Patterns and Best Practices for Dynamic OSGi Applications

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

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)
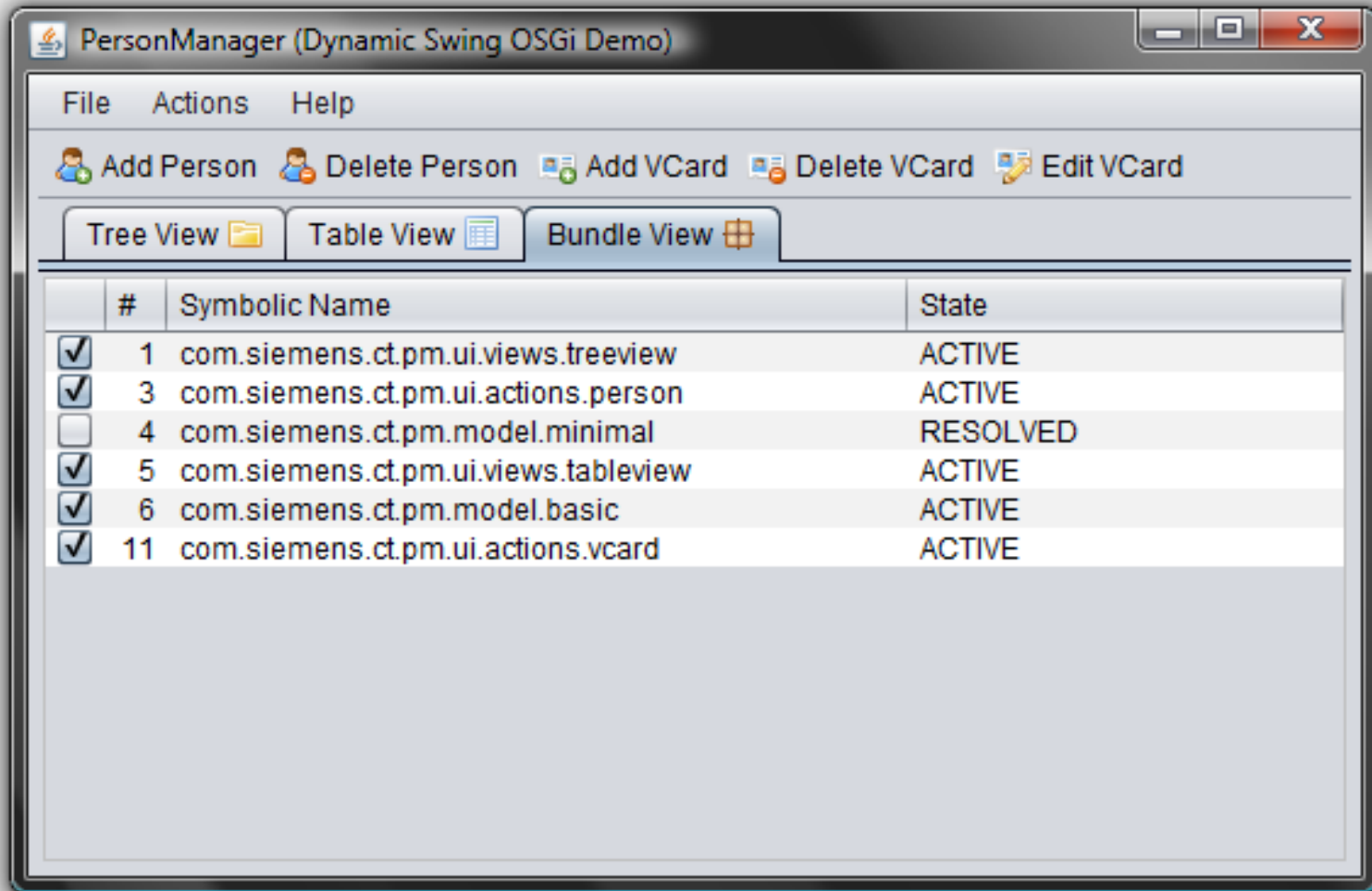
And the result is…

Now we have OSGi

» "OSGi - the dynamic module system for Java"

» You can define:
  » Bundles (aka Modules)
  » Dependencies
  » Visibilities

» **This is a huge step forward !!!**

And OSGi is dynamic!

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?
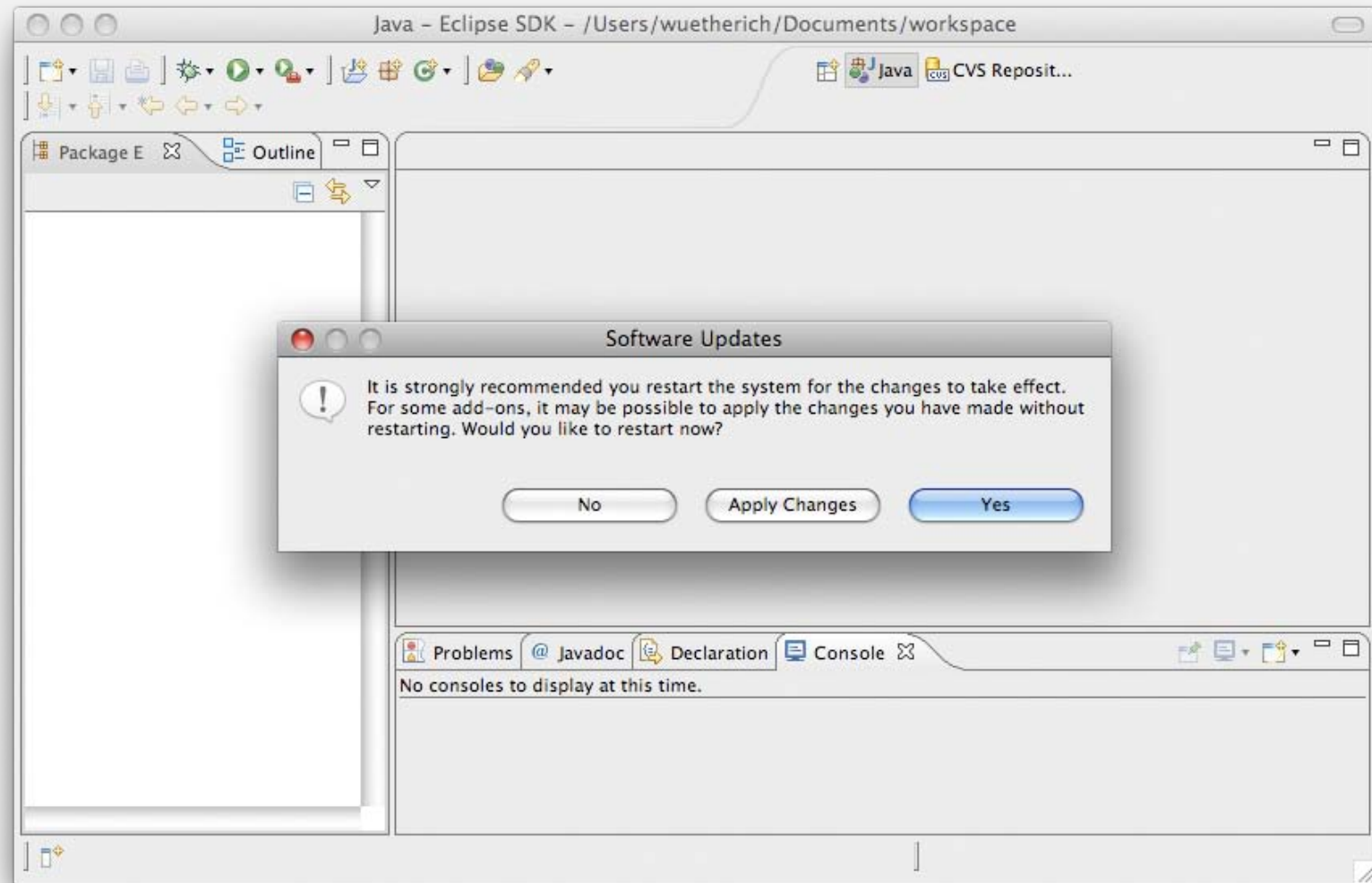
The basic idea

» 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 challenge?

OSGi Framework

JRE / Execution Environment

OS / Hardware
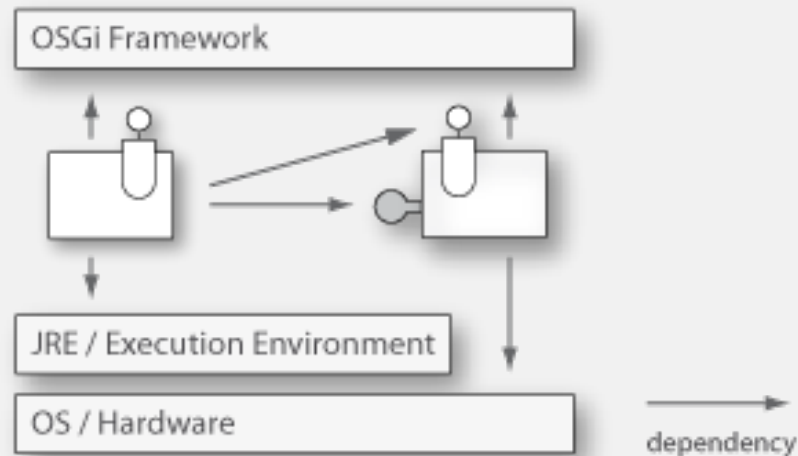
dependency

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

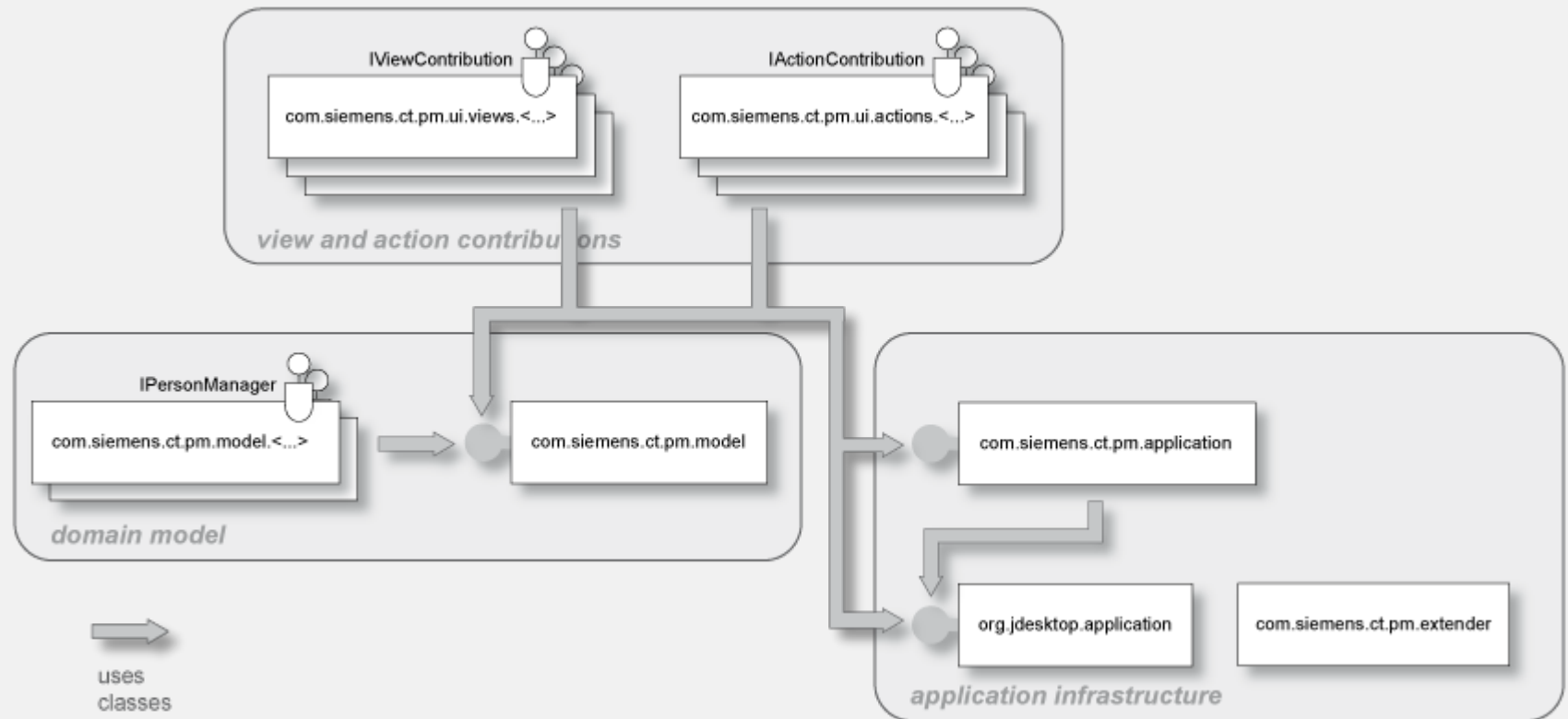» Dependencies have to be handled with respect to the dynamic behavior!

Agenda

» Dynamic OSGi applications

» Basics

>> » *Package dependencies*

>> » Service dependencies

» OSGi Design Techniques

>> » The Whiteboard Pattern
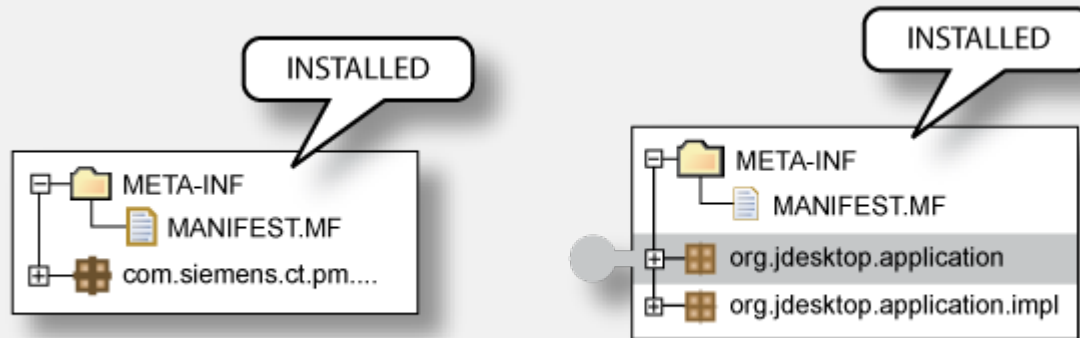
>> » The Extender Pattern

» Conclusion

# System overview
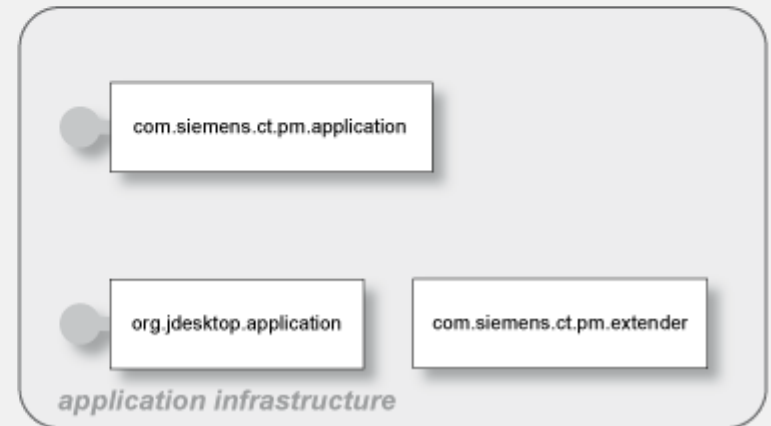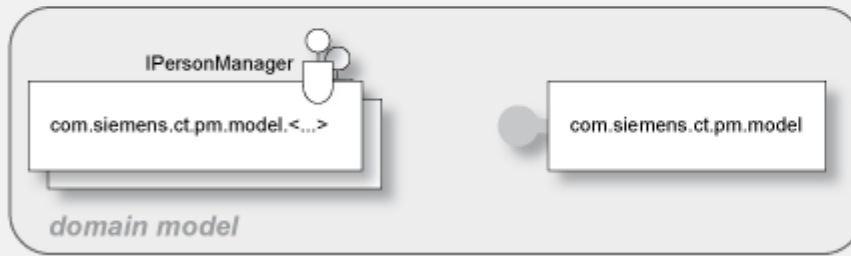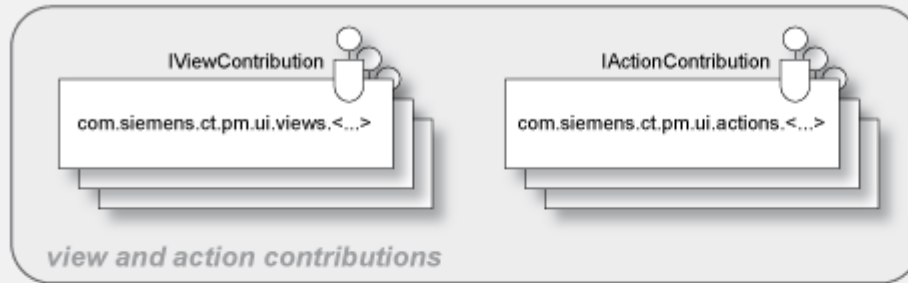
Package Dependencies

» 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
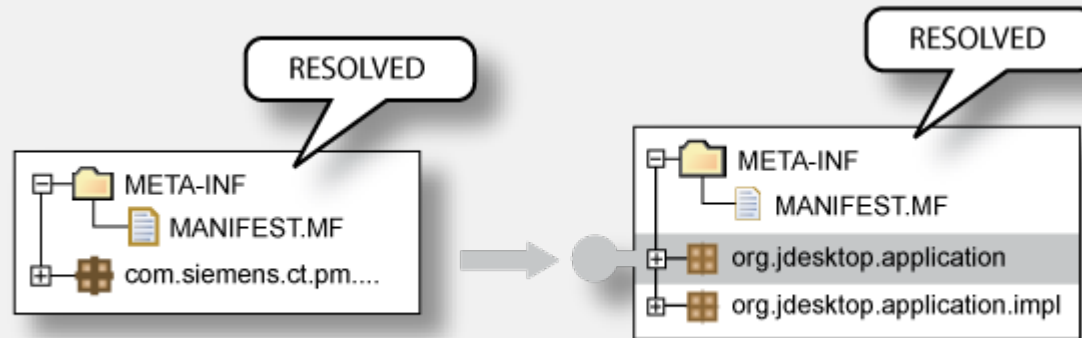
» Wires bundles by matching imports to exports

» Resolving may occur eagerly (after installation) or lazily

» There is no API for resolving

» After resolving -> Bundle is in state RESOLVED

Uninstall
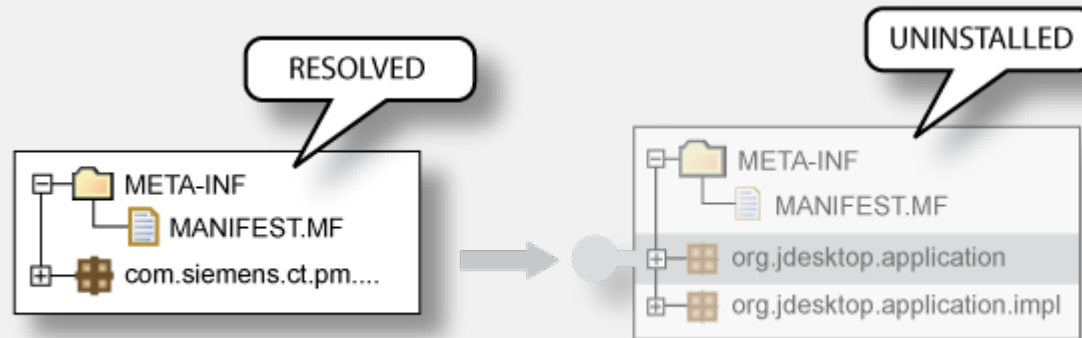
» … removes a Bundle from the OSGi Framework

» 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
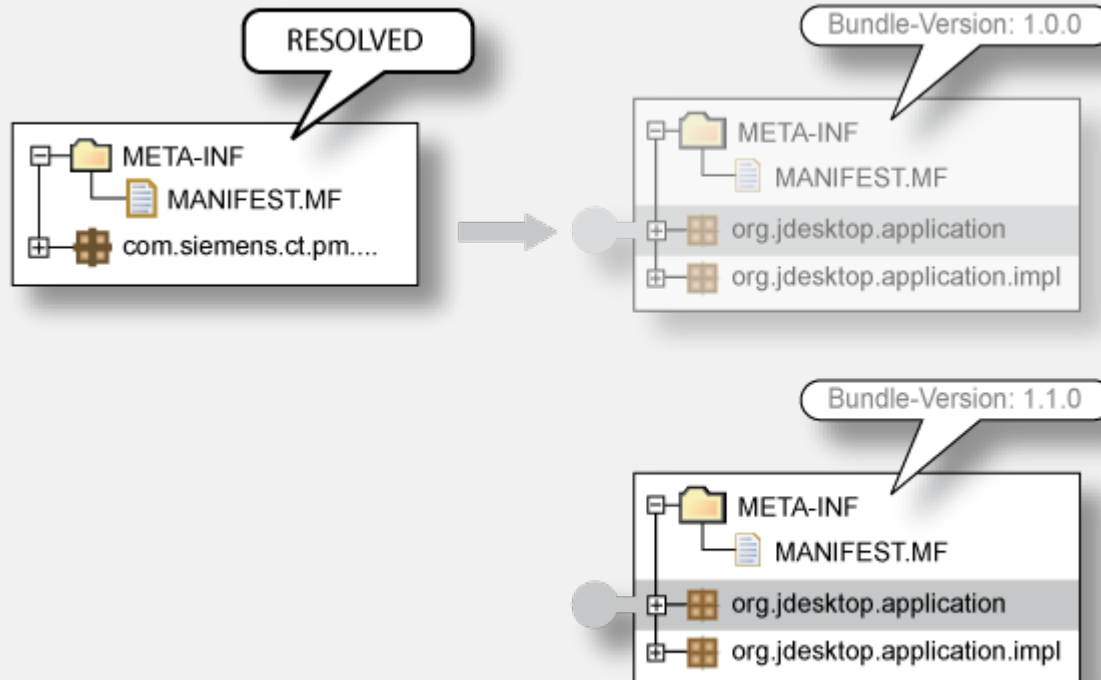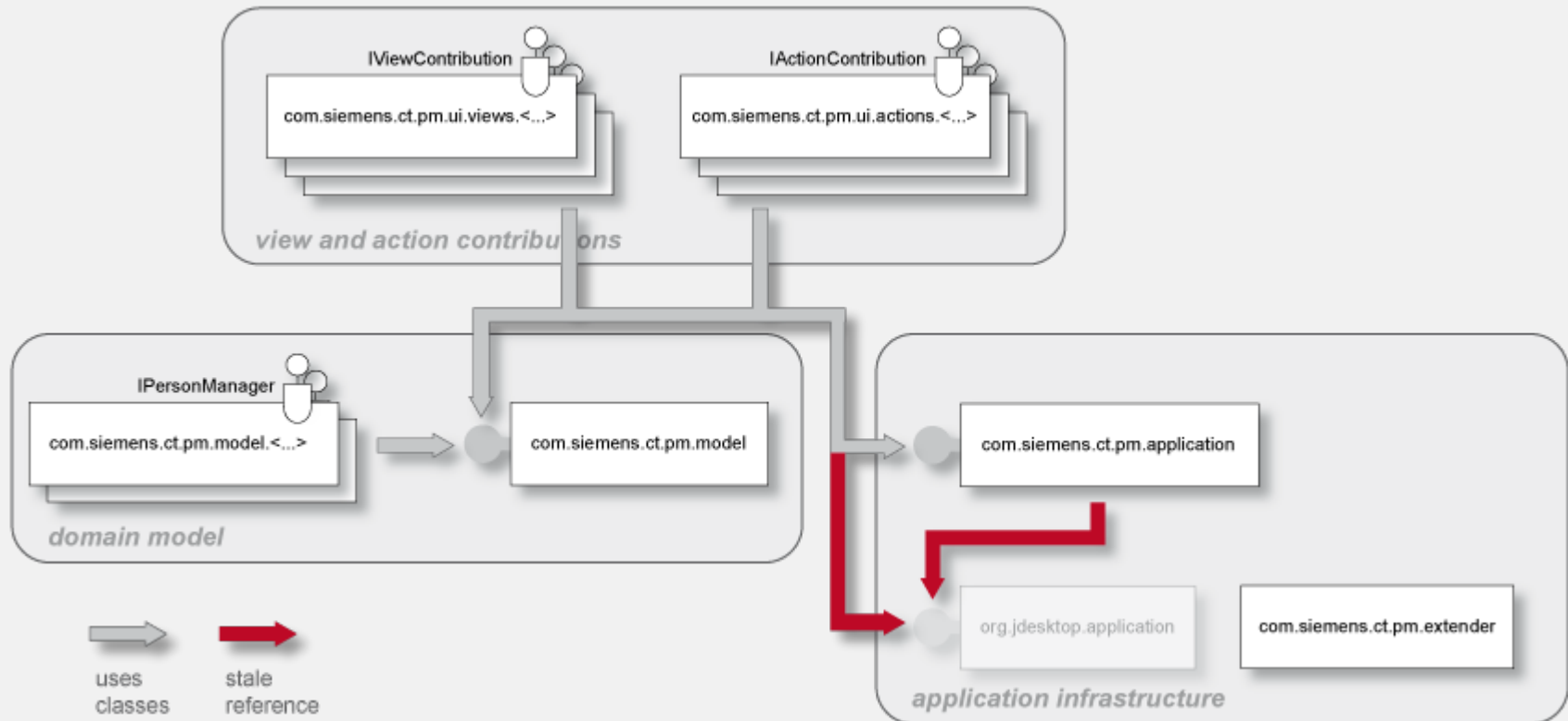

» Update:
  » Reads in the Bundle again
  » If the Bundle is an exporter: Existing wires will remain until the importers are refreshed or the OSGi Framework is restarted

» 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
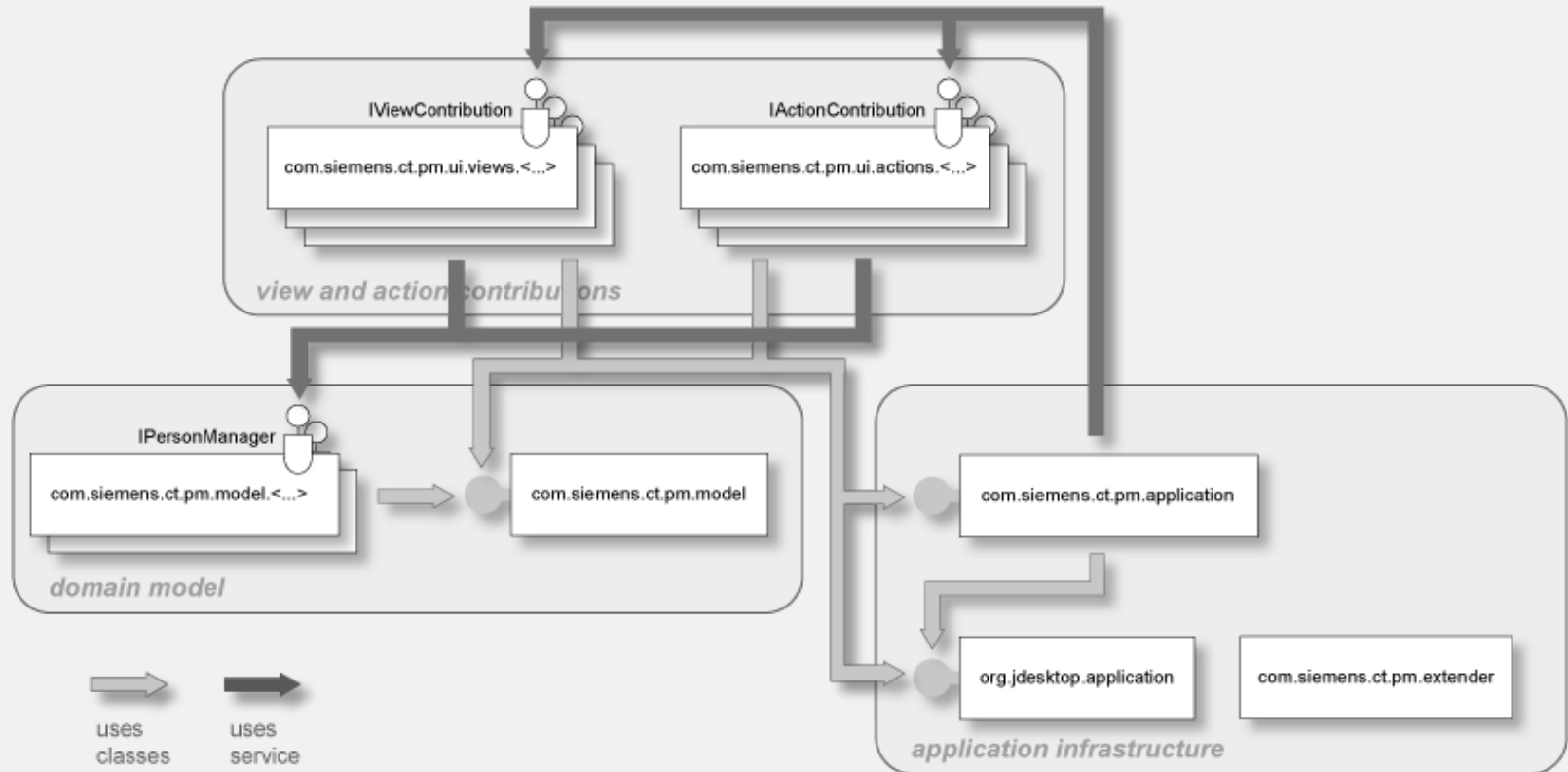
» One way to reduce coupling

» 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/

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
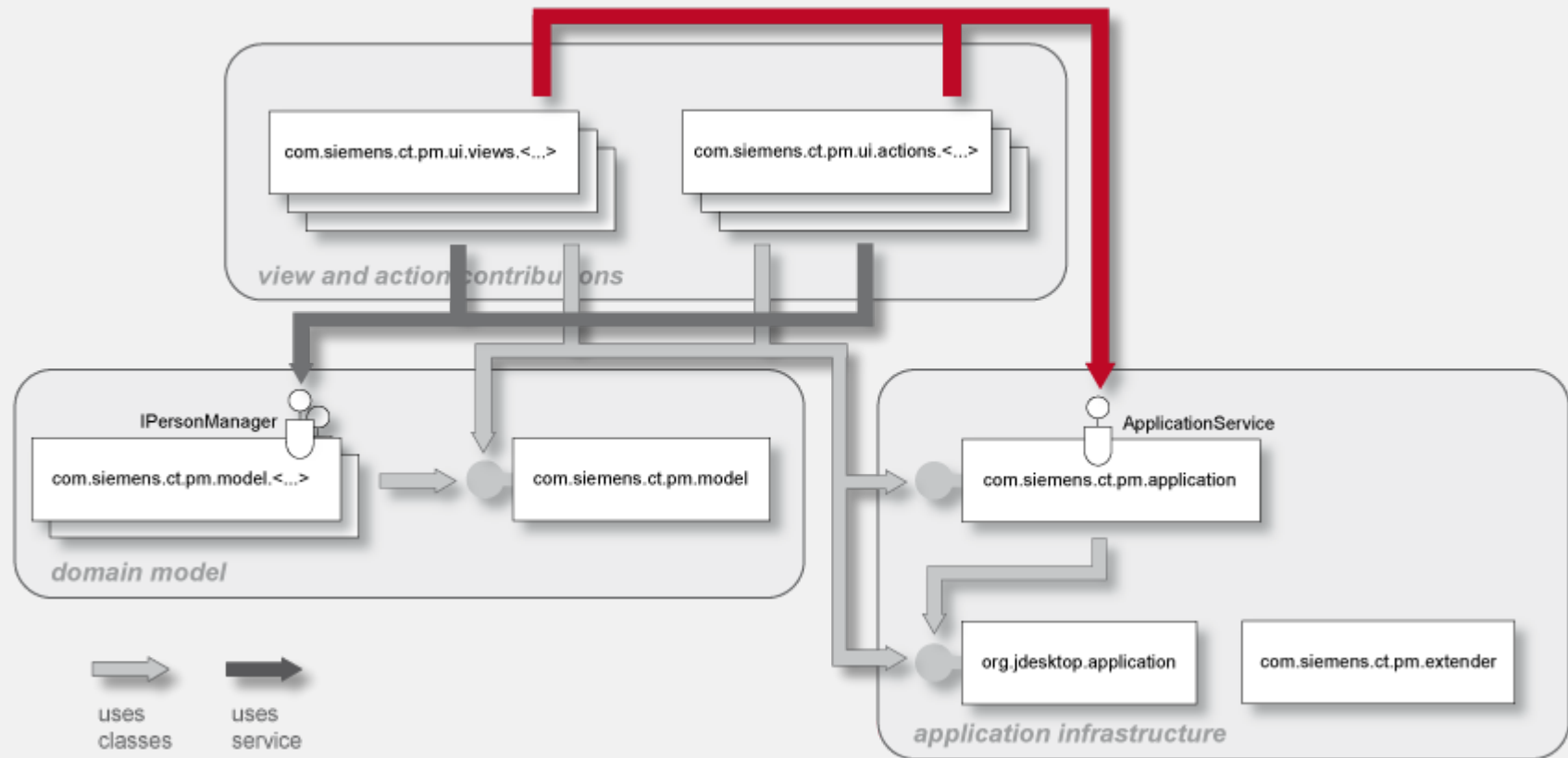
» The OSGi service model provides a service registry with these notification mechanisms already!

» So:

  » Don't get a service and register as listener
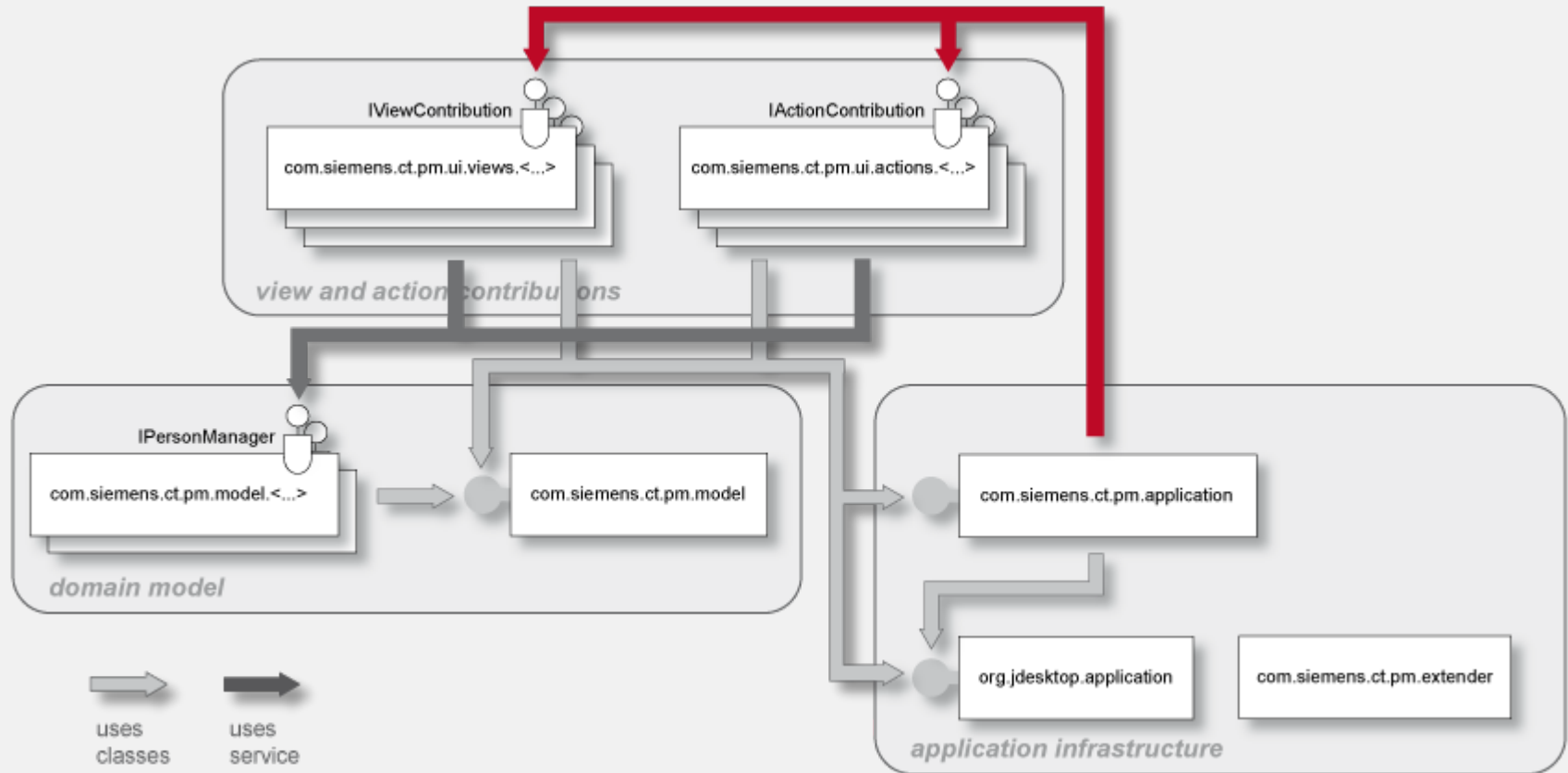
  » Be a service yourself and register with the OSGi service registry!

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

Thank you for your attention!

» Questions welcome!

» Kai Tödter

» Gerd Wütherich

» Martin Lippert