OSGi - The Dynamic Module System for Java
Eclipse RCP

Techniki Obiektowe
i Komponentowe

Kamil Piętak, Aleksander Byrski
{kpietak,olekb}@agh.edu.pl

Agenda

- Komponentowość w świecie Java (przed OSGi)
- Komponentowość proponowana przez OSGi
- Dynamiczne systemy budowane wg OSGi
- Serwisy
- Popularne implementacje
- Eclipse RCP

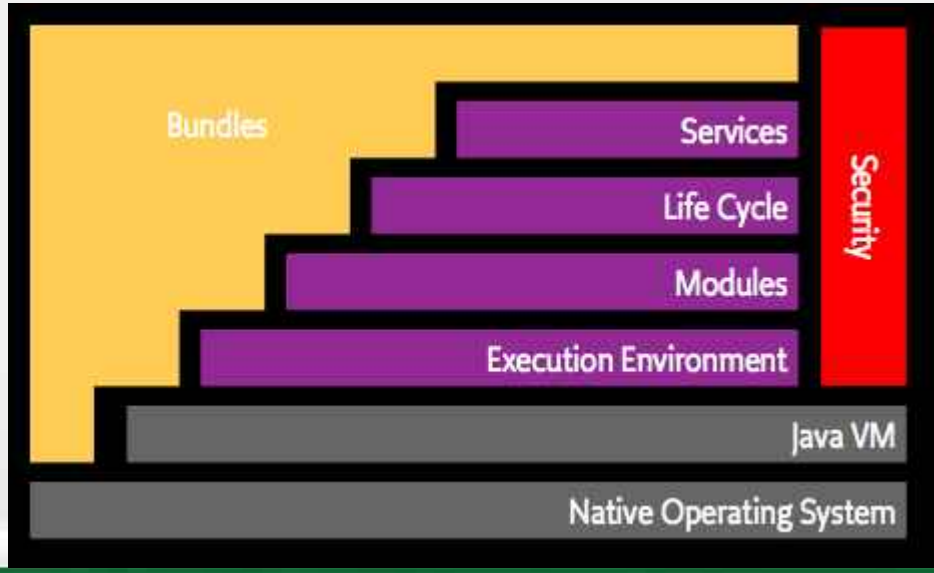
Aspekty OSGi

OSGi – The Dynamic Module System for Java

- Komponentowość w świecie Java
- Dynamika w czasie działania systemu
- Technologia zorientowana na serwisy

Architektura OSGi

- Bundles – komponenty OSGi, mogą wykorzystywać serwisy,
- Services – dynamiczne łączenie bundles (publish-find-bind model dla POJO)
- Life-cycle – start, stop, update, install, uninstall bundle...
- Module – ładowanie klas, import, eksport bundle
- Security – autentykacja itp.
- Execution environment - definicja klas dostępnych na platformie



Komponentowość w świecie Java

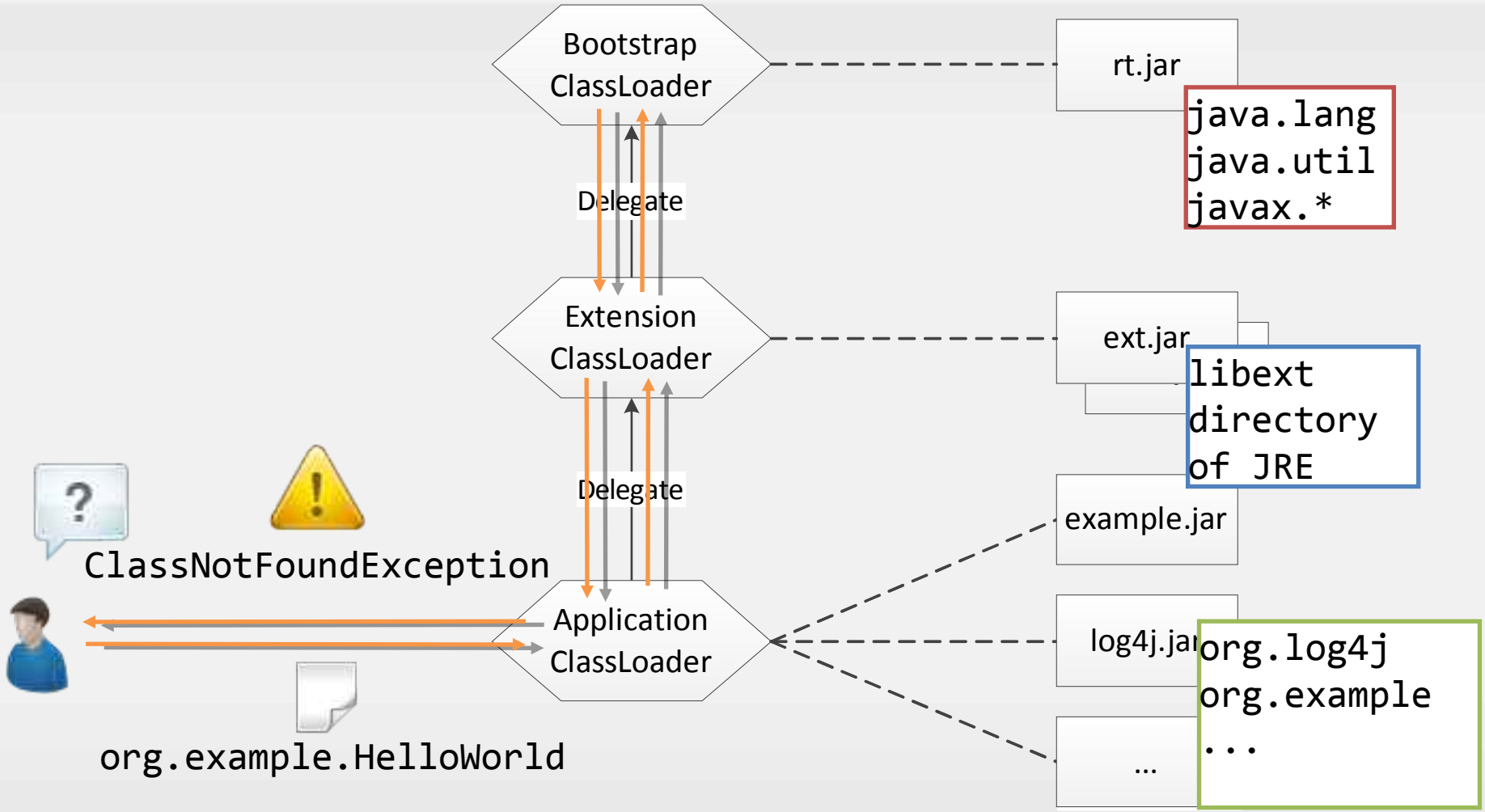
Cechy komponentowe

- Samodzielny (ang. *self-contained*) – komponent jest logiczną całością
- Highly cohesive – komponent powienien dostarczać powiązane logicznie funkcjonalności
- Loosely coupled – komponent nie powinien zależeć od wewnętrznej implementacji innych modułów

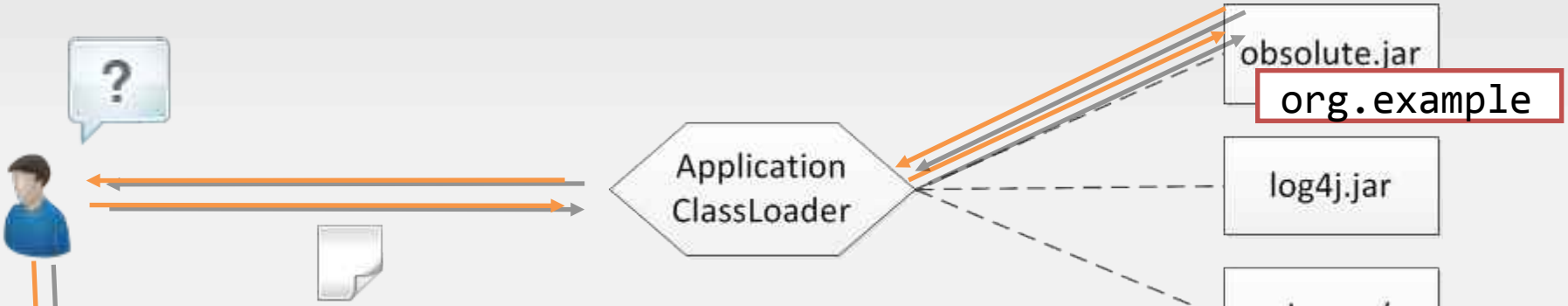
Komponenty w świecie Java

- Klasy
  - Bardzo duża granularyzacja
  - Nie jest jednostką wdrożenia (deployment)
- Pakiety
  - Nie jest jednostką wdrożenia
- Pliki JAR
  - Jednostka wdrożenia
  - JDK dostarcza jedynie b. podstawowe narzędzia do zarządzania plikami JAR → „JAR Hell”

Globalna przestrzeń klas



Konflikty klas

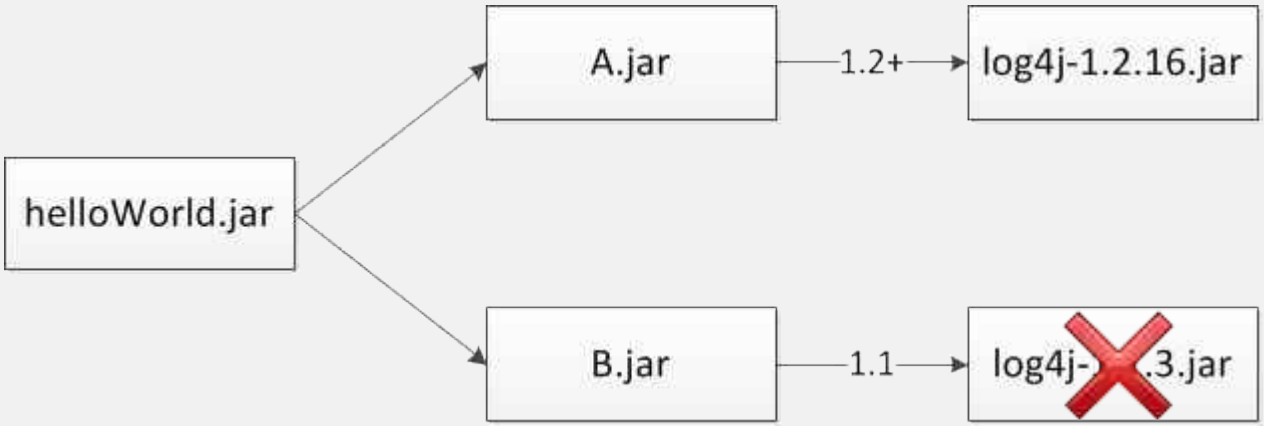



```
org.example.MessageToMyGirlfriend
```

```
msg.sendMessage(„Greetings”); // for Kasia
```

```
Greetings, Joasia!
```

Brak informacji o wersjach



Jedynym rozwiązaniem jest przepisanie/zmiana bibliotek A.jar i B.jar

Problemy z plikami JAR

- Pliki JAR funkcjonują jedynie w czasie budowania i instalacji aplikacji; w czasie działania wszystkie pliki JAR traktowane są jako jedna przestrzeń nazw → *class-path*
- Brak mechanizmu ukrywania informacji pomiędzy plikami JAR
  - Wszystkie klasy dostarczone w pliku JAR są widoczne w class-pathie i mogą być wykorzystane przez inne klasy
- Nie można wykorzystać wielu wersji tej samej klasy jednocześnie
- Brak standardu opisu zależności

OSGi – środowisko komponentowe dla Javy

Główne założenia

- Komponentowe środowisko uruchomieniowe
- Moduł OSGi – bundle – plik JAR z dodatkowymi meta-danymi w pliku MANIFEST.MF
  - Nazwa
  - Wersja
  - Zależności – importowane pakiety; nazwy zależnych bundles
  - Udostępniane (eksportowane) pakiety
- Każdy bundle ma osobny class-path

Metadane w MANIFEST.MF

Podstawowe informacje:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2

Bundle-SymbolicName: pl.edu.agh.email.sender
Bundle-Version: 1.0.0.qualifier

Bundle-Name: Email Sender
Bundle-Vendor: AGH Krakow

Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Bundle-Localization: plugin
```

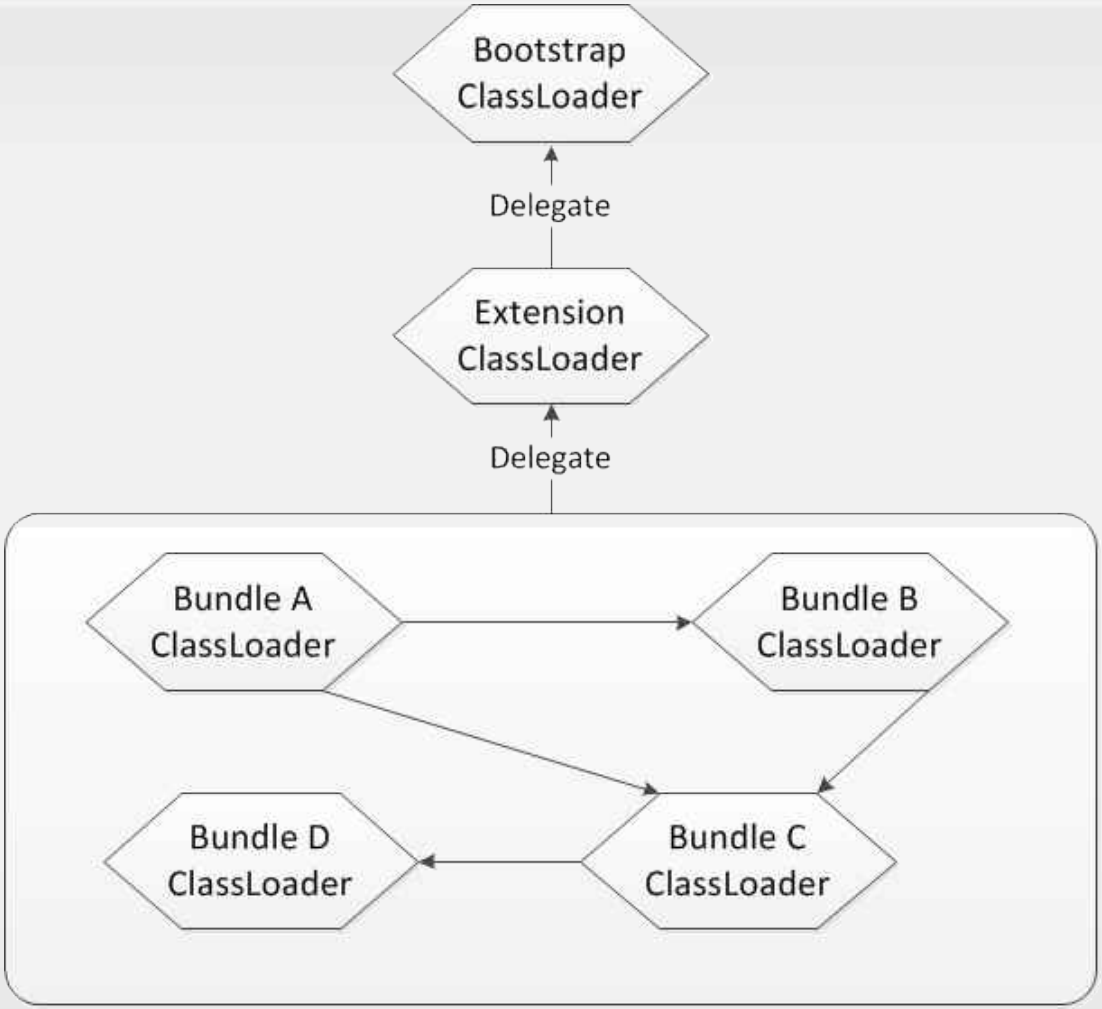
Nagłówek

Identyfikator (nazwa) i wersja

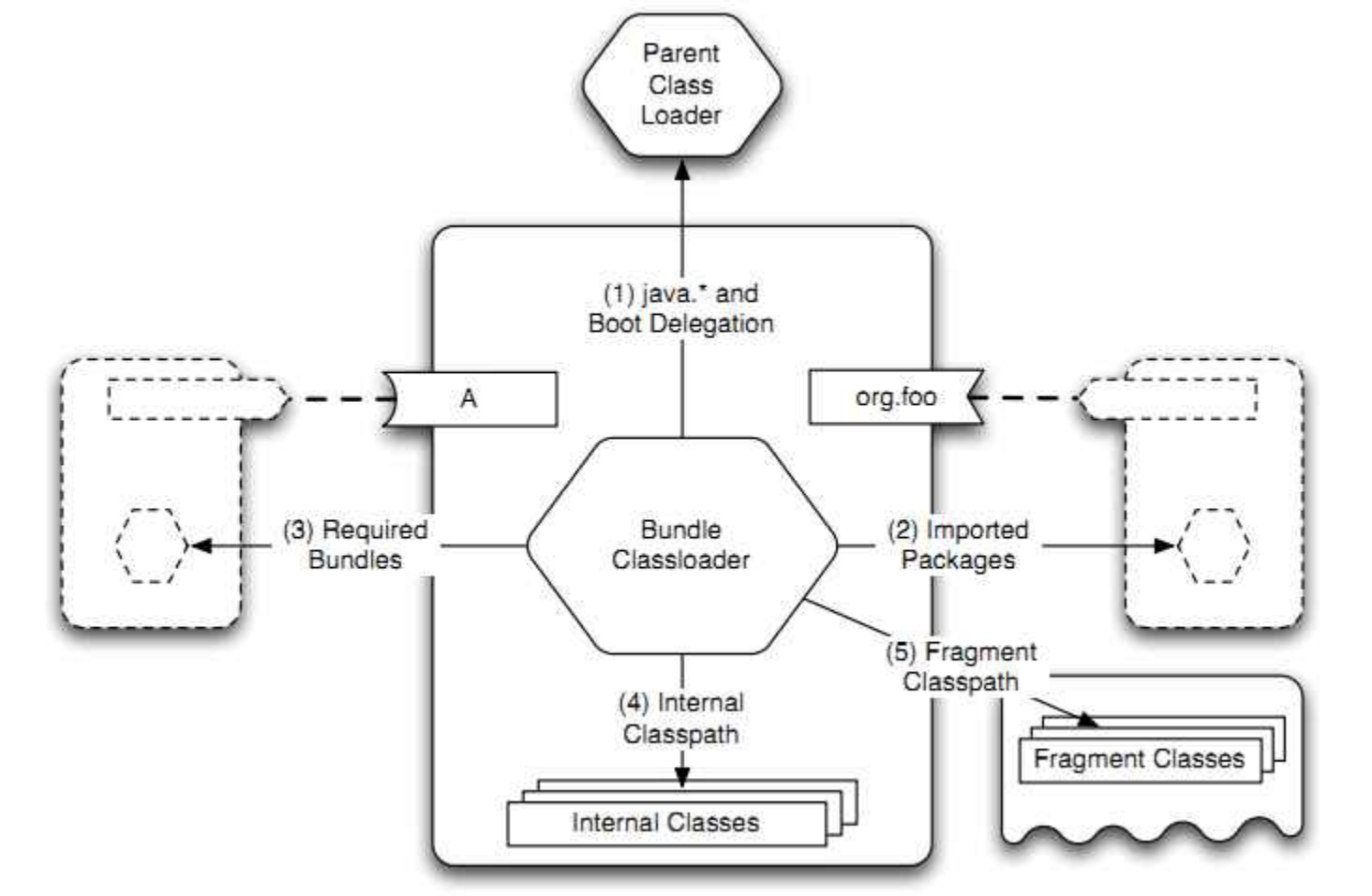
Dodatkowe informacje dla użytkownika

Wymagane wersja środowiska uruchomieniowego i dodatkowe dane

Struktura Class Loaderów

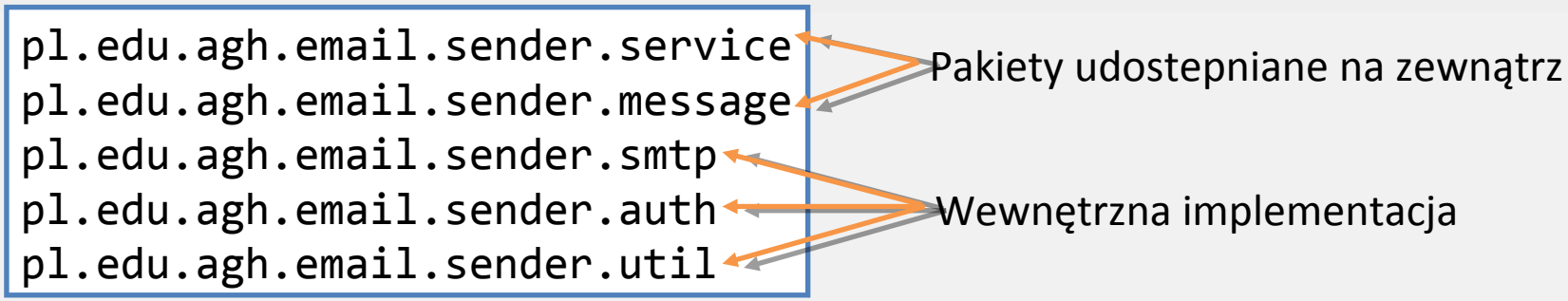


Ładowane klas w OSGi



Ukrywanie wewnętrznej implementacji

- Bundle *email-sender*:



MANIFEST.MF

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-SymbolicName: pl.edu.agh.email.sender
Export-Package: pl.edu.agh.email.sender.service,
  pl.edu.agh.email.sender.message
```

Zależności

- Wymagane pakiety

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-SymbolicName: pl.edu.agh.email.sender
Import-Package: pl.edu.agh.spell.checker.service,
  pl.edu.agh.spell.checker.dictionary,
  pl.edu.agh.addressbook.service,
  pl.edu.agh.addressbook.model
```

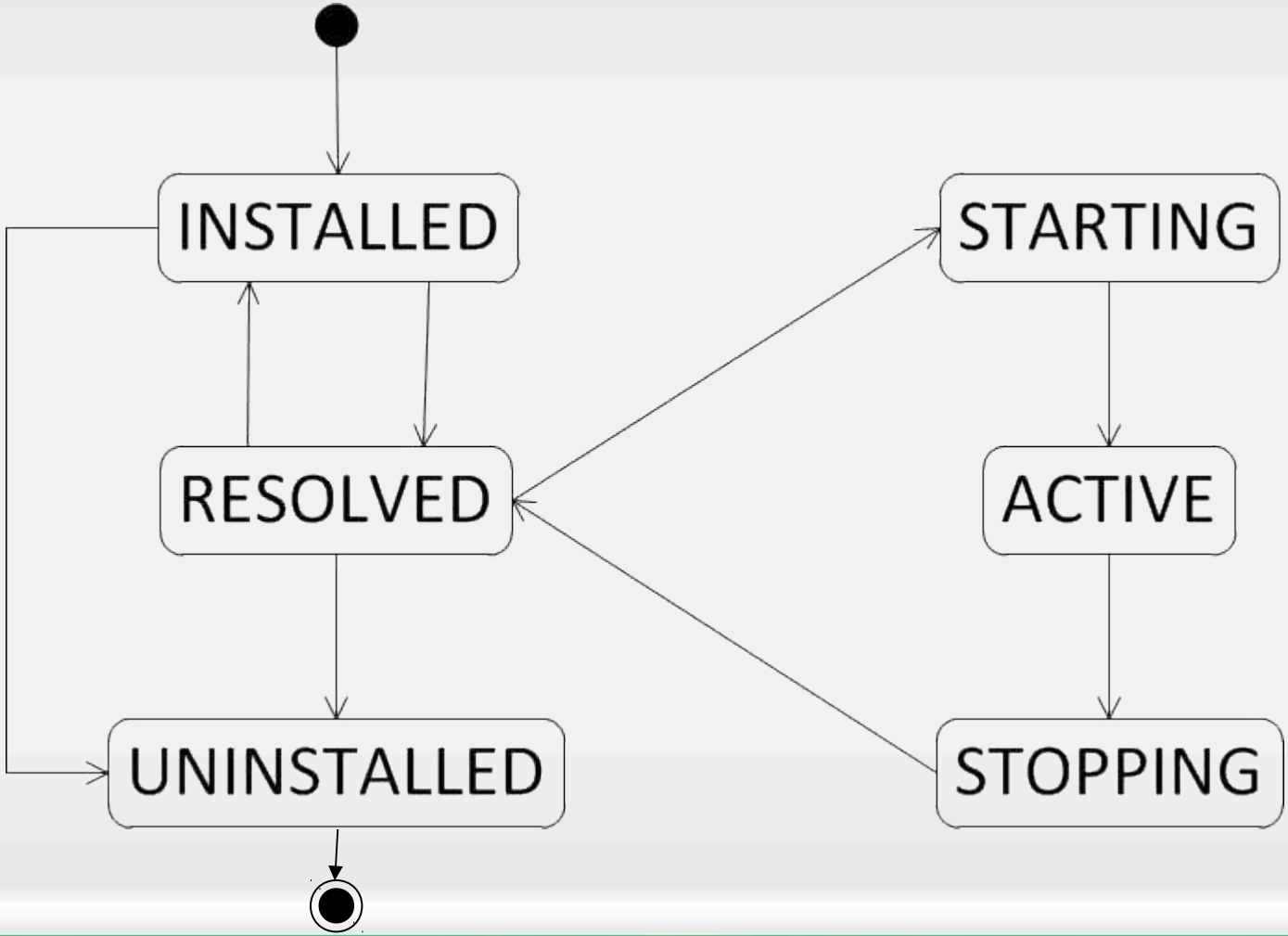
- Zależności od bundles

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-SymbolicName: pl.edu.agh.email.sender
Require-Bundle: pl.edu.agh.spell.checker,
  pl.edu.agh.addressbook
```

Wersjonowanie

- Każdy bundle określa swoją wersję
- W zależnościach możemy określić wersję lub zakres wersji zależnego pakietu lub bundle'a
- Dzięki odesparowanym przestrzeniom nazw w jednym systemie może istnieć wiele wersji tej samej klasy

Cykl życia bundle’a



Klasa aktywatora

```
public class EmailActivator implements BundleActivator {

  public void start(BundleContext context) {
    // perform some initialization here
  }

  public void stop(BundleContext context) {
    // clean up here
  }
}
```

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-SymbolicName: pl.edu.agh.email.sender
Bundle-Activator: pl.edu.agh.email.sender.Activator
```

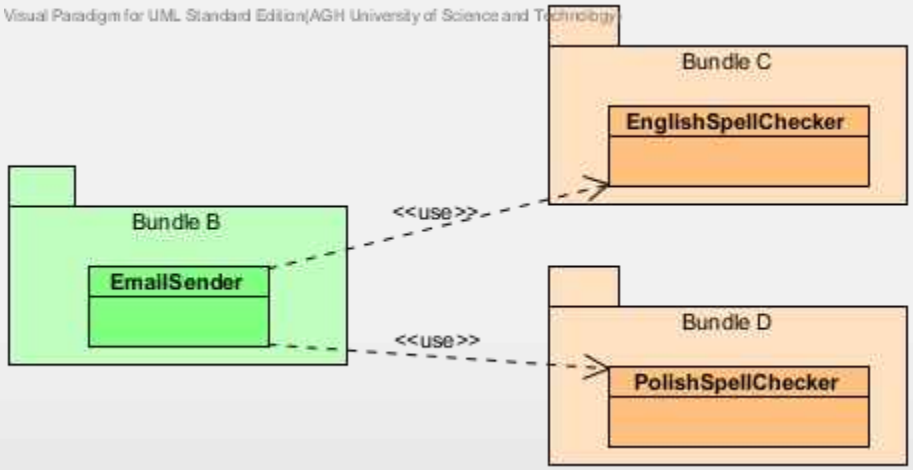
Przykład

- https://github.com/kpietak/osgi-examples
- **First bundle** – Pojedynczy bundle z własnym wątkiem (uruchomienie z konsoli)
- **Dependent bundles** – wykorzystanie klas z innego bundle'a

Serwisy w OSGi

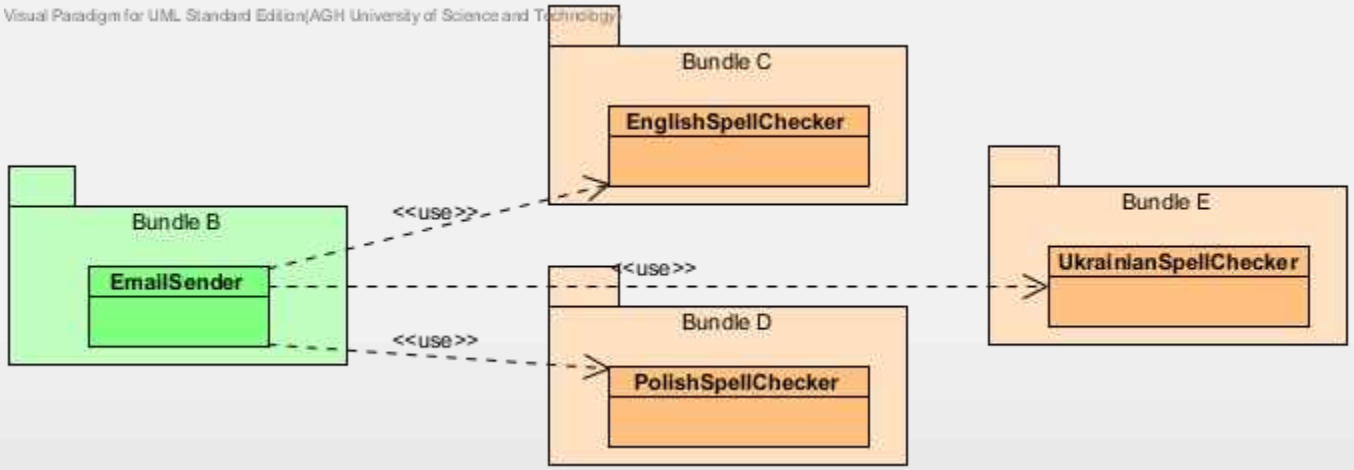
Statyczne zależności to nie wszystko...

- Ścisłe zależności pomiędzy klasami znajdującymi się w zależnych modułach:
  - Nie można usunąć pojedyńczego modułu bez naruszenia grafu zależności – *fragility*



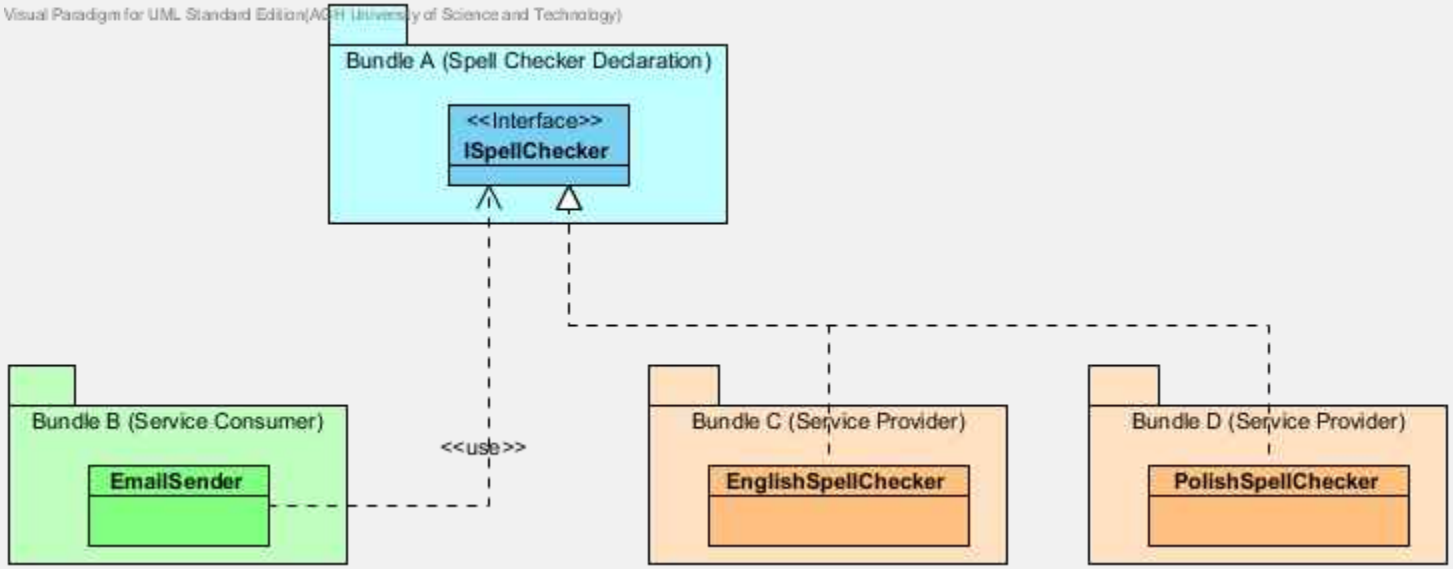
Statyczne zależności to nie wszystko...

- Ścisłe zależności pomiędzy klasami znajdującymi się w zależnych modułach:
  - Nie można dodać nowej funkcjonalności bez rekompilacji niektórych istniejących komponentów, tak aby były świadome nowych bundle'i – *lack of extensibility*



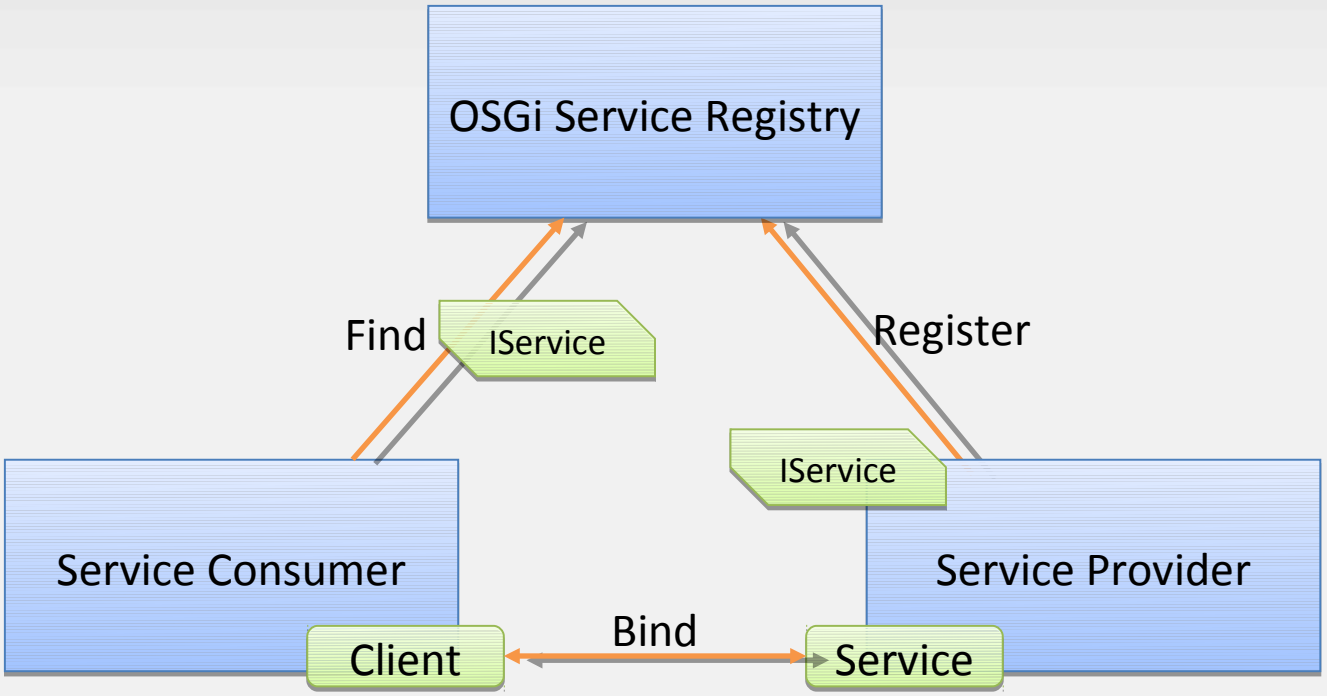
Rozwiązanie

- Separacja interfejsu od implementacji



- Mechanizm późnego wiązania (ang. *late binding*)

Dynamiczne serwisy



Serwis to POJO zarejestrowany w Service Registry

Tworzenie serwisu

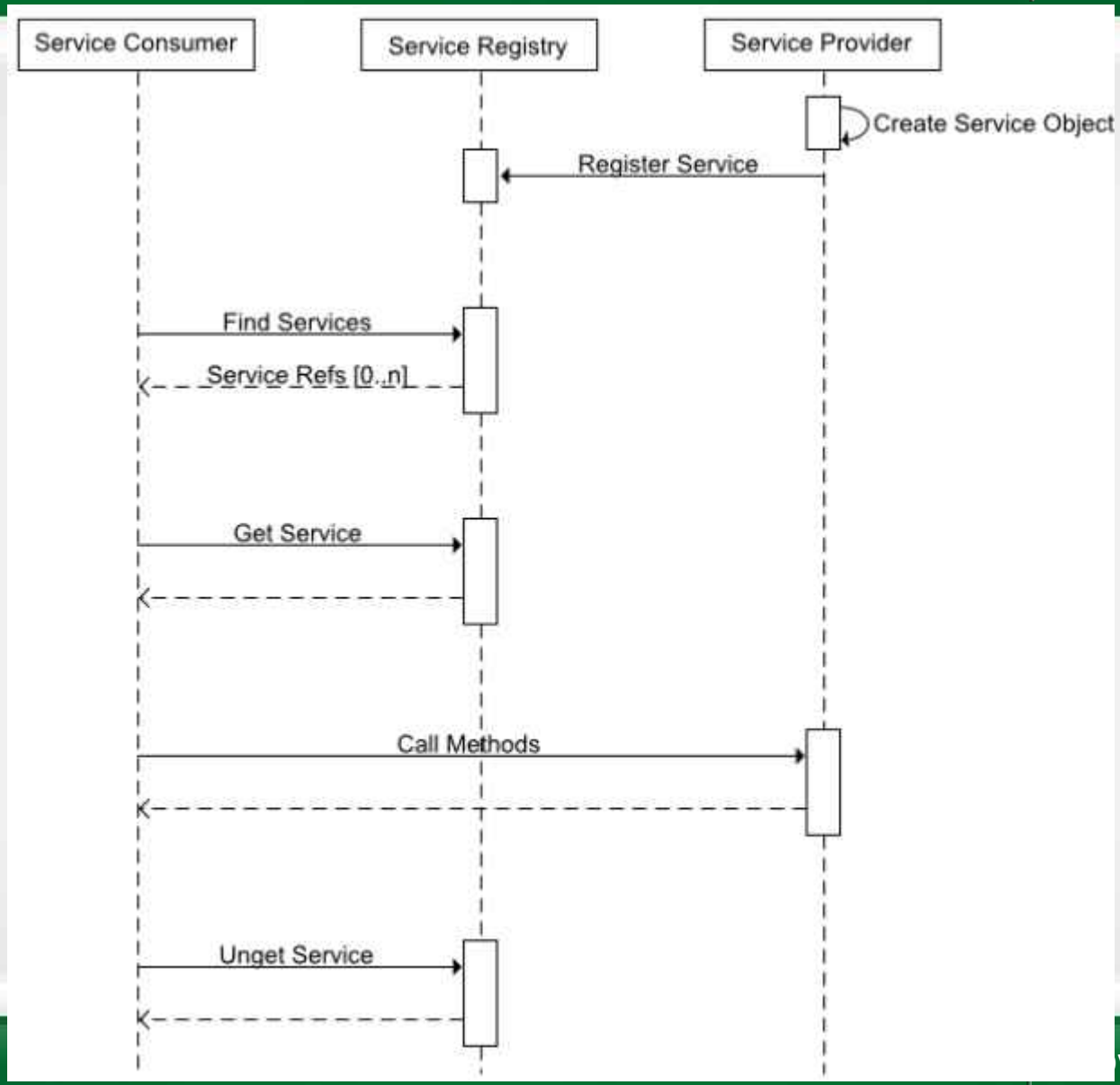
```
public void start(BundleContext context) {
   // Create the service object
   DbCustomerLookup lookup =
         new DbCustomerLookup("jdbc:mysql:localhost/customers");

   // Create the properties to register with the service
   Dictionary properties = new Hashtable();
   properties.put("dbname", "local");

   // Register the service
   context.registerService(ICustomerLookup.class.getName(), lookup,
                  properties);
}
```

Tworzenie obiektu przez bundle oraz rejestracja go jako serwis

Pobranie serwisu

```
public void start(BundleContext context) {
   this.context = context;
}

public String getCustomerName(long id) {
   ServiceReference ref = context.getServiceReference(
                     ICustomerLookup.class.getName());
   if(ref != null) {
      ICustomerLookup lookup = (ICustomerLookup)
                        context.getService(ref);
      if(lookup != null) {
         Customer cust = lookup.findById(id);
         context.ungetService(ref);
         return cust.getName();
      }
   }

   // Couldn't get name -- service not available
   return null;
}
```

Wykorzystywanie serwisów



Śledzenie serwisów

- Ręczne pobieranie serwisów – dużo nadmiarowego kodu, który odpowiada za obsługę dynamiki serwisów
- `ServiceTracker` – klasa *util*, która umożliwia śledzenie serwisu, poprzez zarejestrowanie *listener'a* (`ServiceTrackerCustomizer`), który reaguje na zdarzenia dodanie, modyfikacji i usunięcia danego serwisu

Service tracker

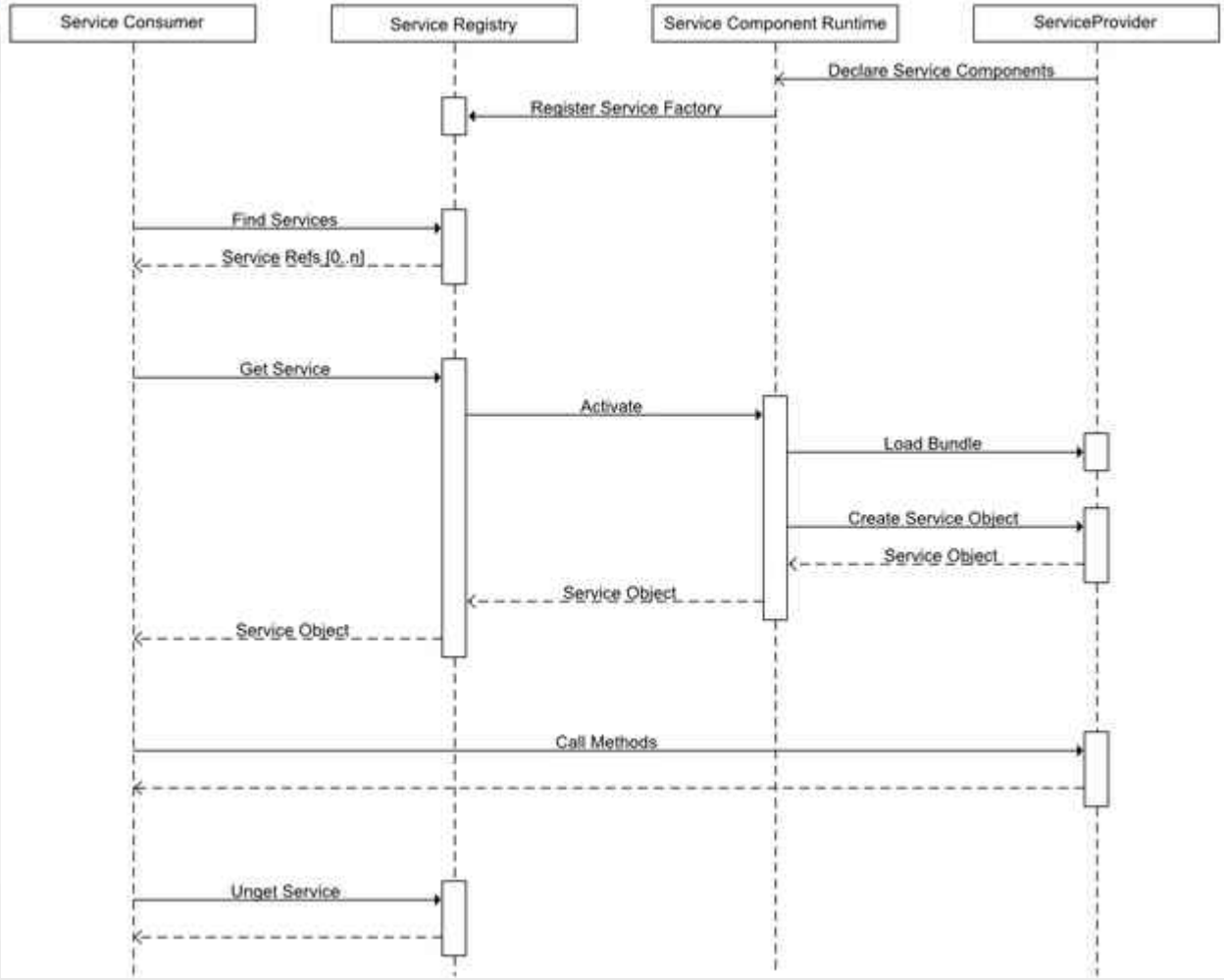
```
public void start(BundleContext context) {
   this.custLookupTracker = new ServiceTracker(context,
                        org.example.ICustomerLookup.class.getName(), null);
   this.custLookupTracker.open();
}

public String getCustomerName(long id) {
   ICustomerLookup lookup = (ICustomerLookup)
                        this.custLookupTracker.getService();
   if(lookup == null) {
      return null; // Alternatively might throw an exception
   } else {
      Customer cust = lookup.findById(id);
      return cust.getName();
   }
}
```

Declarative services

Zwykle rejestrujący serwis i oferujący serwis to ta sama jednostka (bundle)

Rzeczywista implementacja serwisu posiada XML z deklaracją

Istnieje możliwość ,,leniowego'' udostępniania serwisu (SCR tworzy proxy i instancjonuje serwis po pierwszym odwołaniu).



Implementacje OSGi

- Eclipse Equinox
  - Najpopularniejsza implementacja → podstawa środowiska Eclipse RCP
  - Lotus Notes, IBM WebSphere Application Server
- Apache Felix
  - „Najmniejsza” implementacja standardu
- Knopflerfish
  - Rozwijany przez szwedzką firmę Makewave AB

Eclipse RCP

- Framework do tworzenia aplikacji desktopowych
- Oparty o Equinox'a
- Wykorzystuje bibliotekę graficzną SWT
- Tworzony przez Eclipse Foundation
- Licencja
  - EPL (Eclipse Public License)
- Dostarcza narzędzi do wyświetlania i edycji danych
- Dostarcza mechanizmów internacjonalizacji aplikacji

Rich client platform

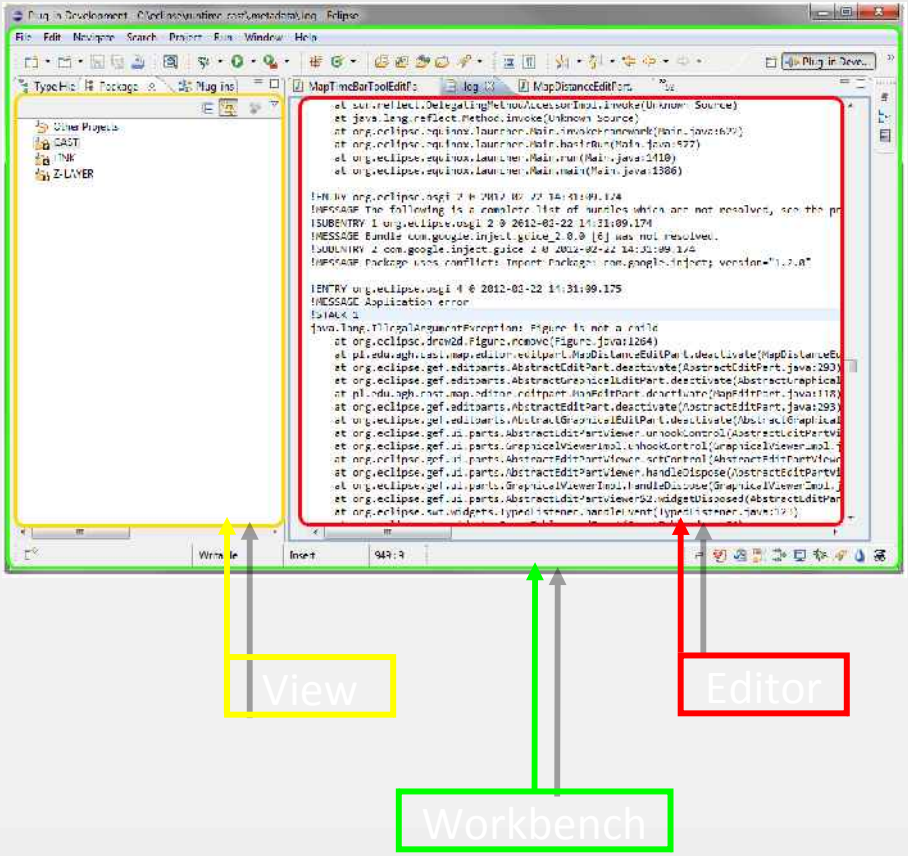
- Rich Client Platform: A Platform for building Client applications with Rich functionality
- Darmowe środowisko powstające w modelu open-source
- Eclipse 2.1 (2003) Eclipse – wyłącznie IDE. Użytkownicy budują aplikacje (również niewymagające IDE) w oparciu o Eclipse. Od Eclipse 3.0 (2004) Integracja z OSGI, wsparcie budowy niezależnych aplikacji.

Eclipse RCP a Platforma Eclipse

- Eclipse RCP:Runtime, SWT, OSGi, UI, JFace
- Eclipse RCP można określić jako minimalny zestaw pluginów wymaganych do zbudowania aplikacji typu ,,rich client".
- Eclipse Platform (powyższe, oraz: Ant, UI IDE, Search, Debug, Help, Team...)
- Eclipse Platform to platforma integrująca otwarte narzędzia programistyczne.

Eclipse RCP - pojęcia

- **Workspace** – katalog roboczy aplikacji.
- **Workbench** – obszar roboczy aplikacji opartej o Eclipse RCP. Agreguje widoki i edytory.
- **View** – wyświetlany w obrębie workbench'a. Wszystkie zmiany wprowadzone w widoku są wprowadzane automatycznie (bez konieczności wybierania „zapisz" z menu).
- **Editor** – edytor jest wyświetlany w obrębie workbench'a. Zmiany dokonane w edytorze wymagają zatwierdzenia (poprzez polecenie „zapisz").
- **Perspektywa** – sposób rozmieszczenia określonych widoków i edytora.



SWT, JFace, Draw2d

SWT:

- biblioteka graficzna dla języka Java, korzystająca z natywnych zasobów systemu operacyjnego
- dostarcza standardowych widget'ów

JFace:

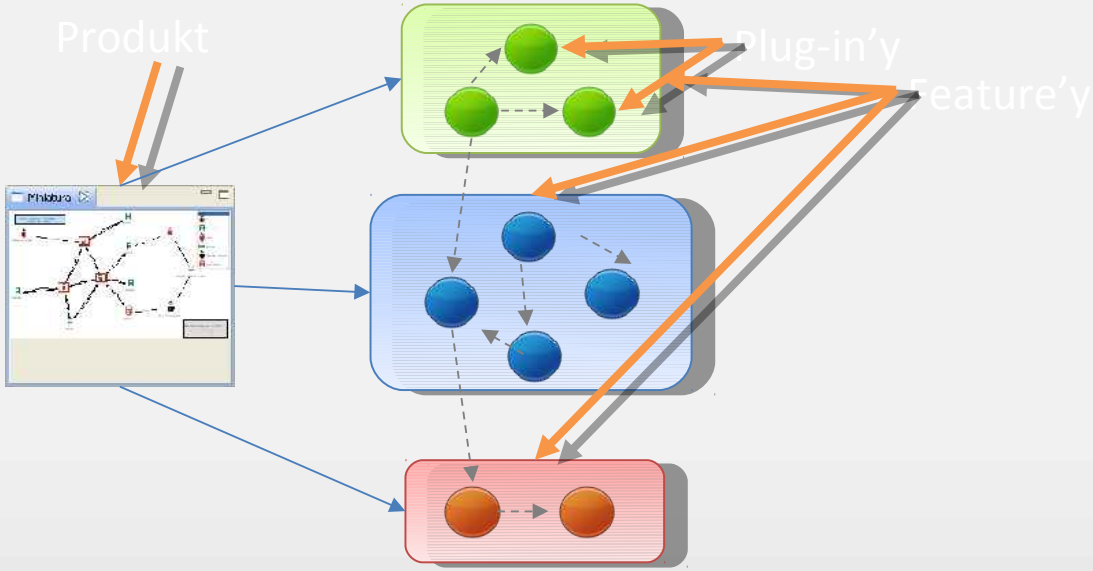
- zbiór dodatków do SWT, obejmujący zarówno nowe widgety, jak i ułatwiający korzystanie z już istniejących

Draw2d/GEF:

- „nakładka" na SWT umożliwiająca tworzenie grafiki dwuwymiarowej (przykładowo diagramów)

Equinox

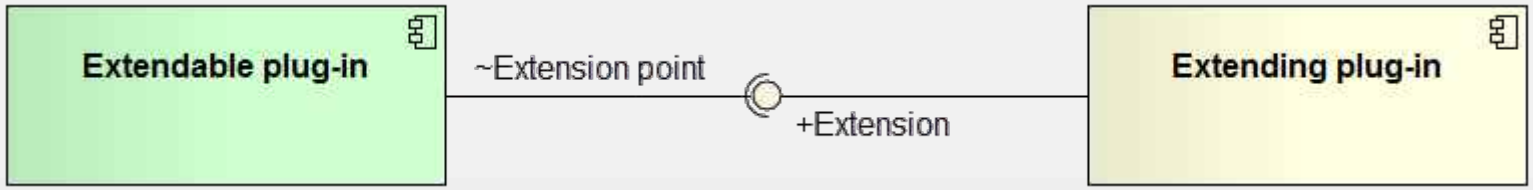
- **Plug-in (wtyczka)** – komponent dostarczający pewnej funkcjonalności. Posiada zależności (wymagane wtyczki/pakiety). Jest bytem samodzielnym.
- **Fragment** – rozszerzenie plug-in'u. W trakcie uruchomienia fragment jest „scalany" z plug-in'em rozszerzanym.
- **Feature** – agreguje wtyczki (np. względem oferowanych funkcjonalności).
- **Produkt** – definicja aplikacji



Equinox – rozszerzenia

Extension Point – określa sposób w jaki dany plug-in (=OSGi Bundle) może być rozszerzany przez inny plug-in. Posiada unikalny identyfikator. Definicje Extension Point'ów znajdują się w osobnych plikach (w formacie xml), wszystkie definicje zarejestrowane są w pliku plugin.xml. Oprócz implementacji interfejsu może posiadać deklaracje metadanych, resource'ów itp..

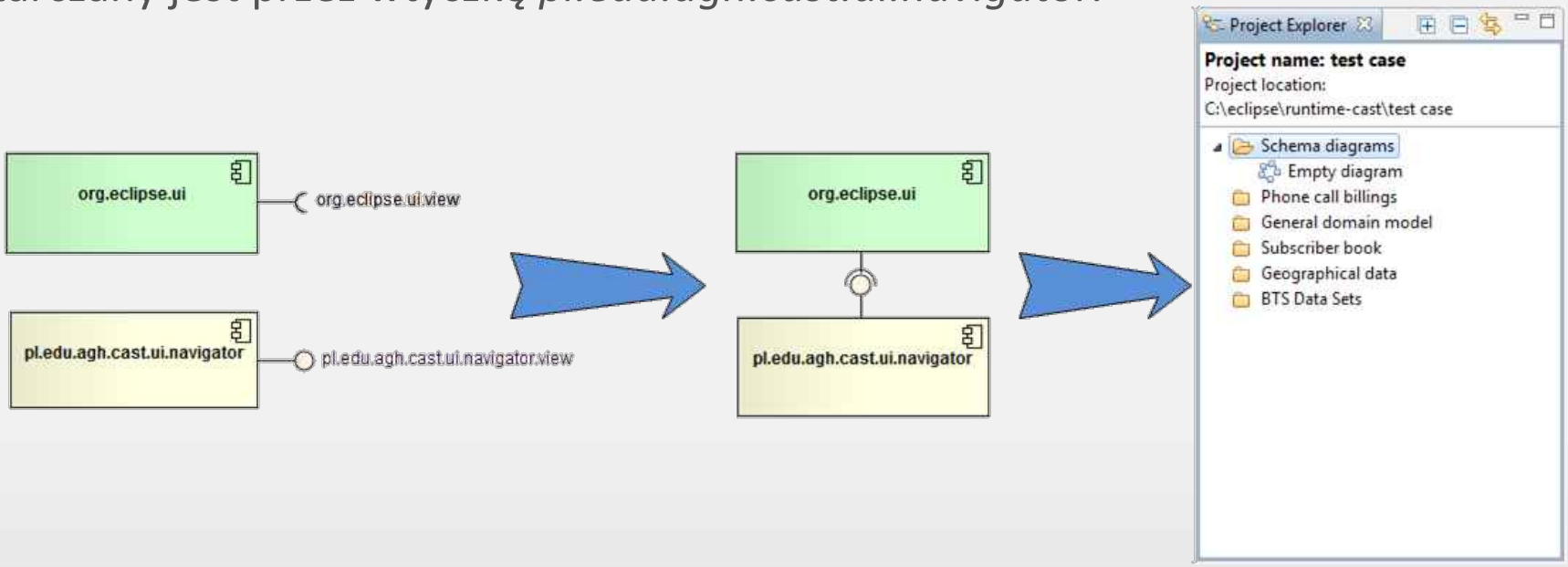
Extension – dostarcza rozszerzenia plug-in'u. Posiada unikalny identyfikator. Rejstracja Extension'ów odbywa się w pliku plugin.xml.



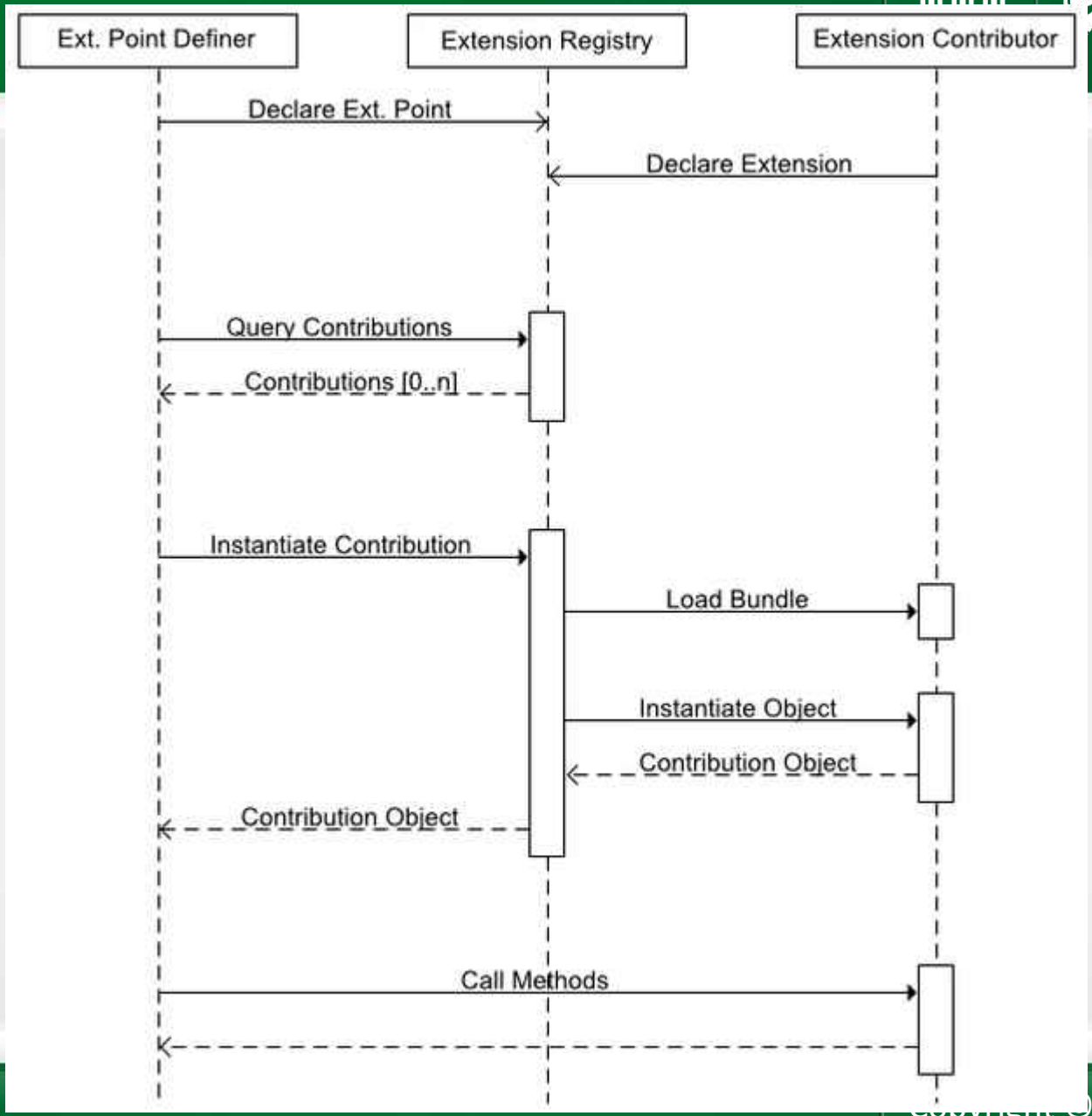
Equinox – rozszerzenia c.d.

Przykład:

- Extension point: *org.eclipse.ui.views* – określa co musi zostać podane aby można było zarejestrować nowy widok, znajduje się we wtyczce *org.eclipse.ui*.
- Extension: *pl.edu.agh.cast.ui.navigator.view* – dodaje widok nawigatora do aplikacji, dostarczany jest przez wtyczkę *pl.edu.agh.cast.ui.navigator.*



Obsługa rejestru rozszerzeń



Bibliografia

- OSGi in Practice, Neil Bartlett, *(only draft version)*
- OSGi in Action: Creating modular applications with Java, Richard Hall, Karl Pauls, Stuart McCulloch, David Savage
- OSGi Specification, http://www.osgi.org/Release4/Download
- OSGi with Eclipse Equinox – Tutorial, Lars Vogel, http://www.vogella.de/articles/OSGi/article.html