zależności między poszczególnymi konstrukcjami w kodzie Javy z użyciem ArchUnit.

Biblioteka ArchUnit



Rysunek: Sprawdzanie prostych zależności między pakietami za pomocą ArchUnit (na podstawie: [3])

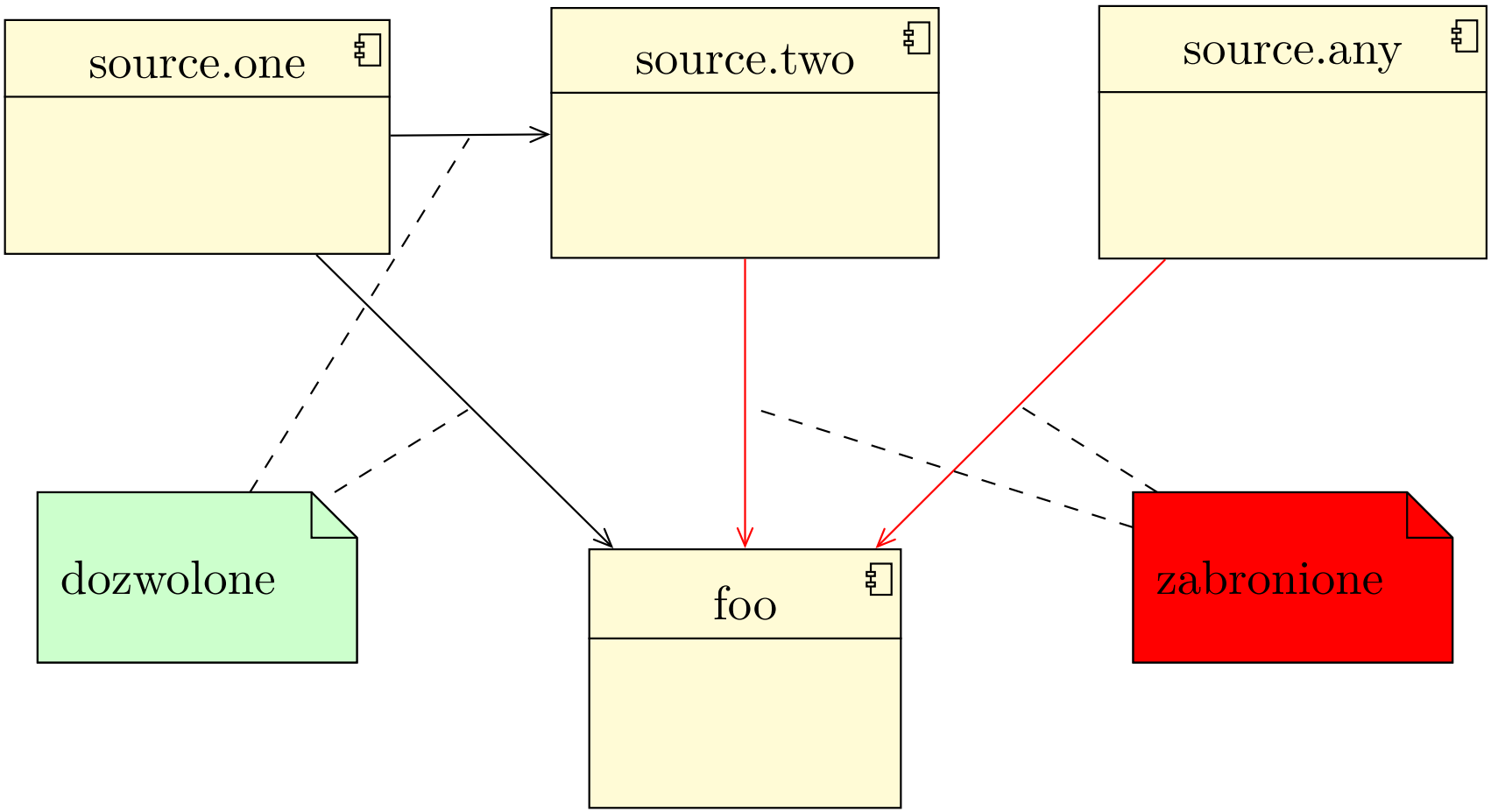
Biblioteka ArchUnit

Reguła z Listingu 1 pozwala sprawdzić, czy klasy, które znajdują się w pakiecie lub pakietach z nazwą zawierającą wyraz „source” nie są zależne od klas w pakietach zawierających w nazwie wyraz „foo”. Diagram komponentów UML z Rysunku 1 ilustruje zarówno przykład poprawnej, jak i niepoprawnej zależności miedzy takimi pakietami. Należy zaznaczyć, że w języku Java komponent UML jest zazwyczaj implementowany jako pakiet.

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

Listing: Sprawdzanie prostych zależności między pakietami za pomocą ArchUnit (źródło: [3])

Biblioteka ArchUnit



Rysunek: Sprawdzanie złożonych zależności między pakietami za pomocą ArchUnit (na podstawie: [3])

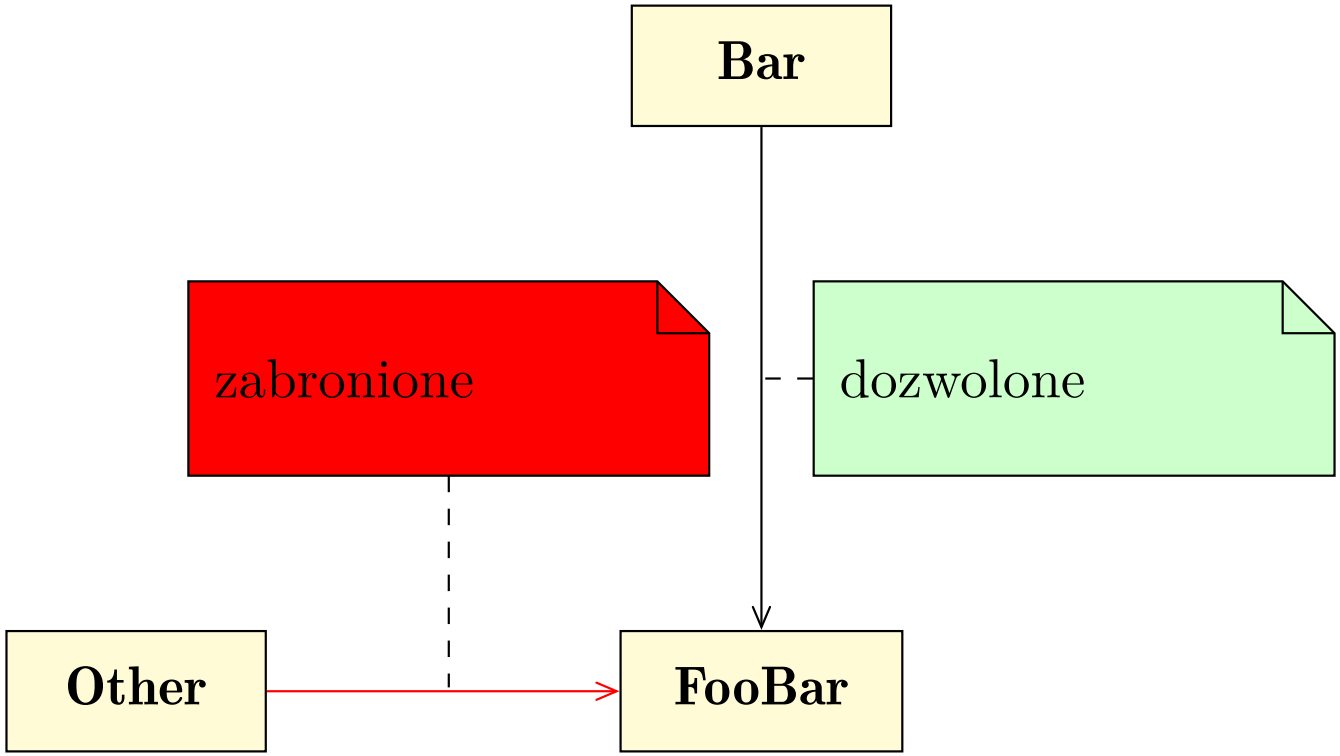
Biblioteka ArchUnit

Reguła z Listingu 2 sprawdza bardziej złożone zależności między pakietami, takie których przykłady zostały podane w diagramie z Rysunku 2.

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

Listing: Sprawdzanie złożonych zależności między pakietami za pomocą ArchUnit (źródło: [3])

Biblioteka ArchUnit



Rysunek: Sprawdzanie prostych zależności między klasami za pomocą ArchUnit (na podstawie: [3])

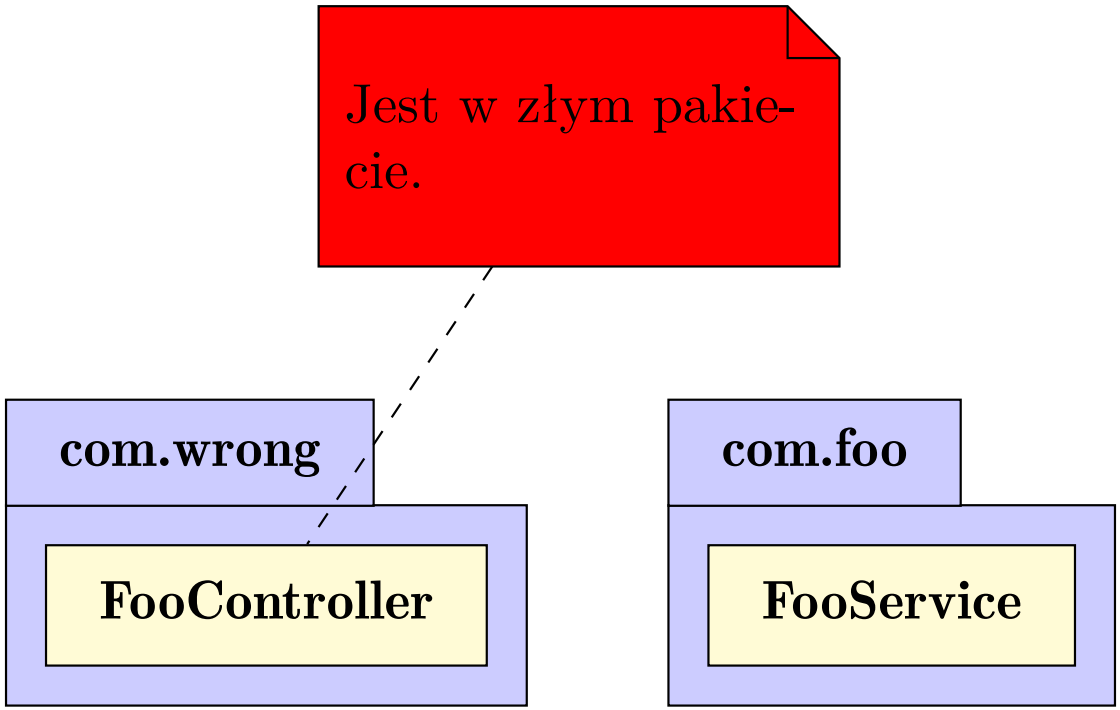
Biblioteka ArchUnit

Reguła z Listingu 3 sprawdza zależności między klasami, których przykłady opisuje diagram z Rysunku 3.

```
classes().that().haveNameMatching(".*Bar")
.should().onlyHaveDependentClassesThat().haveSimpleName("Bar")
```

Listing: Sprawdzanie prostych zależności między klasami za pomocą ArchUnit (źródło: [3])

Biblioteka ArchUnit



Rysunek: Sprawdzanie zawierania klas w pakietach za pomocą ArchUnit (na podstawie: [3])

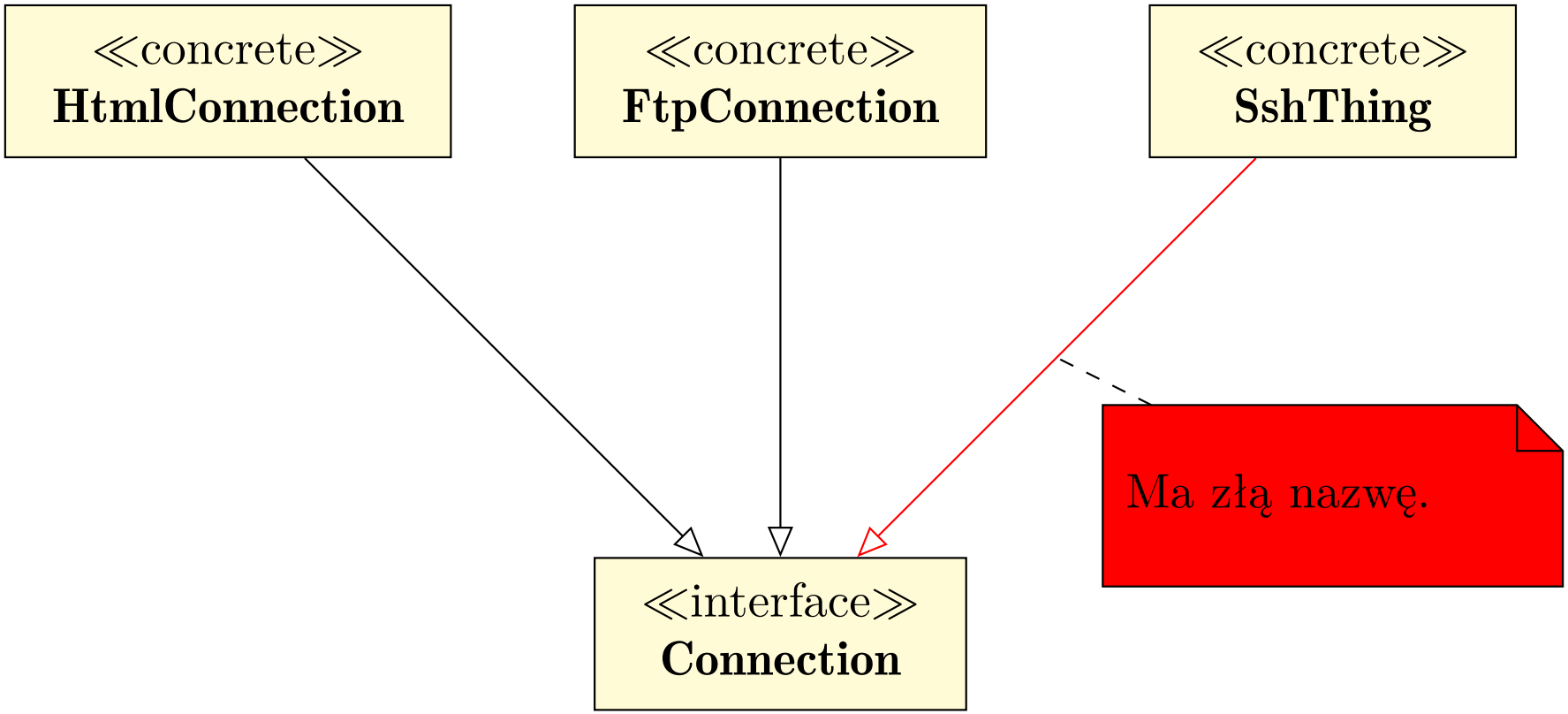
Biblioteka ArchUnit

Reguła 4 sprawdza, czy klasy znajdujące się w określonych pakietach mają odpowiednie nazwy lub czy znajdują się w odpowiednim pakiecie. Należy zwrócić uwagę, że pakiet UML może być w języku Java zaimplementowany jako pakiet albo jako moduł. Przykład ilustruje diagram z Rysunku 4.

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

Listing: Sprawdzanie zawierania klas w pakietach za pomocą ArchUnit (źródło: [3])

Biblioteka ArchUnit



Rysunek: Sprawdzanie konwencji nazewniczej w dziedziczeniu za pomocą ArchUnit (na podstawie: [3])

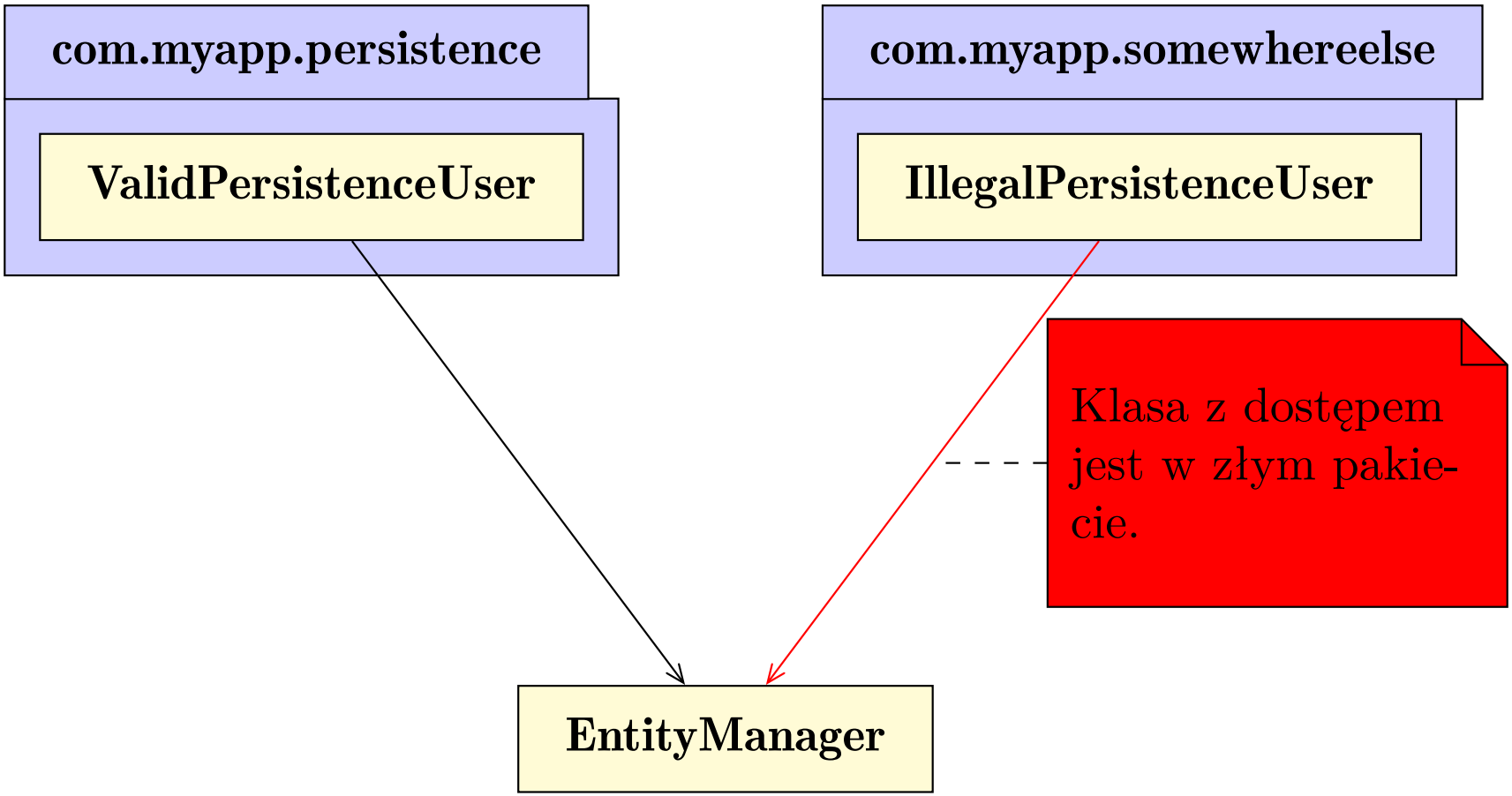
Biblioteka ArchUnit

Reguła z Listingu 5 sprawdza, czy nazwy klas implementujących określony interfejs przestrzegają wymaganej konwencji nazewniczej. Diagram z Rysunku 5 zawiera przykłady prawidłowo nazwanych klas i klasy z nieprawidłową nazwą.

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

Listing: Sprawdzanie konwencji nazewniczej w dziedziczeniu za pomocą ArchUnit (źródło: [3])

Biblioteka ArchUnit



Rysunek: Sprawdzanie poprawności odwołań za pomocą ArchUnit (na podstawie: [3])

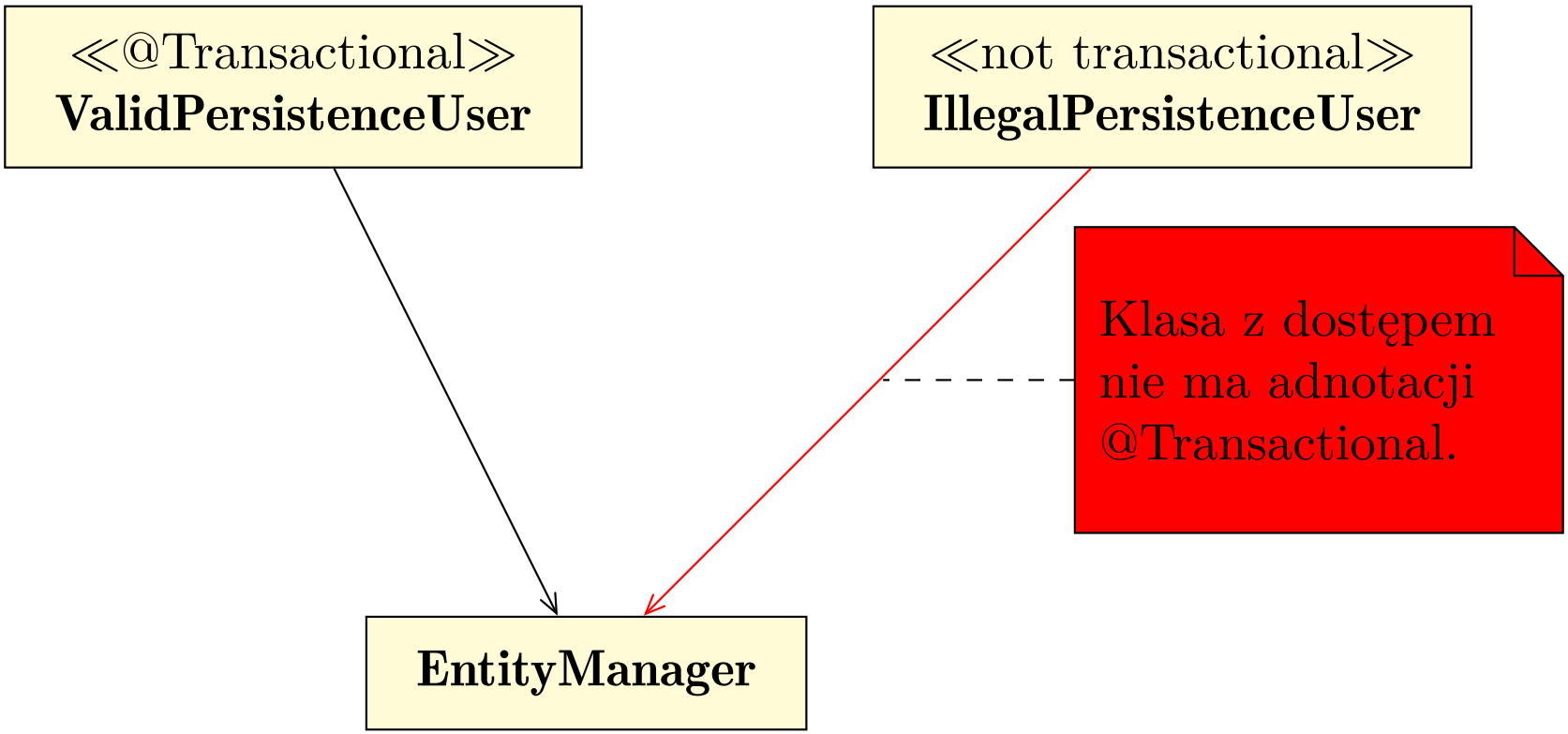
Biblioteka ArchUnit

Reguła z Listingu 6 sprawdza, czy tylko klasy z określonego pakietu mają dostęp do obiektu lub obiektów określonej klasy. Przykład ilustruje diagram 6, w którym do klasy obiektów zarządzających encjami dostęp powinny mieć tylko obiekty klas z pakietu związanego z utrwalaniem (ang. *persistence*) danych.

```
classes().that().areAssignableTo(EntityManager.class)
    .should().onlyHaveDependentClassesThat()
        .resideInAnyPackage("..persistence..")
```

Listing: Sprawdzanie poprawności odwołań za pomocą ArchUnit (źródło: [3])

Biblioteka ArchUnit



Rysunek: Sprawdzanie adnotacji za pomocą ArchUnit (na podstawie: [3])

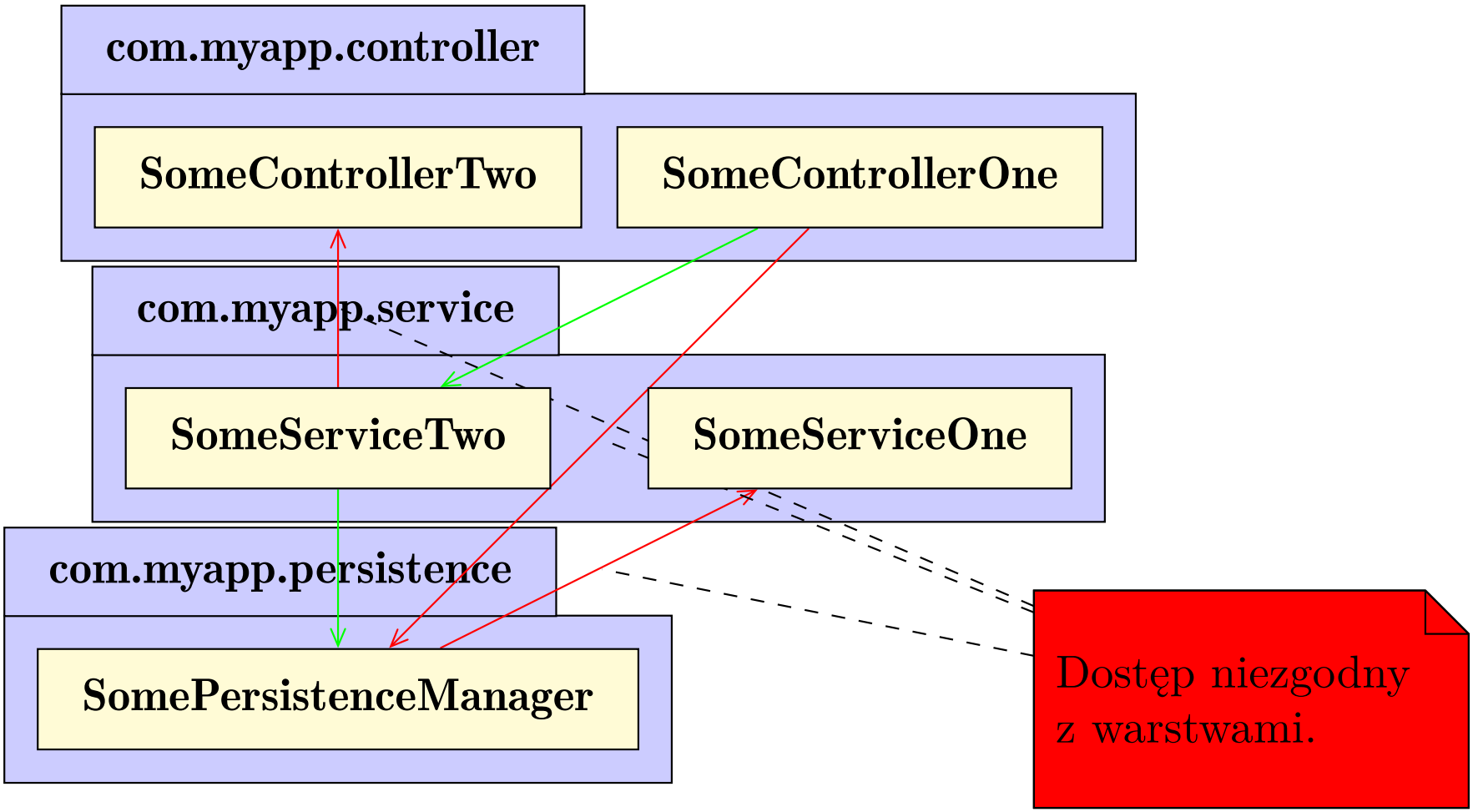
Biblioteka ArchUnit

Reguła z Listingu 7 sprawdza, czy klasy mające dostęp do określonej innej klasy są opisane odpowiednią adnotacją. Diagram z Rysunku 7 zawiera zarówno przykład klas spełniającej, jak i niespełniającej tej reguły.

```
classes().that().areAssignableTo(EntityManager.class)
    .should().onlyHaveDependentClassesThat()
        .areAnnotatedWith(Transactional.class)
```

Listing: Sprawdzanie adnotacji za pomocą ArchUnit (źródło: [3])

Biblioteka ArchUnit



Rysunek: Sprawdzanie zależności między warstwami za pomocą ArchUnit (na podstawie: [3])

Biblioteka ArchUnit

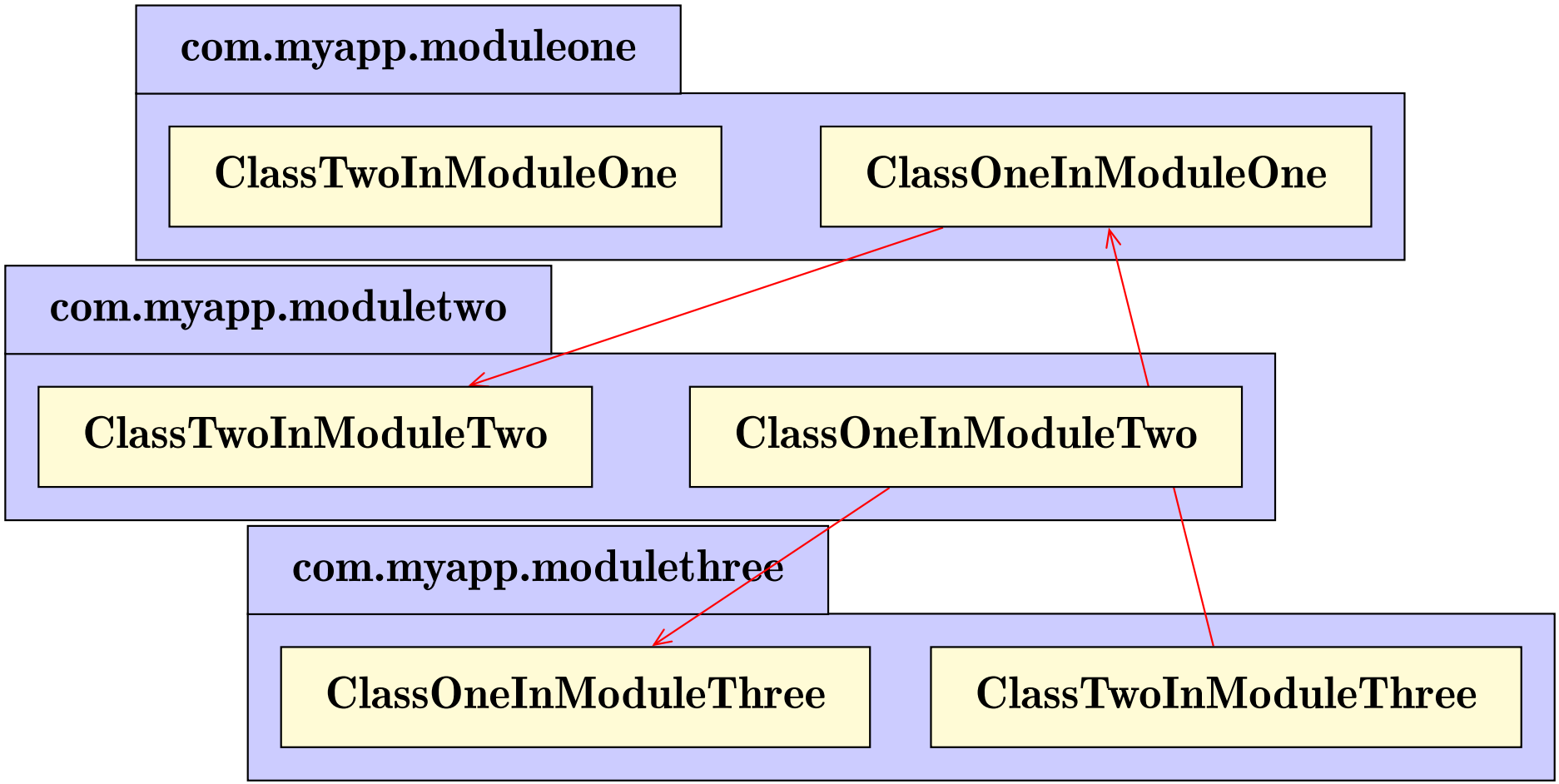
Reguła z Listingu 8 sprawdza prawidłowość porządku odwołań między warstwami. Przykłady poprawnych i niepoprawnych zależności ilustruje diagram z Rysunku 8.

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

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

Listing: Sprawdzanie zależności między warstwami za pomocą ArchUnit (źródło: [3])

Biblioteka ArchUnit



Rysunek: Sprawdzanie zależności cyklicznych ArchUnit (na podstawie: [3])

Biblioteka ArchUnit

Reguła z Listingu 9 sprawdza, czy w oprogramowaniu posiadającym architekturę VAS (ang. *Vertical Slices Architecture*) nie występują niepożądane zależności cykliczne. Diagram z Rysunku 9 zawiera przykład struktury, która nie spełnia tej reguły.

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

Listing: Sprawdzanie zależności cyklicznych ArchUnit (źródło: [3])

Bezpieczeństwo łańcuchów dostaw

Współcześnie oprogramowanie tworzone jest z użyciem gotowych komponentów (bibliotek, platform) pochodzących od innych producentów (ang. *third parties*), którzy udostępniają je na zasadach open source lub komercyjnych. Z kolei te komponenty są tworzone przy pomocy innych, które mogą wykorzystywać jeszcze inne elementy, pochodzące od jeszcze innych producentów. W tej sposób tworzy się ciąg zależności, który nazywany jest *łańcuchem dostaw* (ang. *supply chain*) [2]. Ten termin został zapożyczony w informatyce z inżynierii produkcji. Istnienie takich zależności ma niezaprzeczalne korzyści, ale z punktu widzenia bezpieczeństwa jest potencjalną słabością. Znane są przykłady wykorzystania łańcucha dostaw do dokonywania cyberataków (np. SolarWinds, biblioteka XZ, Equifax).

Bezpieczeństwo łańcuchów dostaw

Aby uniknąć takich ataków dostawca oprogramowania musi wiedzieć z jakich elementów są stworzone komponenty, z których korzysta, jakie są ich wersje i czy istnieją podatności, które zostały w nich wykryte. Dla producentów oprogramowania komercyjnego mogą być ważne także licencje na jakich komponenty typu open source są udostępniane. Takie wykazy wykorzystywanych komponentów, wraz z podanymi informacjami na ich temat, nazywane są *Listami Materiałowymi Oprogramowania* (ang. *Software Bill of Materials* — SBOM) [4]. Opracowano narzędzia, które pozwalają na tworzenie i zarządzanie taką listą. Niektóre środowiska do programowania (np. IntelliJ Idea) automatycznie sprawdzają, czy używane wersje komponentów, np. wymienione w pliku konfiguracyjnym narzędzia Maven, nie zawierają znanych podatności. Tworzenie SBOM jest zalecane przez wielu ekspertów ds. cyberbezpieczeństwa, a w przypadku firm dostarczających oprogramowanie dla instytucji rządu USA, wprost wymagane.

Wycofywanie oprogramowania z użycia

Systemy oprogramowania są zastępowane w całości lub częściowo nowymi. Pojawia się zatem potrzeba wyłączenia starszych ich odpowiedników z użycia. Okazuje się, że nawet w tym działaniu pojawiają się kwestie, które związane są z cyberbezpieczeństwem [5]. Zaleca się opracowanie procedury wycofywania oprogramowania z użycia, która powinna obejmować następujące kroki:

1. Zbadanie, na podstawie listy zasobów, wpływu wycofania określonego zasobu na działanie całości systemu.
2. Ustalenie z właścicielami wycofywanych zasobów dalszych działań.
3. Ocenę ryzyka.
4. Utworzenie koniecznych kopii zapasowych.
5. Bezpieczne usunięcie danych.
6. Utworzenie raportu wycofania, wraz z listą pozostałych ryzyk, które muszą być monitorowane.

Wycofywanie oprogramowania z użycia

Ocena ryzyka

Ocena ryzyka dokonywana jest podobnie jak w przypadku zasobów aktywnych, ale przy założeniu że niektóre komponenty systemu lub całe usługi przestają być dostępne. Może ona przebiegać np. zgodnie z następującą listą pytań:

- Jak bardzo wrażliwe dane są przechowywane w zasobie?
- Czy te dane muszą być zachowane, przeniesione, czy bezpiecznie usunięte?
- Czy proces wycofania może spowodować powstanie tymczasowych ryzyk, którym należy przeciwdziałać?
- Jaki wpływ ma ten proces na inne zasoby, usługi lub systemy?
- Czy wycofanie zwiększy ryzyko w innych obszarach usługi?
- Czy wycofanie pozostawi zależne zasoby lub infrastrukturę podatnymi?

Źródła I


Heather Adkins, Betsy Beyer, Paul Blankinship, Piotr Lewandowski, Ana Oprea i Adam Stubblefield. *Building Secure & Reliable Systems: Best Practices for Design, Implementing Maintaining Systems*. O'Reilly Media, Sebastopol, CA, USA, 2020. URL: https://sre.google/books/building-secure-reliable-systems/.


Dan Geer, Bentz Tozer i John Speed Mayers. For Good Measure Counting Broken Links: A Quant's View of Software Supply Chain Security. *;login:* 45(4):83–86, 2020. URL: https://www.usenix.org/system/files/login/articles/login_winter20_17_geer.pdf (termin wizyty 05.01.2025).


ArchUnit. URL: https://www.archunit.org/ (termin wizyty 04.01.2025).

Źródła II

- Software Bill of Materials. URL: https://www.cisa.gov/sbom (termin wizyty 05. 01. 2025).
- Secure by Design Principles. URL: https://www.security.gov.uk/policy-and-guidance/secure-by-design/principles/ (termin wizyty 03. 01. 2025).

Pytania

?

Koniec

Dziękuję Państwu za uwagę!