Kai Tödter, Siemens Corporate Technology
Gerd Wütherich, Freelancer
Martin Lippert, akquinet it-agile GmbH

Patterns and Best Practices for dynamic OSGi Applications

Agenda

- ***Dynamic OSGi applications***
- Basics
  - Package dependencies
  - Service dependencies
- OSGi Design Techniques
  - The Whiteboard Pattern
  - The Extender Pattern
- Conclusion

"Classic" Java applications

Java Standard Edition:

» Linear global class path

» Only one version of every library per application

» No component nor module concept above the classes level

» Totally different deployment models for different kind of environments

Java Enterprise Edition:

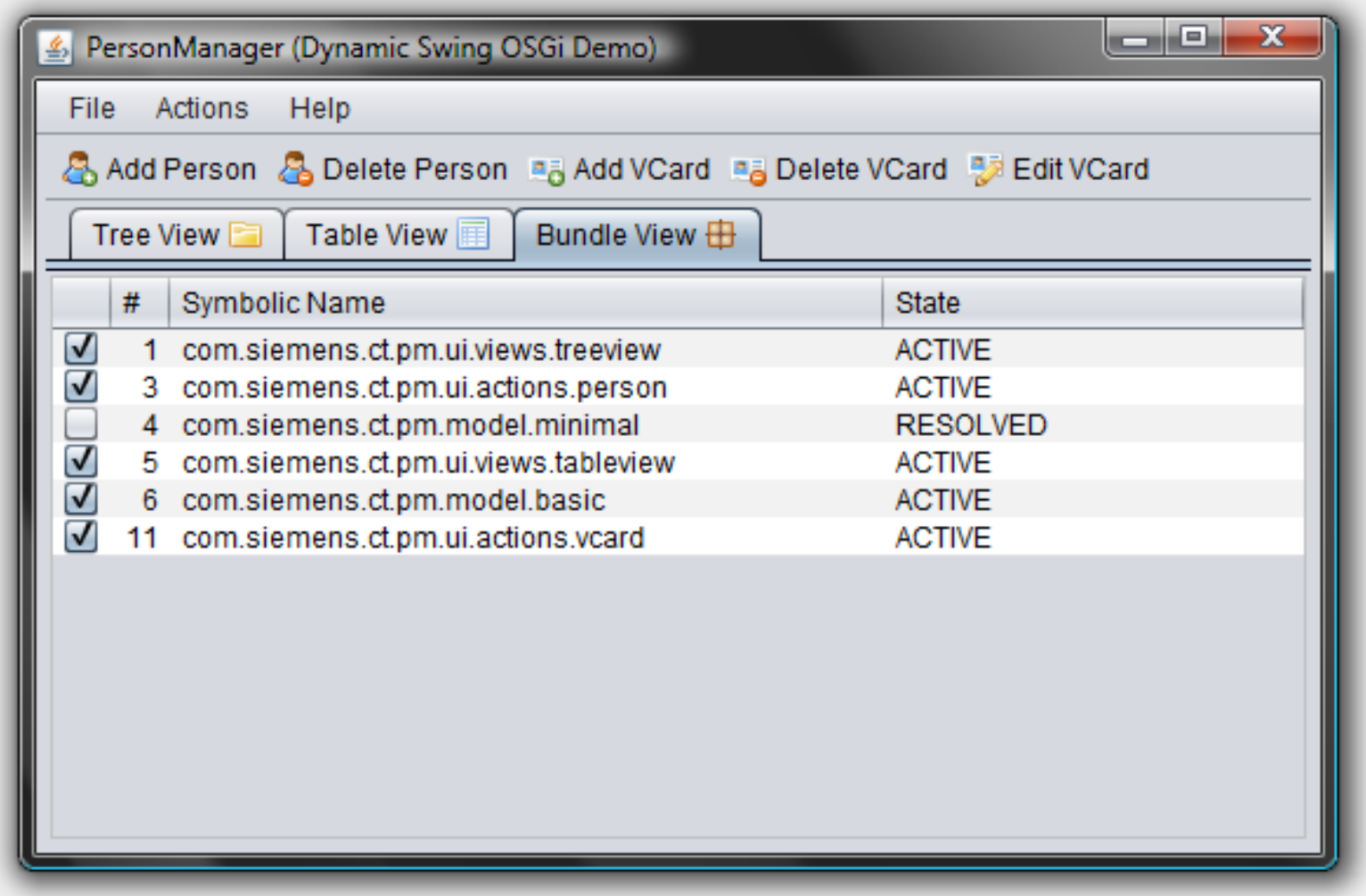
» Hot deployment possible, but requires special deployment types (e.g. WARs, RARs, EARs)

OSGi is dynamic!

Dynamic OSGi applications

- Deployment unit:
  - Bundle = JAR + additional manifest headers
- Supports dynamic scenarios (during runtime)
  - Update
  - Installation
  - Deinstallation

Dynamic Swing OSGi Demo



How to get the Demo?

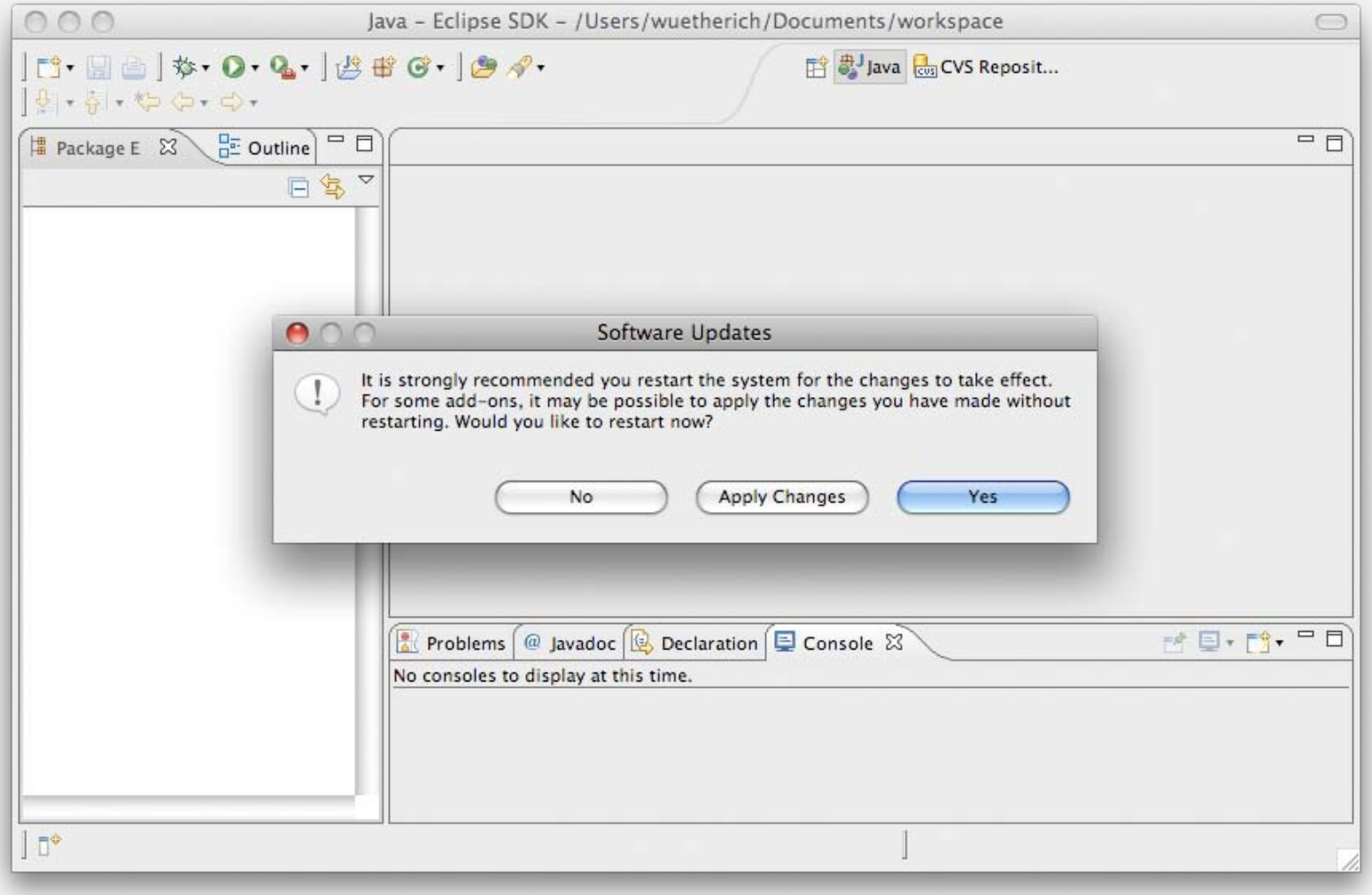
- The PM Demo project home page is: http://max-server.myftp.org/trac/pm
- There you find
  - Wiki with some documentation
  - Anonymous Subversion access
  - Trac issue tracking
- Licenses
  - All PM project sources are licensed under EPL
  - Swing Application Framework (JSR 296) implementation is licensed under LGPL
  - Swing Worker is licensed under LGPL
  - The nice icons from FamFamFam are licensed under the Creative Commons Attribution 2.5 License.

The first impressions

- "Wow - OSGi does dynamic install, uninstall and update of bundles, this is cool…"
  - I don’t need to take care of dynamics anymore
  - I don’t need to think about this at all
  - Everything is done automatically under the hood
  - Objects are changed/migrated and references to objects are managed all automatically
  - Huge bulk of magic

- **This is all wrong!!!**

If its all magic, why this?

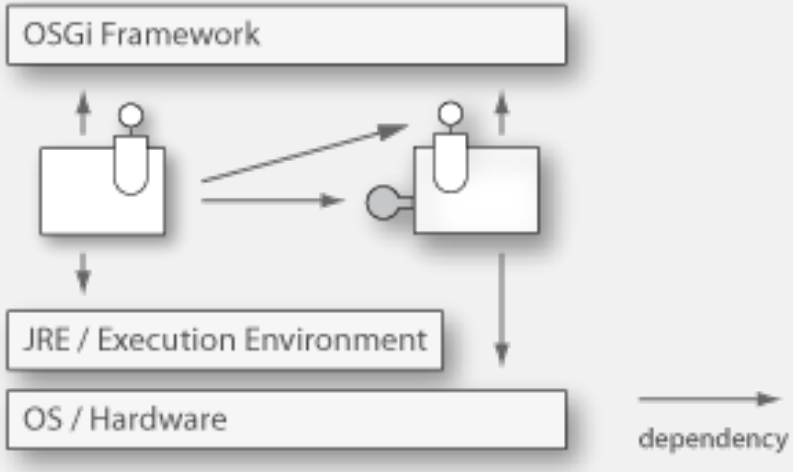


The basic idea

- OSGi controls the lifecycle of bundles
  - It allows you to install, uninstall and update bundles at runtime
  - It gives you feedback on all those actions
  - But it does not change any objects or references for you
    - "No magic"

- **OSGi gives you the power to implement dynamic applications**
- **How you use this power is up to you**

What is the problem?



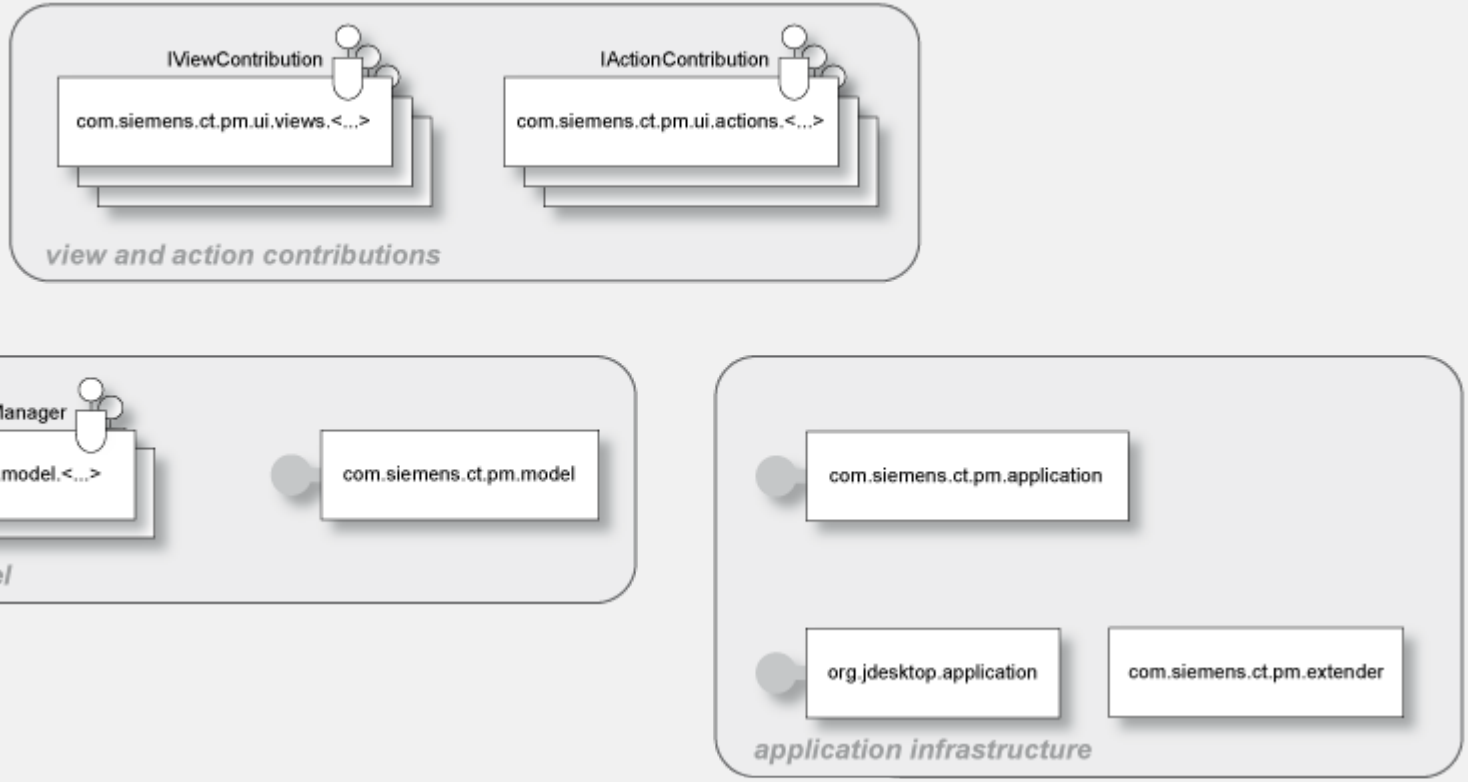
» Bundles have dependencies, e.g. package or service dependencies

» Dependencies have to be handled with respect to the dynamic behavior!

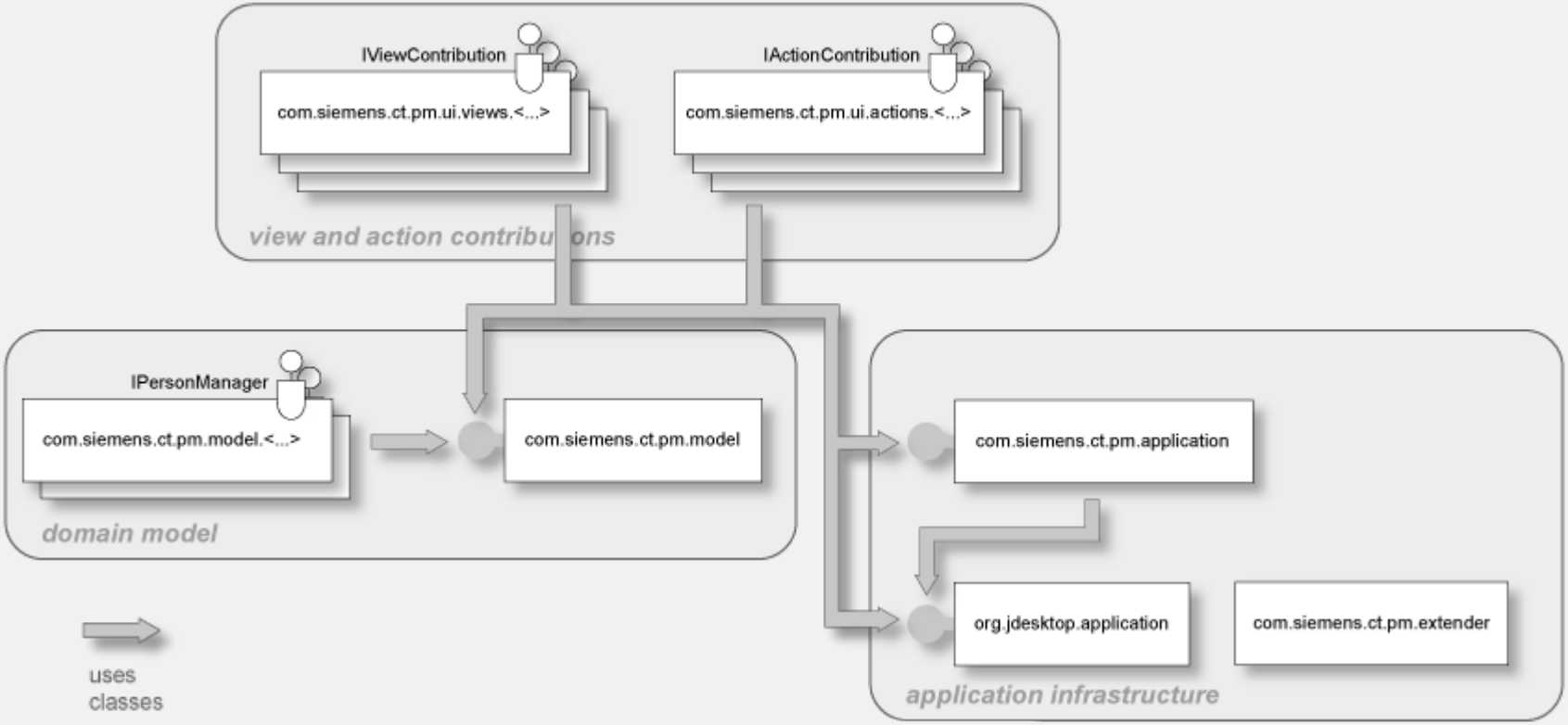
Agenda

- Dynamic OSGi applications
- Basics
  - ***Package dependencies***
  - Service dependencies
- OSGi Design Techniques
  - The Whiteboard Pattern
  - The Extender Pattern
- Conclusion

System overview



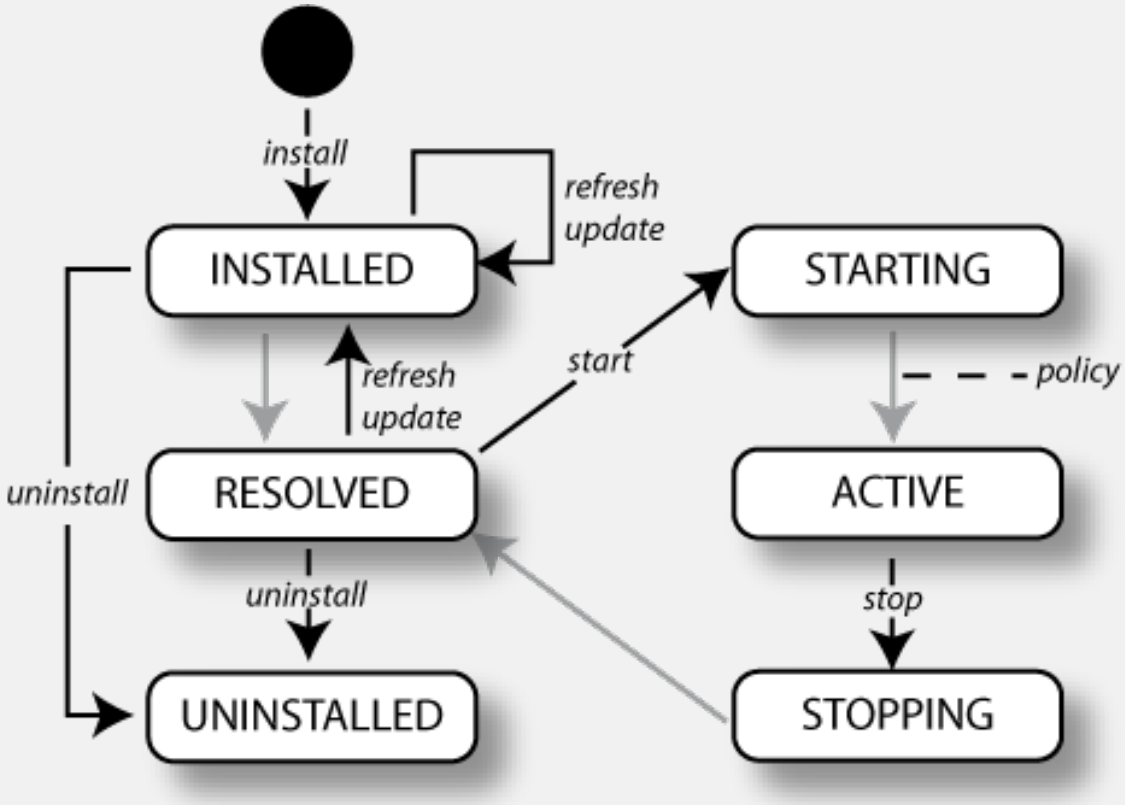
Package Dependencies



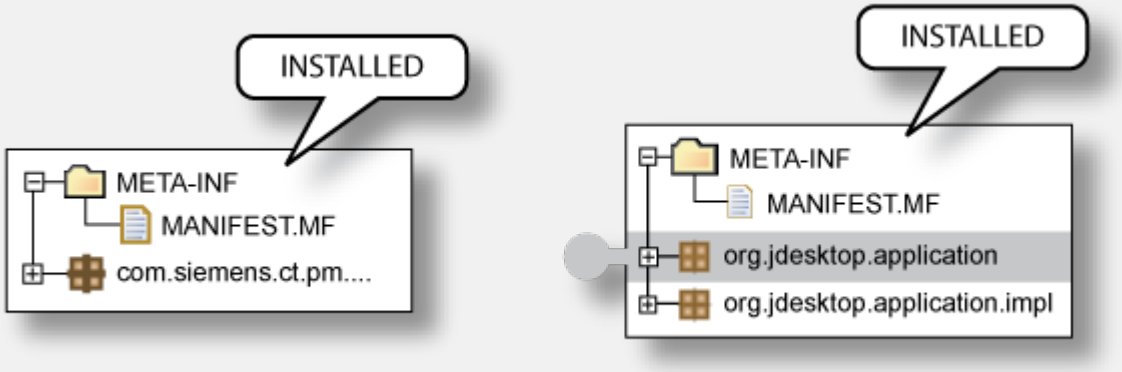
» Export of packages with **Export-Package**

» Import of packages via **Import-Package** or **Require-Bundle**

Digression: Bundle-Lifecycle

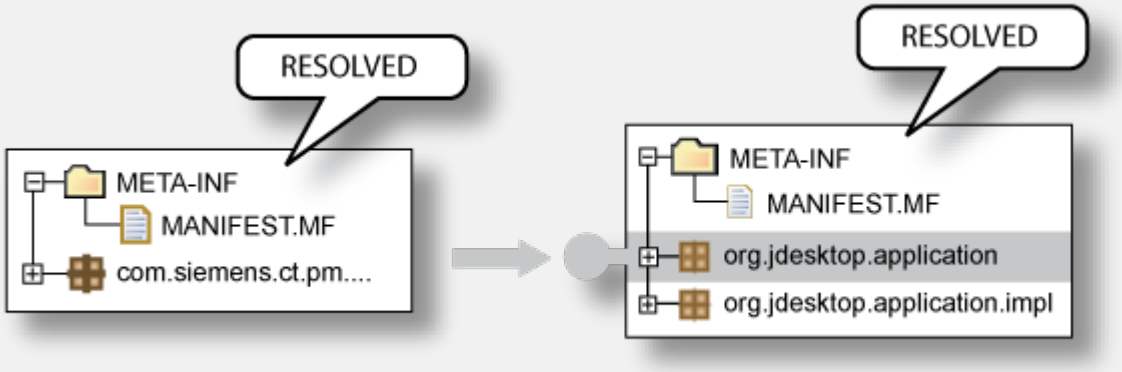


Installing



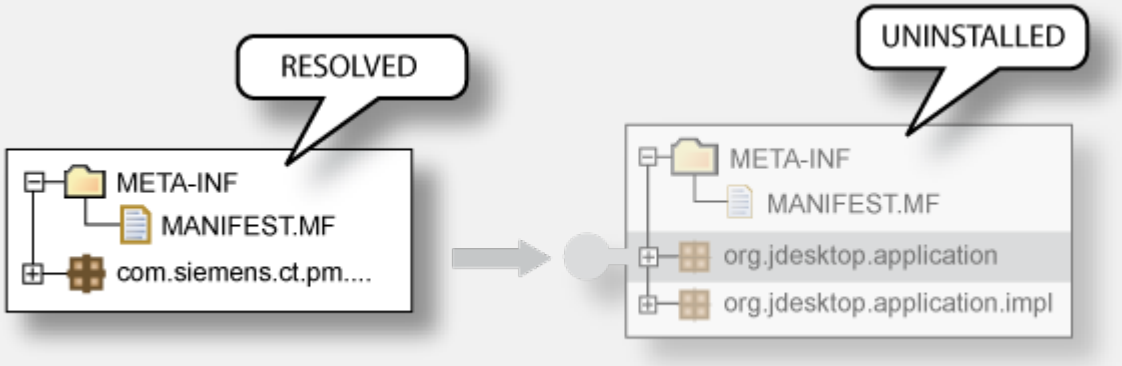
- Makes a Bundle persistently available in the OSGi Framework
  - The Bundle is assigned a unique Bundle identifier (long)
  - The Bundle State is set to INSTALLED
  - The Bundle will remain in the OSGi Framework until explicitly uninstalled

Resolving



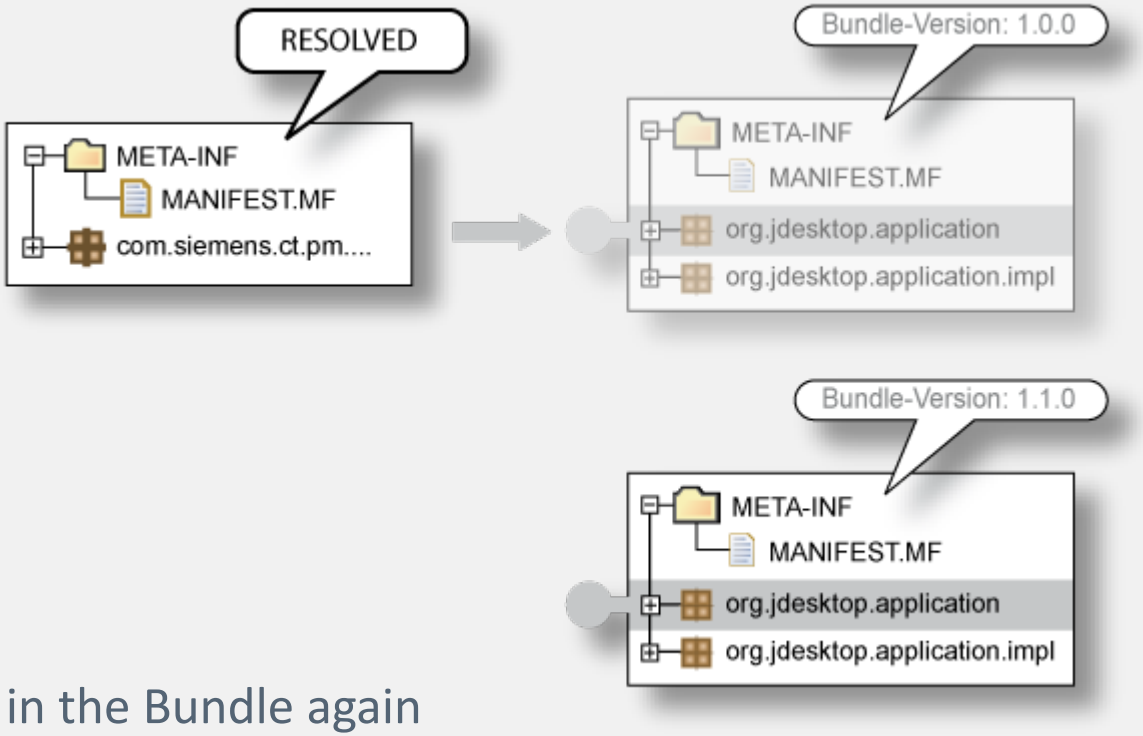
- Wires bundles by matching imports to exports
- Resolving may occur eagerly (after installation) or lazily
- There is no API for resolving
- After resolving -> Bundle is in state RESOLVED

Uninstall



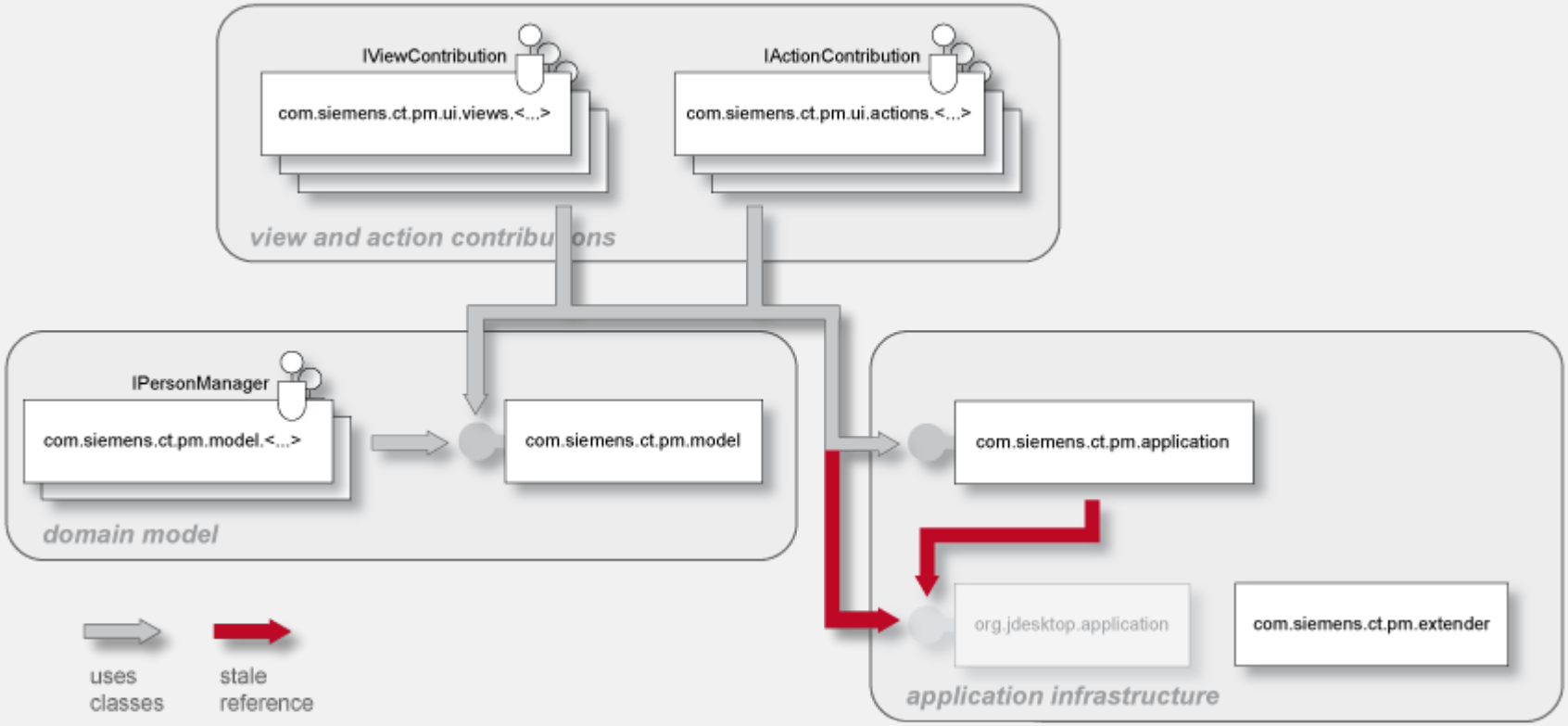
- … removes a Bundle from the OSGi Framework
- The Bundle State is set to UNINSTALLED
- If the Bundle is an exporter: Existing wires will remain until
  - the importers are refreshed or
  - the OSGi Framework is restarted

Update and Refresh



- Update:
  - Reads in the Bundle again
  - If the Bundle is an exporter: Existing wires will remain until the importers are refreshed or the OSGi Framework is restarted
- Refresh:
  - All the bundle dependencies will be resolved again

What does this mean?



» Update or uninstall of bundles can lead to stale package references

» Refresh -> restart of the bundles

We need to re-think designs

- Just modularizing into bundles with clearly defined package dependencies is not enough!

- We need to think about dynamics while building the system
- We need to think even more about dependencies
- We need to re-think typical well-known designs
  - More will follow

Best Practices: Package Dependencies

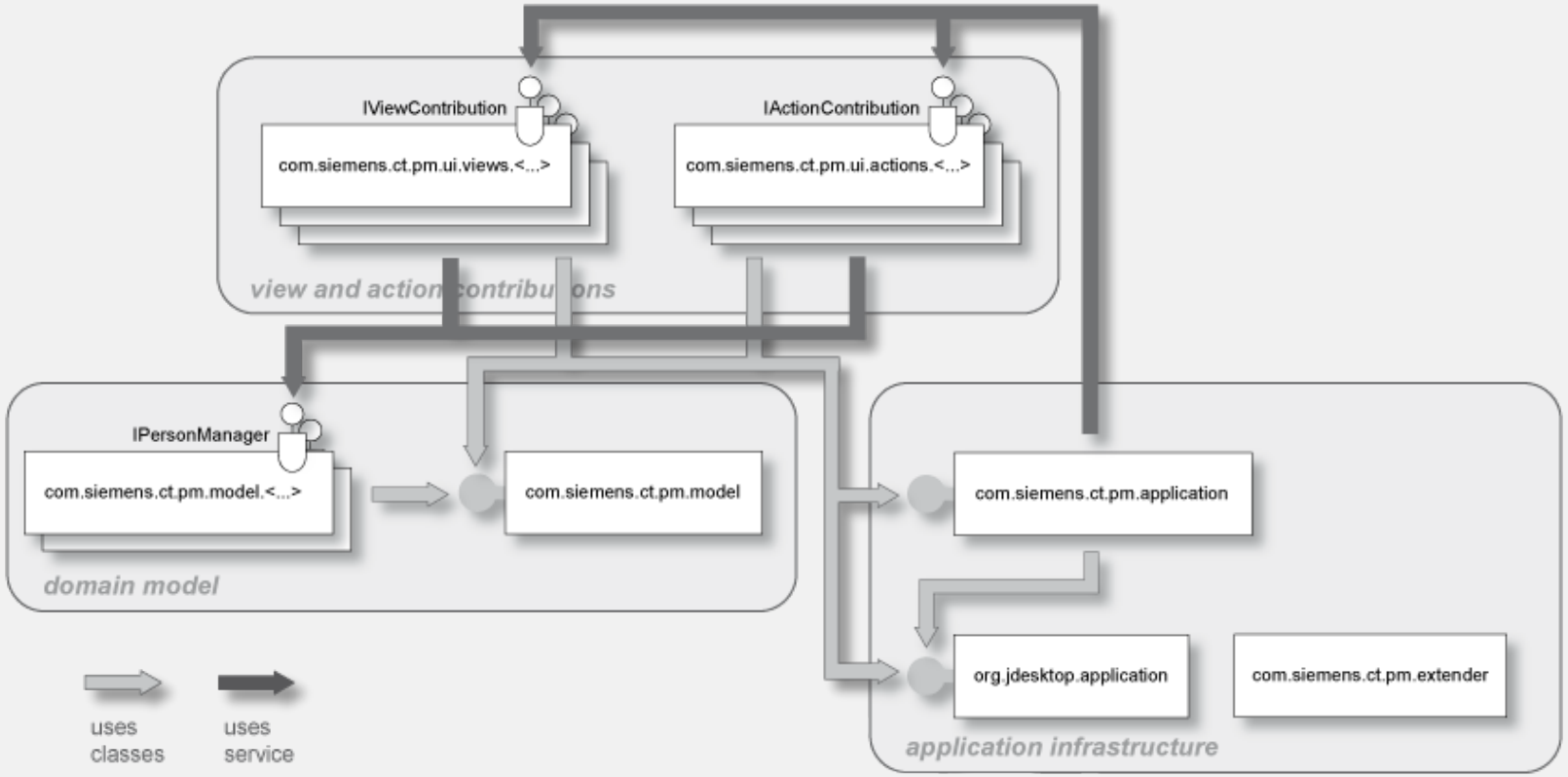
- Only import packages that are really used/needed
- Use Import-Package rather Require-Bundle
- Only use Require-Bundle when it comes to split-packages
  - This is the unfortunately the case in many bundles of the Eclipse platform!

- -> Reduce coupling

Agenda

- Dynamic OSGi applications
- Basics
  - Package dependencies
  - ***Service dependencies***
- OSGi Design Techniques
  - The Whiteboard Pattern
  - The Extender Pattern
- Conclusion

Service dependencies



» One way to reduce coupling
  » Split interface and implementation into different bundles
  » Lookup implementation(s) dynamically

ServiceListener / ServiceTracker

- But be careful:
  - If you lookup a service implementation, you get the direct reference to that object
  - If the implementing bundle goes away, you need to be careful not to keep this object referenced

- ServiceListener / ServiceTracker help you
  - ServiceListener: calls you back if something changes
  - ServiceTracker: listens to service listener events for you (less code than using service listeners manually)

Declarative (and other) Approaches

- **Declarative Services**
  - Part of the OSGi specification, declarative description of services with XML
- **Spring Dynamic Modules**
  - Spring goes dynamic with help of OSGi
  - http://www.springframework.org/osgi
- **iPojo**
  - “Original” DI framework for OSGi
  - http://ipojo.org
- **Guice - Peaberry**
  - Guice: Performant, lightweight DI Framework
  - Peaberry: Extension of Guice for OSGi
  - http://code.google.com/p/peaberry/
  - http://code.google.com/p/google-guice/

Declarative Services (DS)

- DS is part of the OSGi R4 spec
- DS let you declare components in xml
- The declarations live in OSGI-INF/<component>.xml
- Components can provide services
- Components can depend on other services
  - Uses dependency injection for references to other services:
  - These services are bound to defined bind/unbind methods in the components
  - A cardinality and a creation policy can be defined

PM Example DS Component

```
<component name="pm.ui.actions.person.ActionContribution">
    <implementation
        class="pm.ui.actions.person.ActionContribution"/>
    <service>
       <provide interface=
            "pm.application.service.IActionContribution"/>
    </service>
    <reference name="PersonManager"
        interface="pm.model.IPersonManager"
        bind="setPersonManager"
        unbind="removePersonManager"
        cardinality="0..1"
        policy="dynamic"/>
</component>
```

Spring Dynamic Modules (DM)

- Integration of Spring and OSGi
- Implemented using Extender pattern
- XML files live in META-INF/spring
- Best Practice: Two XML files
  - One to define a Spring bean
  - One to map this bean to an OSGi service
- Uses Spring dependency injection for references to other services
- Similar but more flexible/powerful approach compared to DS
  - But needs 15 additional Spring and logging bundles to run

PM Example Spring DM Component

XML for Spring Bean:

```
<beans (Schema attributes omitted)>
  <bean name="savePerson"
        class="pm.ui.actions.save.ActionContribution"/>
</beans>
```

XML for OSGi service mapping:

```
<beans (Schema attributes omitted)>
<osgi:service id="savePersonOSGi" ref="savePerson"
  interface="pm.application.service.IActionContribution"/>
</beans>
```

iPOJO

- Part of Felix, the Apache OSGi implementation
- Maven and Ant integration (also Eclipse Plug-in available)
- Supports both XML and Java annotations
  - Component instances have to be specified in XML
- Similar approach compared to DS
- Manipulates the bundle jar file
  - Extra build step necessary
  - Makes development using Eclipse tedious

PM Example iPOJO Component

XML for iPOJO component:

```
<ipojo>
<component
  classname="com.siemens.ct.pm.ui.views.treeview.ipojo.TreeView"
  name="TreeView">
    <requires>
            <callback type="bind" method="setPersonManager" />
            <callback type="unbind" method="removePersonManager" />
    </requires>
    <provides />
</component>
<instance component="TreeView" />
</ipojo>
```

PM iPOJO Annotation Example

```
@Component(name = "AnnotatedTreeView")
@Provides
public class AnnotatedTreeView implements
  IViewContribution, IPersonListener {
   ...

@Bind
public synchronized void
  bindPersonManager(IPersonManager
  personManager) {
   ...
```

Guice / Peaberry

- Based on Google Guice dependency injection framework
- Uses builder pattern
- Pure Java code
- Small footprint
- In version 1.0, OSGi service lifecycle management could be improved
  - Will be in version 1.1

PM Guice / Peaberry Example

```
Injector inj = createInjector(osgiModule(context),
   new AbstractModule() {
      @Override
      protected void configure() {
          bind(IPersonManager.class).toProvider(
             service(IPersonManager.class)
             .out(new PMScope()).single());
          bind(export(IViewContribution.class)).toProvider(
              service(treeView).export());
      }
    });
inj.injectMembers(this);
```

Best Practices: Services

- Use a ServiceTracker
  - Don’t do all the service getting manually
  - Service tracker help you with dynamically coming and going services
- Better: Use declarative approaches!
  - Either DS or Spring DM
  - Both help you with service dependencies and dependency injection

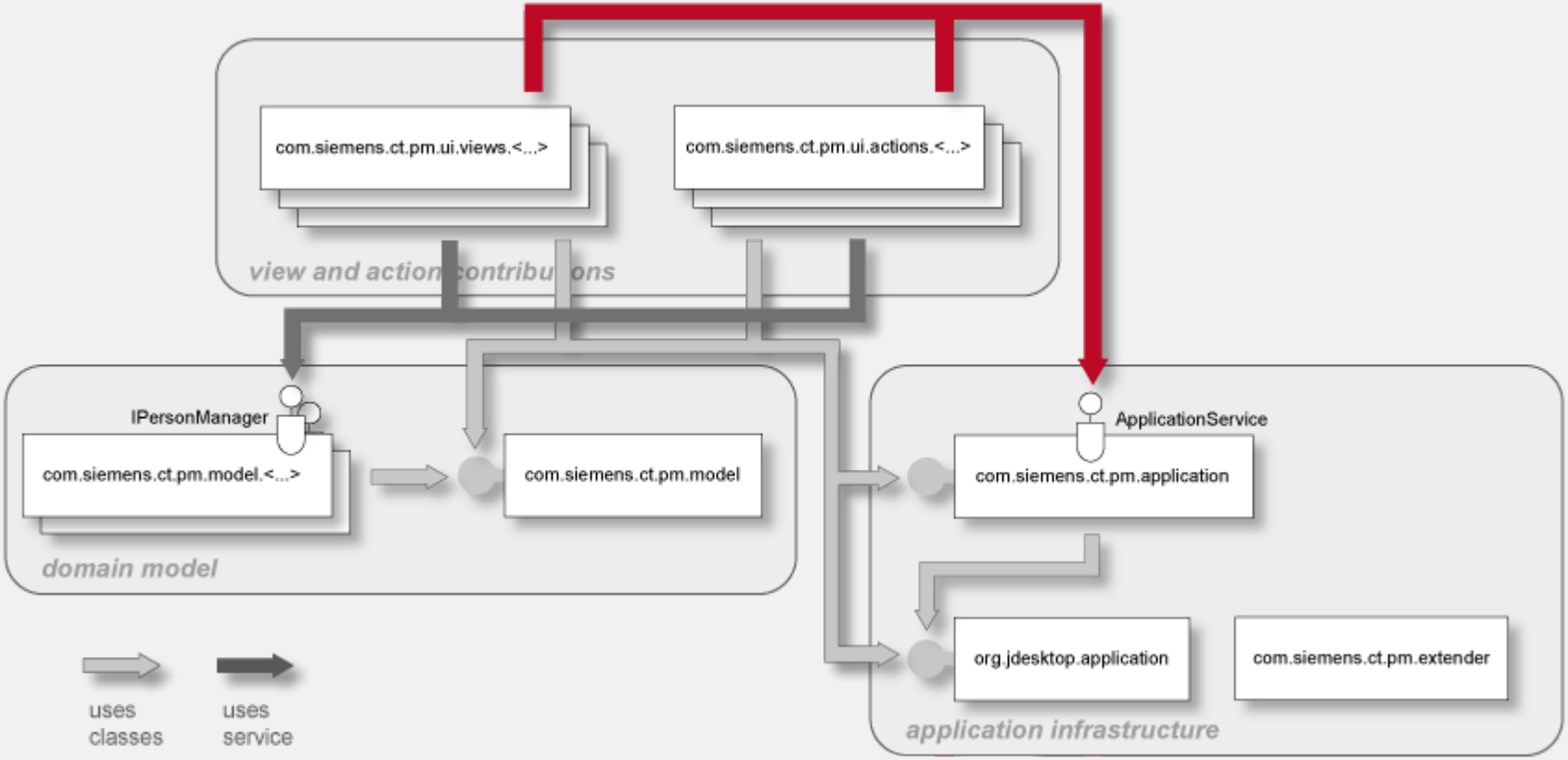
Agenda

- Dynamic OSGi applications
- Basics
  - Package dependencies
  - Service dependencies
- OSGi Design Techniques
  - ***The Whiteboard Pattern***
  - The Extender Pattern
- Conclusion

The Whiteboard-Pattern

- Problem:
  Often a service provides an implementation of the publisher/subscriber design pattern and provides methods to register listeners for notifications
- The OSGi service model provides a service registry with these notification mechanisms already!
- So:
  - Don’t get a service and register as listener
  - Be a service yourself and register with the OSGi service registry!

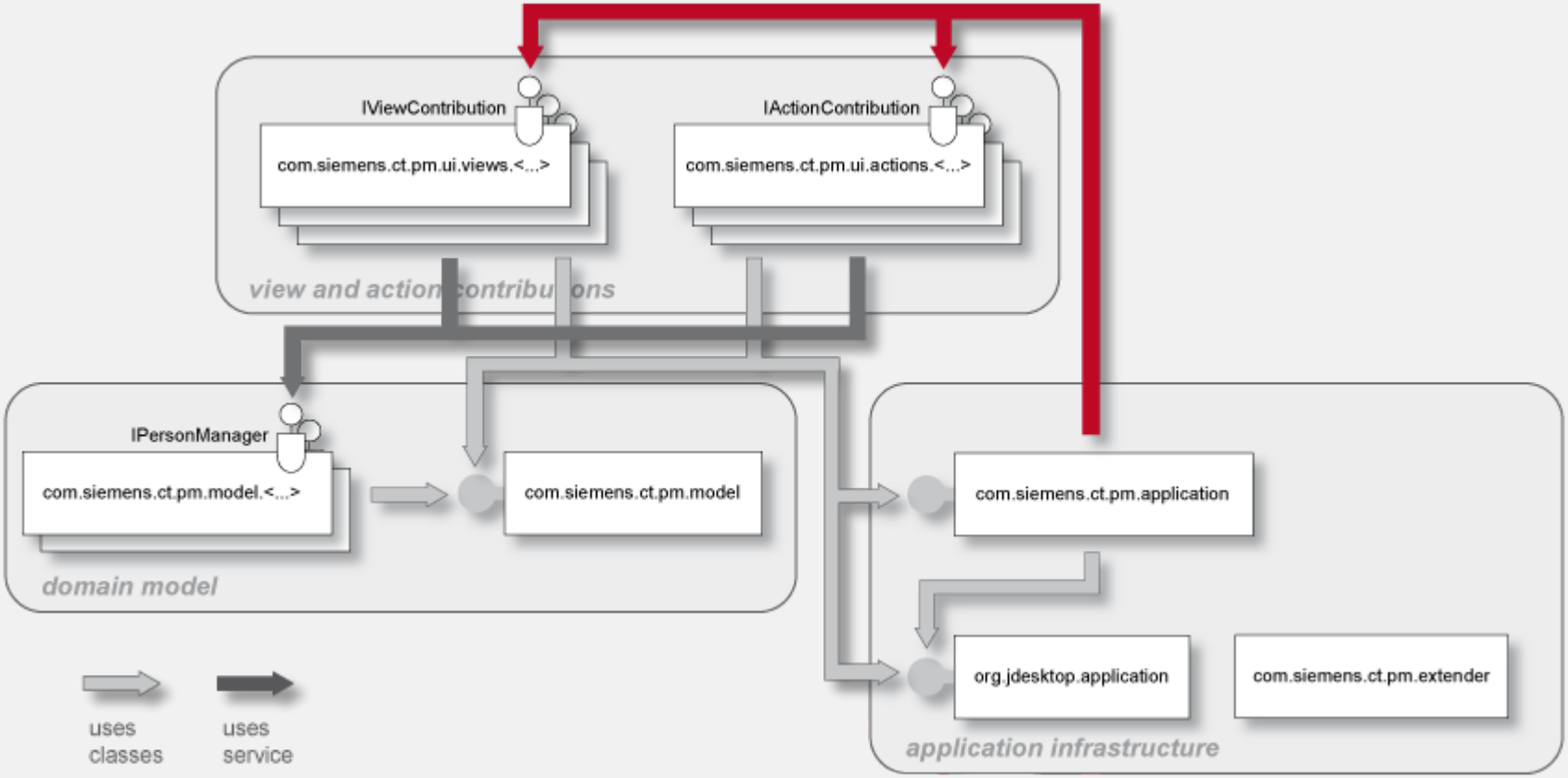
Example: The Listener Pattern



» Clients use ApplicationService to register view and action contributions

» Client is responsible for handling dynamic behavior

Example: The Whiteboard Pattern



» Clients register view and action contributions as services

» Application manager is responsible for handling dynamic behavior

Whiteboard Pattern in PM Demo

- The Action and View contribution managers are NOT services
  - Instead, they are wrapped in a DS component
- All action and view contributions are OSGi services and implement
  - IActionContribution
  - IViewContribution
- Take a look at the bundles
  - com.siemens.ct.pm.application
  - com.siemens.ct.pm.ui.actions.*
  - com.siemens.ct.pm.ui.views.*

Agenda

- Dynamic OSGi applications
- Basics
  - Package dependencies
  - Service dependencies
- OSGi Design Techniques
  - The Whiteboard Pattern
  - ***The Extender Pattern***
- Conclusion

The Extender Pattern

- The **extender pattern** allows bundles to extend the functionality in a specific domain
- It uses the synchronous bundle listener
- The extender adds a bundle listener to the BundleContext
- The bundle listener overwrites
  `public void bundleChanged(BundleEvent event)`
- Then the listener checks the started bundle for a specific handler and performs some (domain)specific action
- The extender should also check all already started bundles in its activator

PM Demo Extender: Registering Services

- The following example shows a demo extender
- Implemented in `com.siemens.ct.pm.extender`
- Registers a bundle listener
- Looks for the manifest header "Action-Contribution" in every bundle
- When found in a started bundle
  - Parses the value as class name
  - Registers the class as service implementation for `com.siemens.ct.pm.application.service.IActionContribution`
- When found in a stopped bundle
  - Unregisters the service

PM Demo Extender (1)

```
public class Activator implements BundleActivator,
  SynchronousBundleListener {

    public void start(BundleContext context)
                                          throws Exception {
        context.addBundleListener(this);
        // search for existing bundles
        Bundle[] bundles = context.getBundles();
        for (Bundle bundle : bundles) {
        if (Bundle.ACTIVE == bundle.getState()) {
             greet(bundle);
        }
    }
}
```

PM Demo Extender (2)

```
public void stop(BundleContext context) throws Exception {
    context.removeBundleListener(this);
}

// React on bundle events
public void bundleChanged(BundleEvent event) {
    if (BundleEvent.STARTED == event.getType()) {
        addService(event.getBundle());
    } else if (BundleEvent.STOPPED == event.getType()) {
        removeService(event.getBundle());
    }
}
```

PM Demo Extender (3)

```
private void addService(Bundle bundle) {
  String className =
       (String) bundle.getHeaders().get("Action-Contribution");
  try {
    if (className != null) {
       Class clazz = bundle.loadClass(className);
       ServiceRegistration serviceRegistration =
           context.registerService("pm.service.IActionContribution",
           clazz.newInstance(), null);
       serviceMap.put(bundle.getSymbolicName(),
                      serviceRegistration);
    } catch (Exception e) {
       // Catch all Exceptions
    }
  }
}
```

PM Demo Extender (4)

```
private void removeService(Bundle bundle) {
    ServiceRegistration serviceRegistration =
        serviceMap.remove(bundle.getSymbolicName());

    if (serviceRegistration != null) {
        serviceRegistration.unregister();
    }
}
```

Discussion