

**Apache Felix –
A Standard Plugin Model
for Apache Projects**

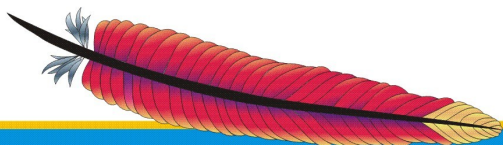
Richard S. Hall

Amsterdam, Netherlands
April 9th, 2008



Agenda

- ① Why OSGi technology?
- ② OSGi technology overview
- ③ Apache Felix status
- ④ Example application
- ⑤ OSGi application approaches
- ⑥ Example application demo
- ⑦ Advanced approaches
- ⑧ Conclusion



①

Why OSGi Technology?

② ③ ④ ⑤ ⑥ ⑦ ⑧



Motivation (1/2)

- Growing complexity requires not only highly modular code, but also systems that are dynamically extensible
- This is true no matter which problem domain is your area of concern
 - Embedded systems need to adapt to changing requirements even though they are deployed out in the field
 - Client applications must respond to user desires for new functionality instantaneously
 - Server applications must be configurable and manageable without down time

Motivation (2/2)

- Java provides the mechanisms to do these things, but they are
 - Low level
 - Error prone
 - Ad hoc
- Java's shortcomings are particularly evident in its support for both modularity and dynamism



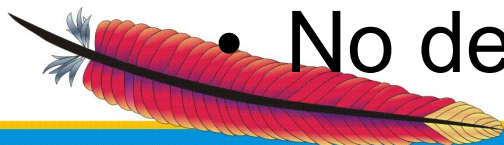
Java Modularity Limitations (1/2)

- Limited scoping mechanisms
 - No module access modifier
- Simplistic version handling
 - Class path is first version found
 - JAR files assume backwards compatibility at best
- Implicit dependencies
 - Dependencies are implicit in class path ordering
 - JAR files add improvements for extensions, but cannot control visibility



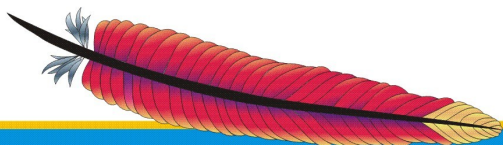
Java Modularity Limitations (2/2)

- Split packages by default
 - Class path approach searches until it finds, which leads to shadowing or version mixing
 - JAR files can provide sealing
- Unsophisticated consistency model
 - Cuts across previous issues, it is difficult to ensure class space consistency
- Missing module concept
 - Classes are too fine grained, packages are too simplistic, class loaders are too low level
- No deployment support



Java Dynamism Limitations

- Low-level support for dynamics
 - Class loaders are complicated to use and error prone
- Support for dynamics is still purely manual
 - Must be completely managed by the programmer
 - Leads to many ad hoc, incompatible solutions
- Limited deployment support



OSGi Technology

- Resolves many deficiencies associated with standard Java support for modularity and dynamism
 - Defines a module concept
 - Explicit sharing of code (i.e., importing and exporting)
 - Automatic management of code dependencies
 - Enforces sophisticated consistency rules for class loading
 - Life-cycle management
 - Manages dynamic deployment and configuration



②

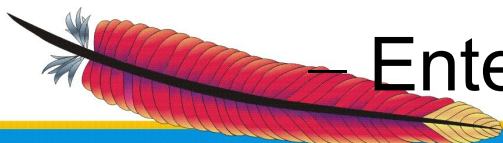
OSGi Technology Overview

③ ④ ⑤ ⑥ ⑦ ⑧

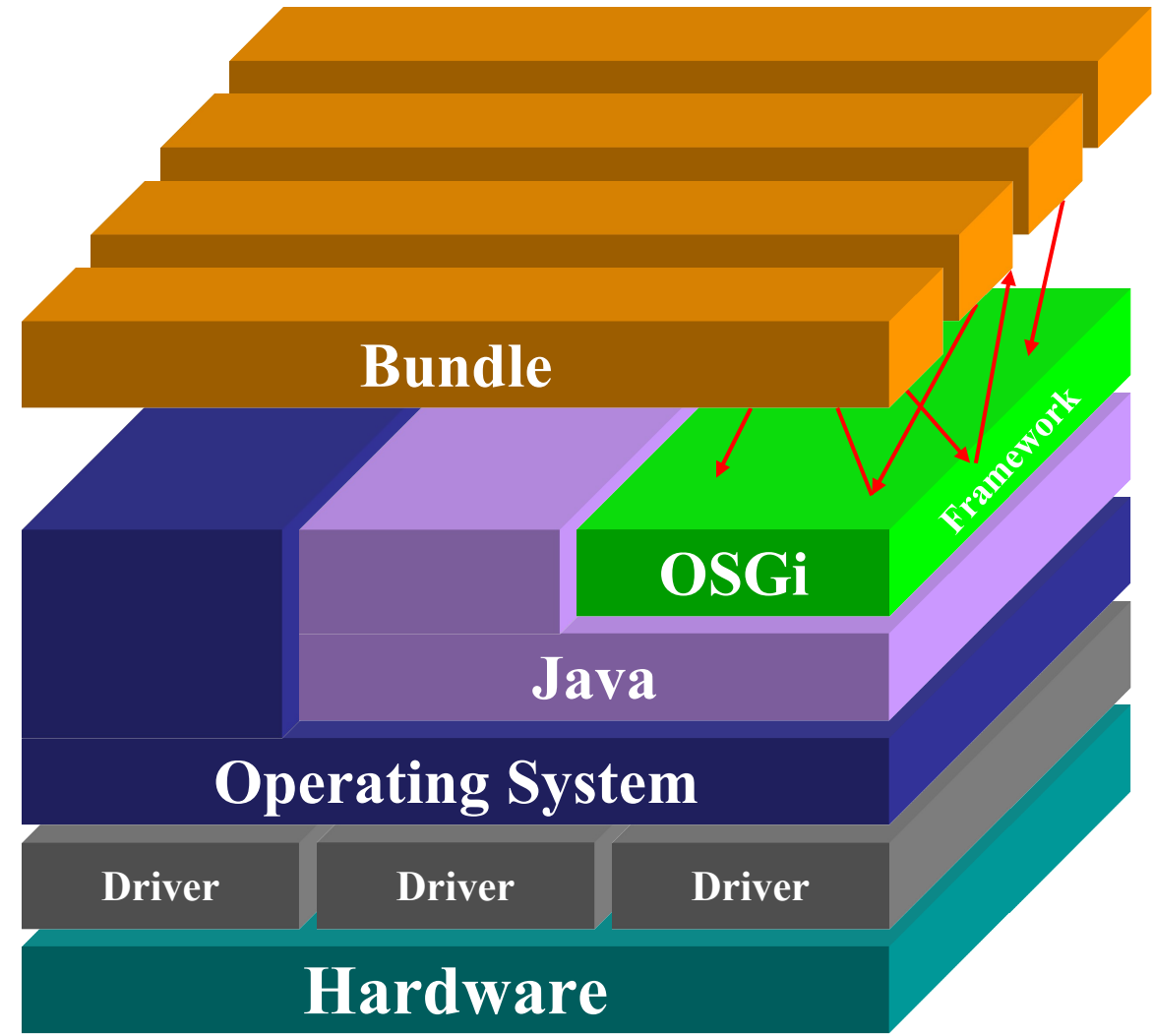


OSGi Alliance

- Industry consortium
- ***OSGi Service Platform*** specification
 - Framework specification for hosting dynamically downloadable services
 - Standard service specifications
- Several expert groups define the specifications
 - Core Platform Expert Group (CPEG)
 - Mobile Expert Group (MEG)
 - Vehicle Expert Group (VEG)
 - Enterprise Expert Group (EEG)



OSGi Architectural Overview



OSGi Framework Layering

SERVICE MODEL

L3 - Provides a publish/find/bind service model to decouple bundles

LIFECYCLE

L2 - Manages the life cycle of bundle in a bundle repository without requiring the VM be restarted

MODULE

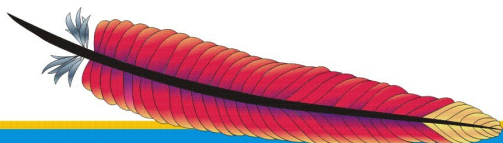
L1 - Creates the concept of modules (aka. bundles) that use classes from each other in a controlled way according to system and bundle constraints

Execution Environment

L0 -
•OSGi Minimum Execution Environment
•CDC/Foundation
•JavaSE

OSGi Framework (1/2)

- Component-oriented framework
 - **Bundles** (i.e., modules/components)
 - Package sharing and version management
 - Life-cycle management and notification
- Service-oriented interaction pattern
 - Publish/find/bind intra-VM service model



OSGi Framework (2/2)

- Runs multiple applications and services
- Single VM instance
- Separate class loader per bundle
 - Class loader graph
 - Independent namespaces
 - Class sharing at the Java package level
- Uses Java permissions for security
- Explicitly considers dynamic scenarios
 - Run-time install, update, and uninstall of bundles



OSGi Modularity (1/4)

- Multi-version support
 - i.e., side-by-side versions
- Explicit code boundaries and dependencies
 - i.e., package imports and exports
- Support for various sharing policies
 - i.e., arbitrary version range support



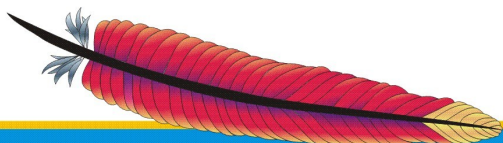
OSGi Modularity (2/4)

- Arbitrary export/import attributes for more control
 - Influence package selection
- Sophisticated class space consistency model
 - Ensures code constraints are not violated
- Package filtering for fine-grained class visibility
 - Exporters may include/exclude specific classes from exported package



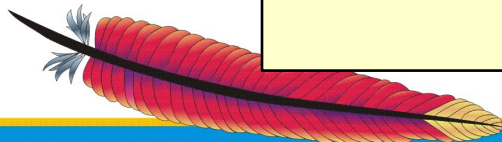
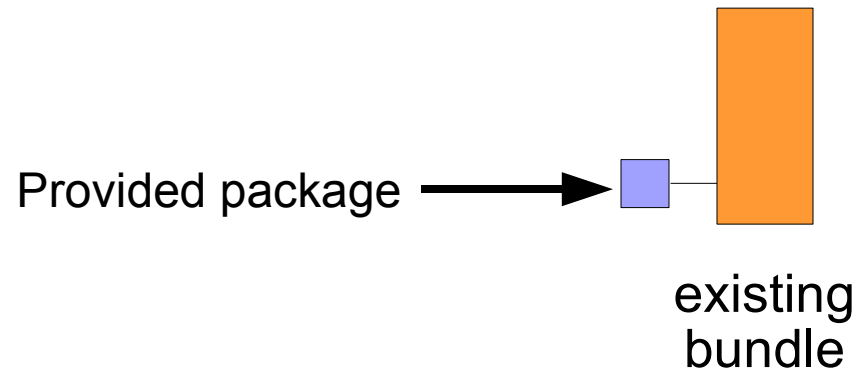
OSGi Modularity (3/4)

- Bundle fragments
 - A single logical module in multiple physical bundles
- Bundle dependencies
 - Allows for tight coupling when required
- Managed life cycle
 - Modules can be dynamically installed, updated, uninstalled



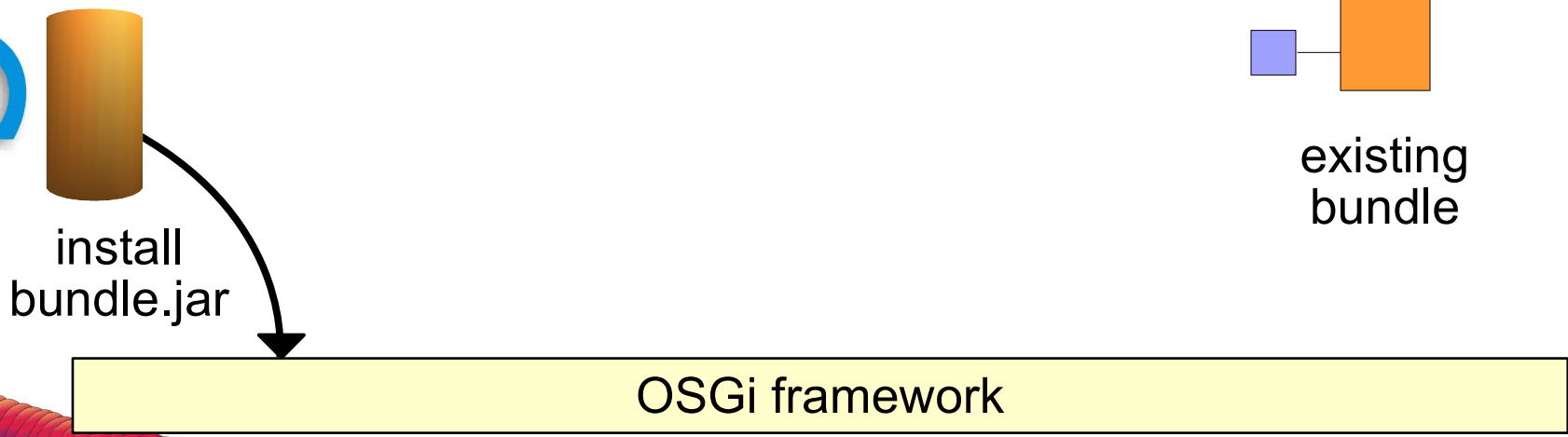
OSGi Modularity (4/4)

- Dynamic module deployment and dependency resolution



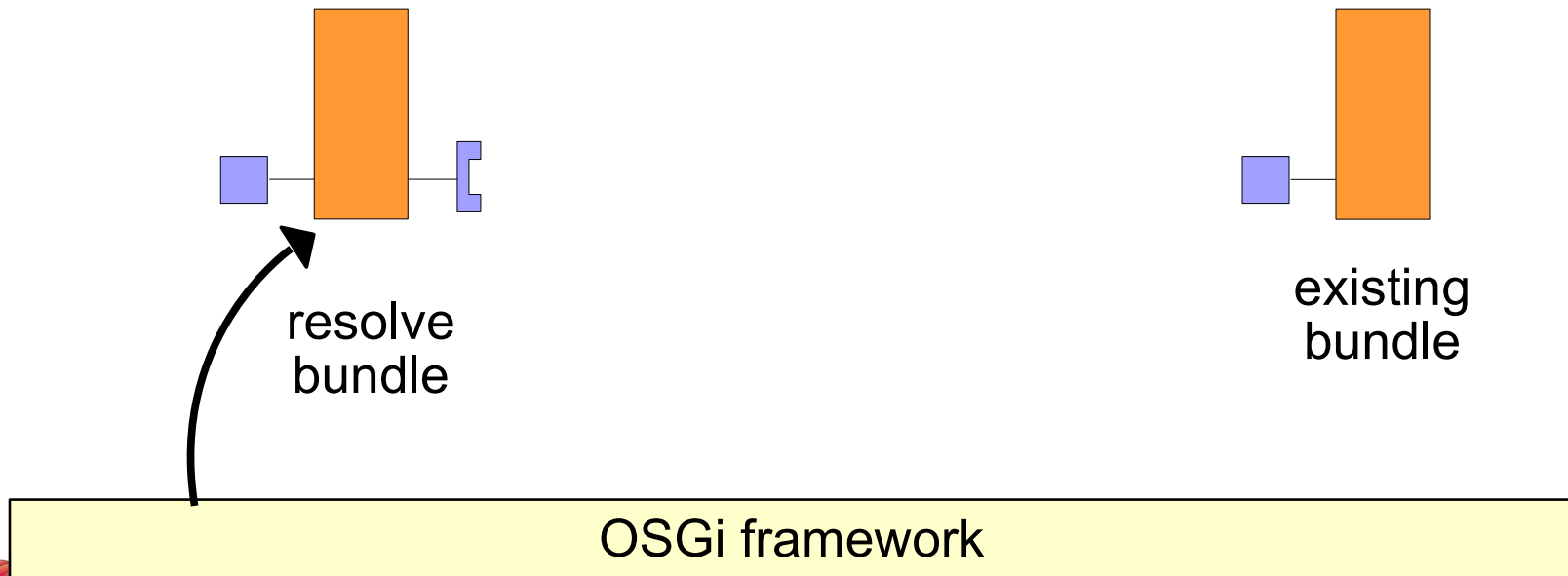
OSGi Modularity (4/4)

- Dynamic module deployment and dependency resolution



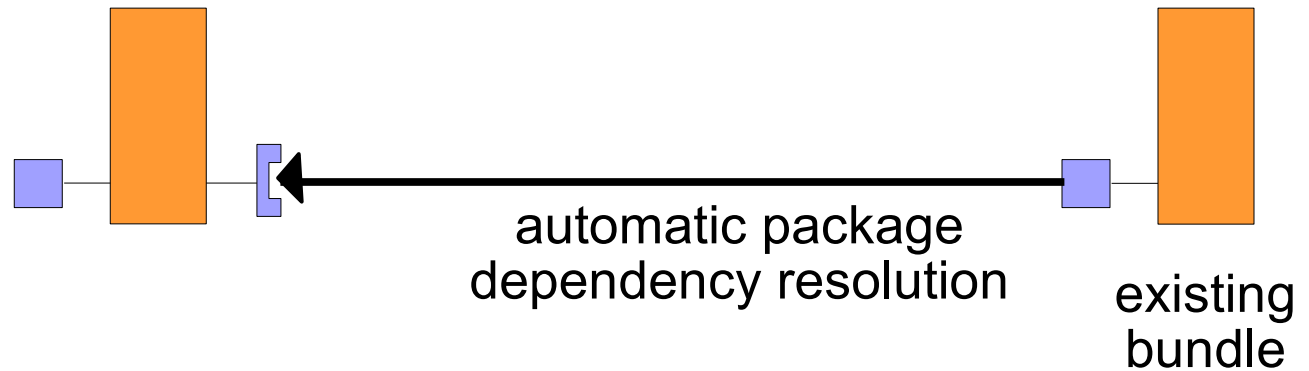
OSGi Modularity (4/4)

- Dynamic module deployment and dependency resolution



OSGi Modularity (4/4)

- Dynamic module deployment and dependency resolution



OSGi framework

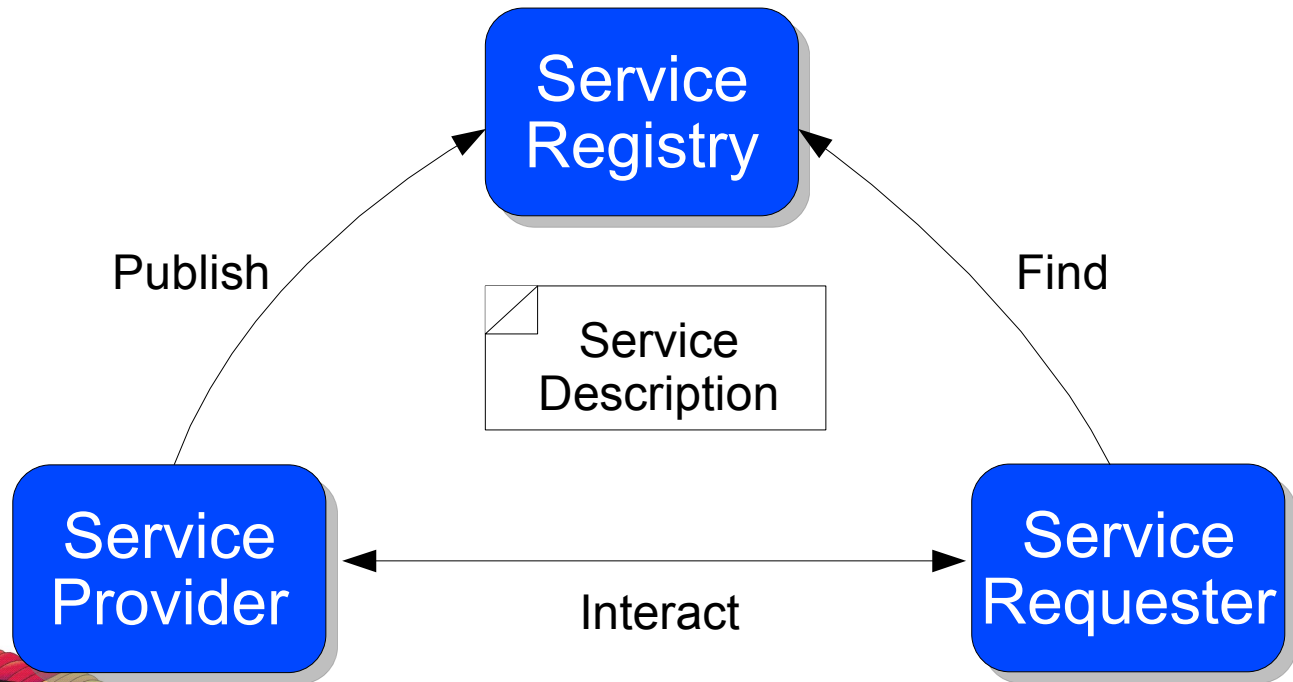
Leveraging OSGi Modularity

- Text editor + `jar`
 - Just add metadata to your JAR file's manifest
- Eclipse
 - Plug-in Development Environment (**PDE**) directly supports bundles
- Bundle packaging tools
 - **BND** from Peter Kriens
 - Apache Felix *maven-bundle-plugin* based on BND



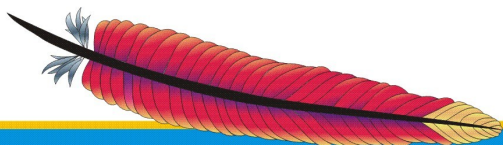
OSGi Services (1/3)

- OSGi framework promotes service-oriented interaction pattern among bundles
 - Possible to use modules without services



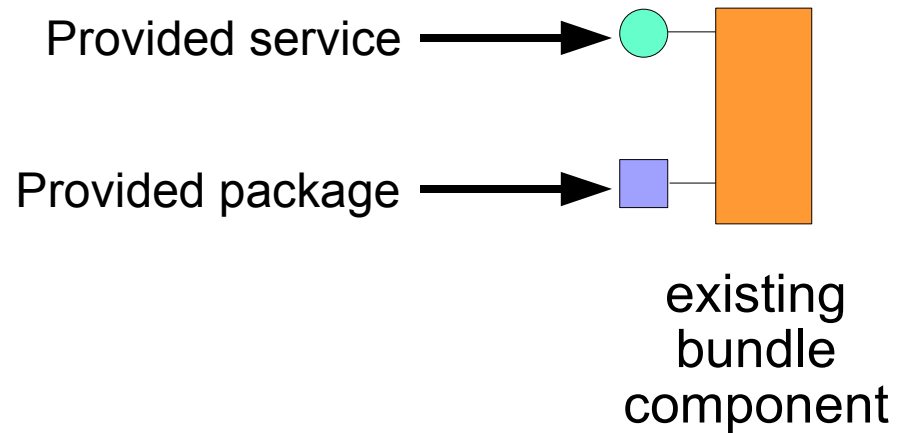
OSGi Services (2/3)

- An OSGi application is...
 - A collection of bundles that interact via service interfaces
 - Bundles may be independently developed and deployed
 - Bundles and their associated services may appear or disappear at any time
- Resulting application follows a **Service-Oriented Component Model** approach



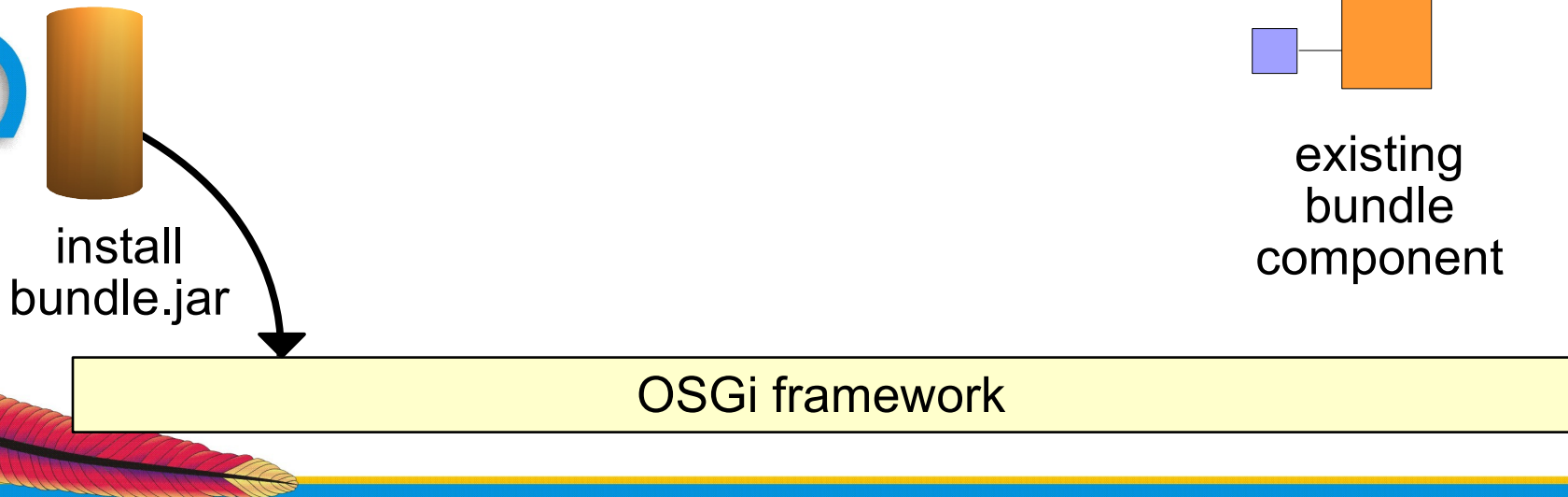
OSGi Services (3/3)

- Dynamic service lookup



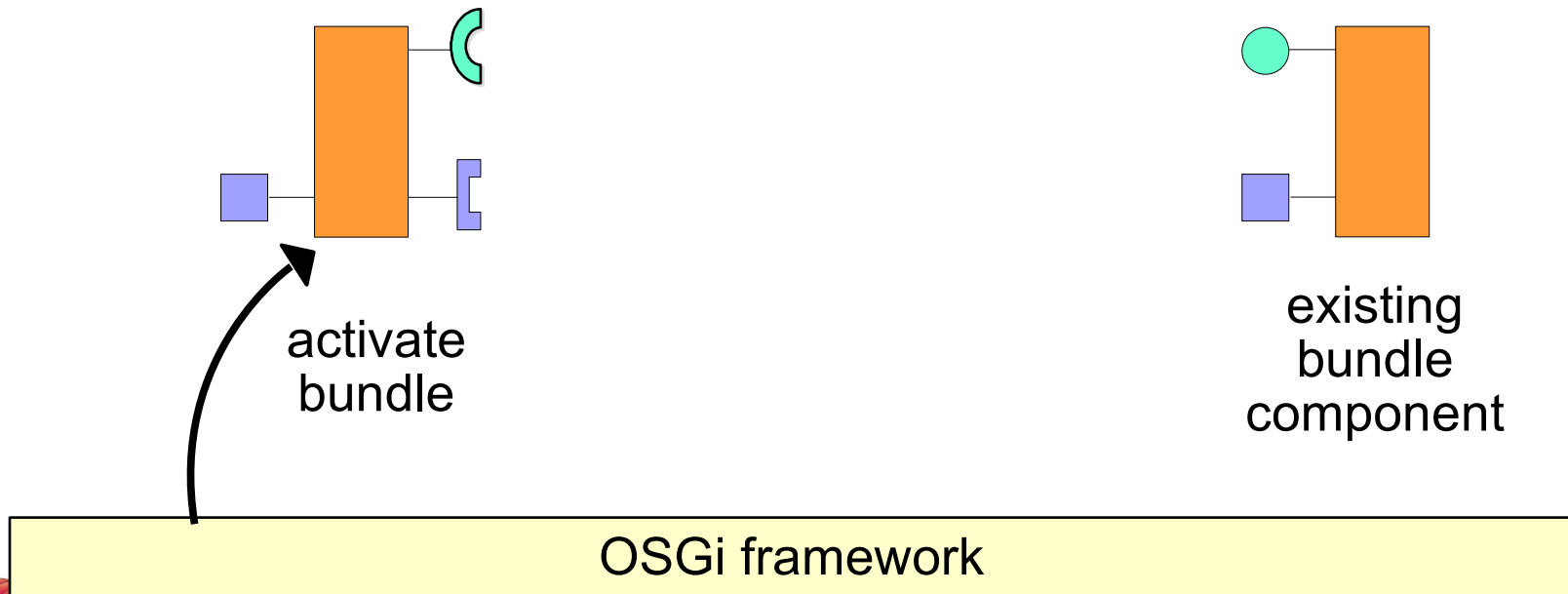
OSGi Services (3/3)

- Dynamic service lookup



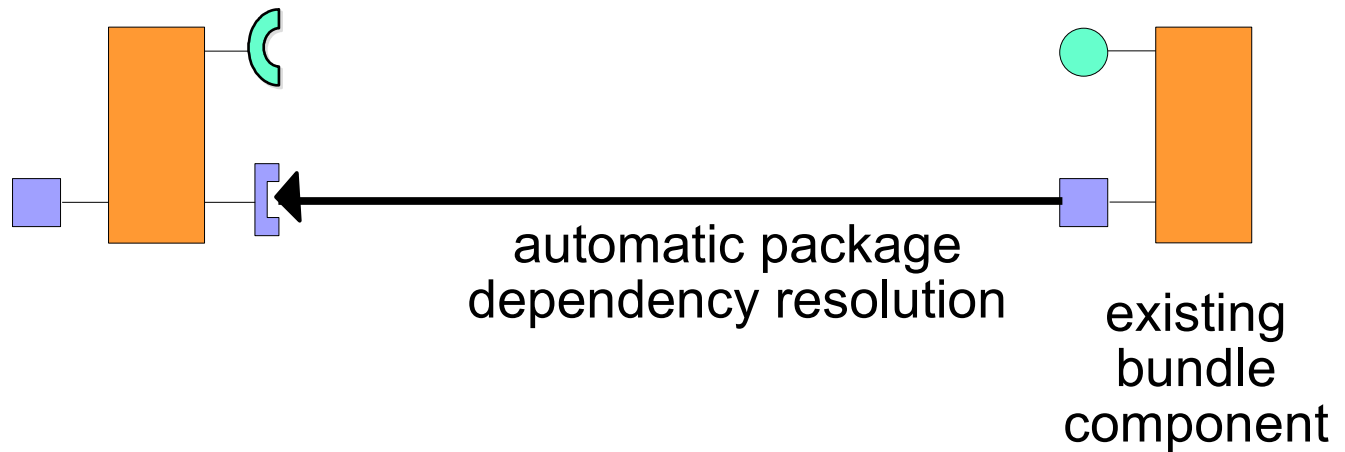
OSGi Services (3/3)

- Dynamic service lookup



OSGi Services (3/3)

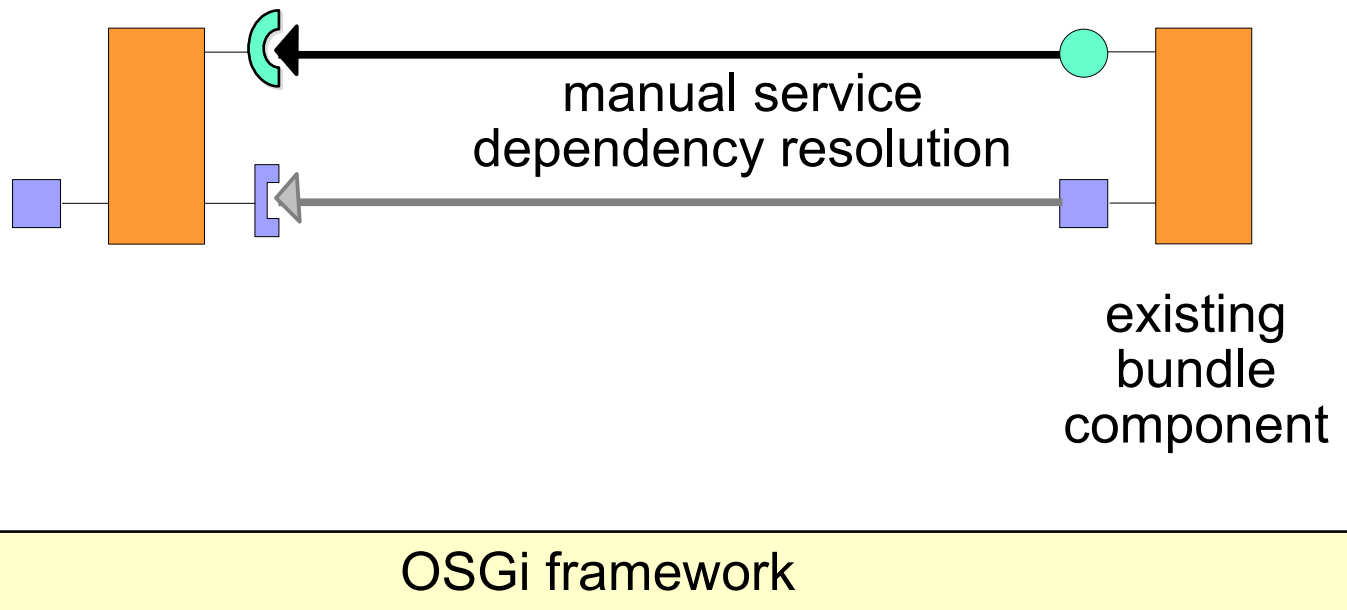
- Dynamic service lookup



OSGi framework

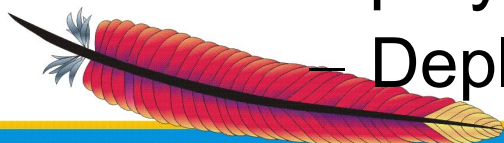
OSGi Services (3/3)

- Dynamic service lookup



OSGi Services Advantages

- Lightweight services
 - Direct method invocation
- Good design practice
 - Separates interface from implementation
 - Enables reuse, substitutability, loose coupling, and late binding
- Dynamic
 - Loose coupling and late binding make it possible to support run-time dynamism
- Application's configuration is simply the set of deployed bundles
 - Deploy only the bundles that you need



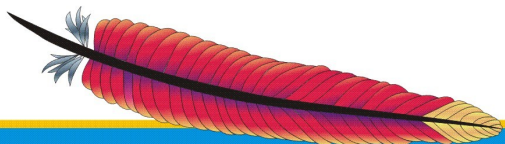
OSGi Services Issues

- More sophisticated, but more complicated
 - Requires a different way of thinking
 - Things might appear/disappear at any moment
 - Must manually resolve and track services
- There is help
 - Service Tracker
 - Still somewhat of a manual approach
 - Declarative Services, Spring DM, iPOJO
 - Sophisticated service-oriented component frameworks
 - Automated dependency injection and more
 - More modern, POJO-oriented approaches

③

Apache Felix Status

④ ⑤ ⑥ ⑦ ⑧



Apache Felix (1/4)

- Top-level project (March 2007)
- Apache licensed open source implementation of OSGi R4
 - Framework (in progress, stable and functional)
 - Version 1.0.3 currently available
 - Services (in progress, stable and functional)
 - Package Admin, Start Level, URL Handlers, Declarative Services, UPnP Device, HTTP Service, Configuration Admin, Preferences, Wire Admin, Event Admin, Meta Type, Deployment Admin and Log
 - OSGi Bundle Repository (OBR), Dependency Manager, Service Binder, Shell, **iPOJO**, Mangan



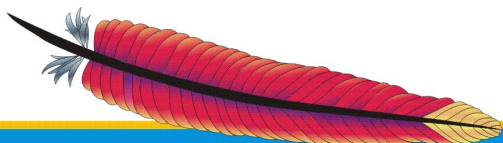
Apache Felix (2/4)

- Felix community is growing
 - Code granted and contributed from several organizations and communities
 - Grenoble University, ObjectWeb, CNR-ISTI, Ascert, Luminis, INSA, DIT UPM, Day Management AG
 - Several community member contributions
 - Various Apache Java-based projects have expressed interested in Felix and/or OSGi
 - e.g., ServiceMix, Directory, Sling, Tuscan, etc.



Apache Felix (3/4)

- Felix bundle developer support
 - Apache Maven2 bundle plugin
 - Merges OSGi bundle manifest with Maven2 POM file
 - Automatically generates metadata, such as `Bundle-ClassPath`, `Import-Package`, and `Export-Package`
 - Greatly simplifies bundle development by eliminating error-prone manual header creation process
 - Automatically creates final bundle JAR file
 - Also supports embedding required packages, instead of importing them



Apache Felix (4/4)

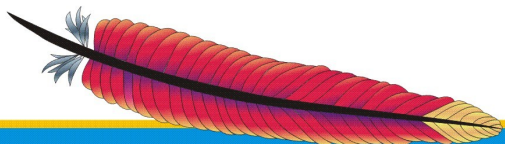
- Felix Commons
 - Effort to bundle-ize common open source libraries
 - Recently started
 - Includes bundles such as antlr, cglib, commons-collections, etc.
 - All community donated wrappers
- Roadmap
 - Continue toward R4 and R4.1 compliance
 - Largely only missing support for fragments, but work has started



④

Example Application

⑤ ⑥ ⑦ ⑧



Paint Program Plan

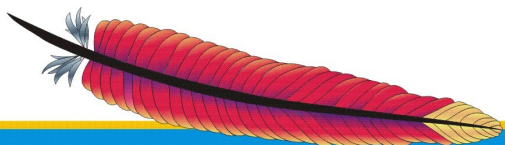
- Create a simple Swing-based paint program
- Define a `SimpleShape` interface to draw shapes
 - Different implementations of `SimpleShape` can be created to draw different shapes
 - Each shape has name and icon properties
 - Available shapes are displayed in tool bar
- To draw a shape, click on its button and then click in the drawing canvas
 - Shapes can be dragged, but not resized
- Support dynamic deployment of shapes



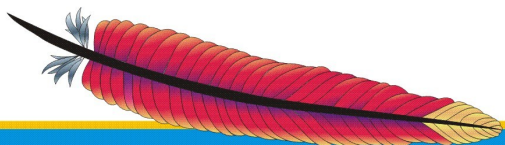
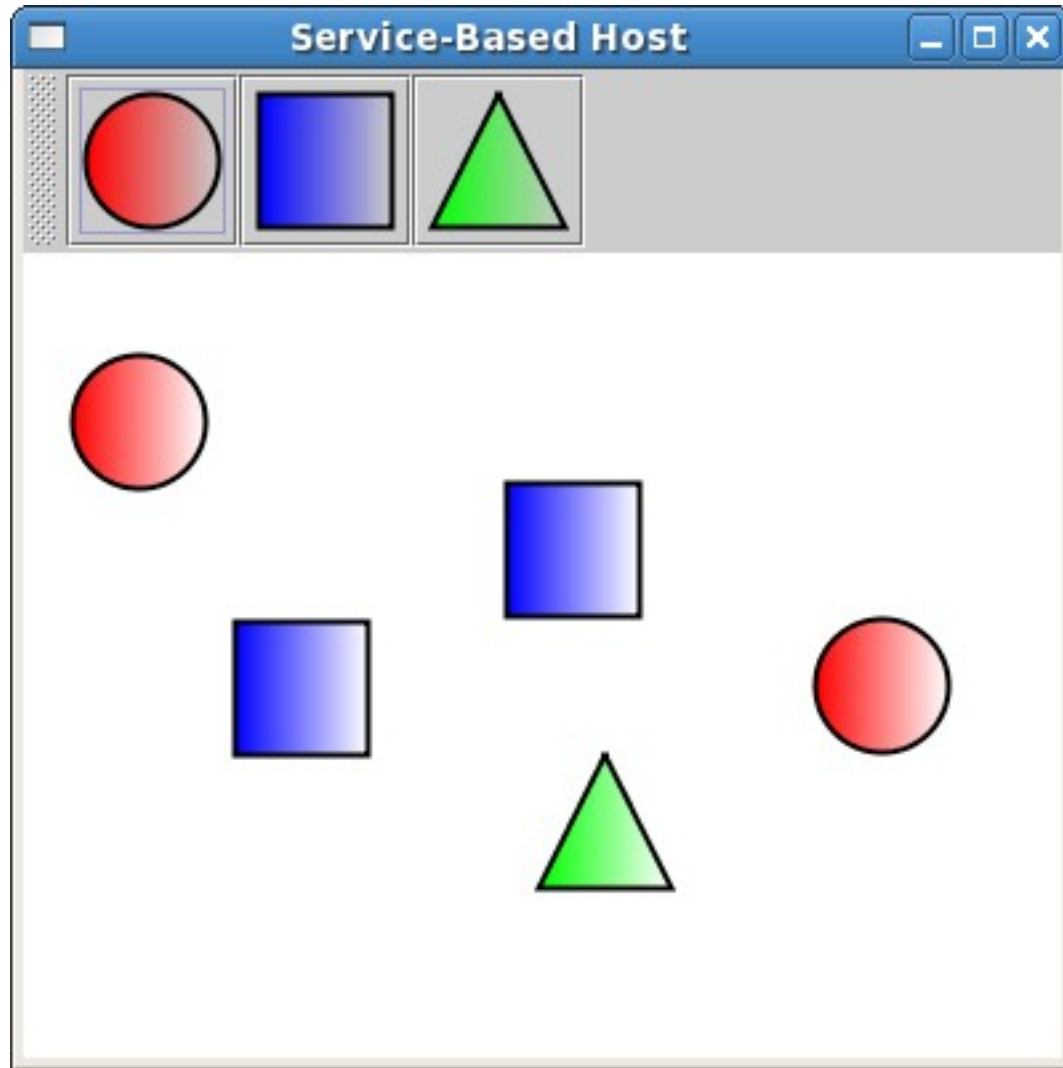
Shape Abstraction

- Conceptual SimpleShape interface

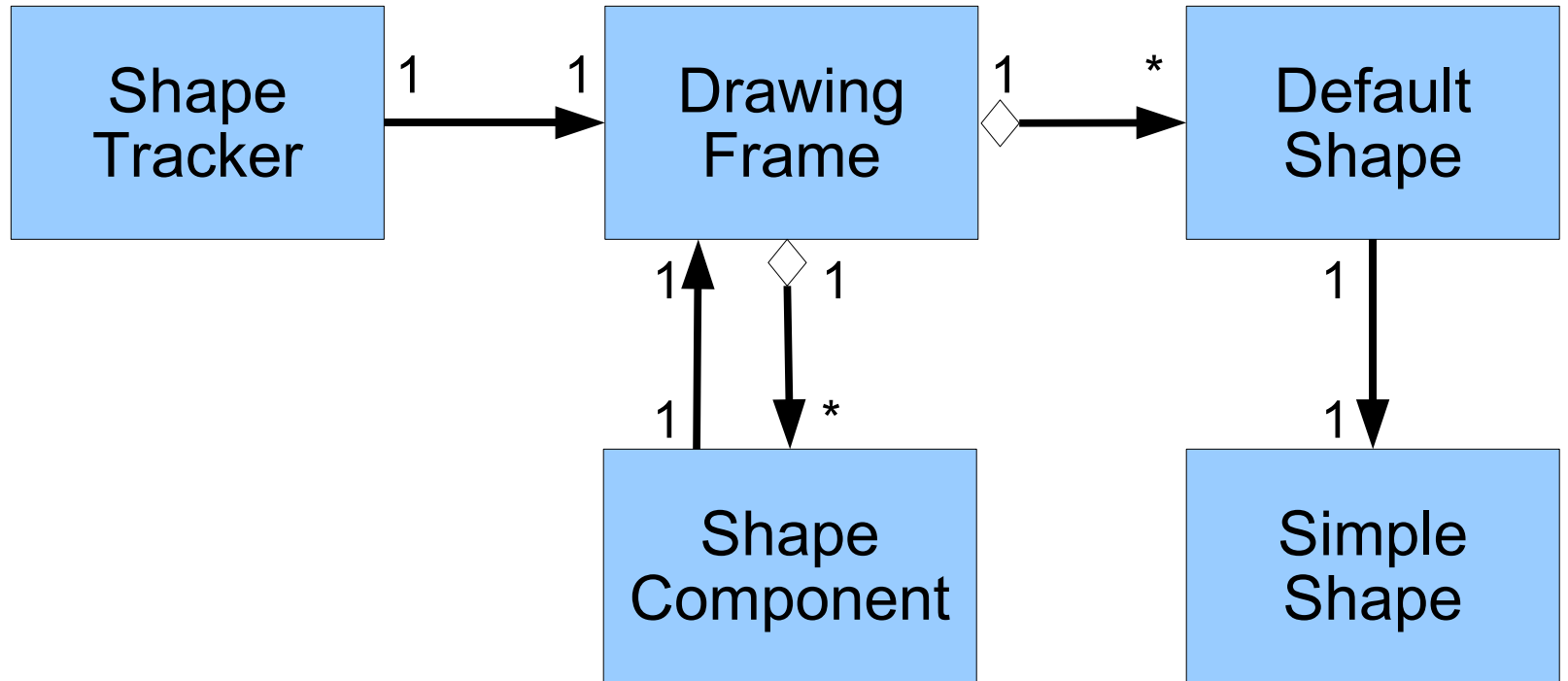
```
public interface SimpleShape
{
    /**
     * Method to draw the shape of the service.
     * @param g2 The graphics object used for
     *           painting.
     * @param p The position to paint the shape.
     */
    public void draw(Graphics2D g2, Point p);
}
```



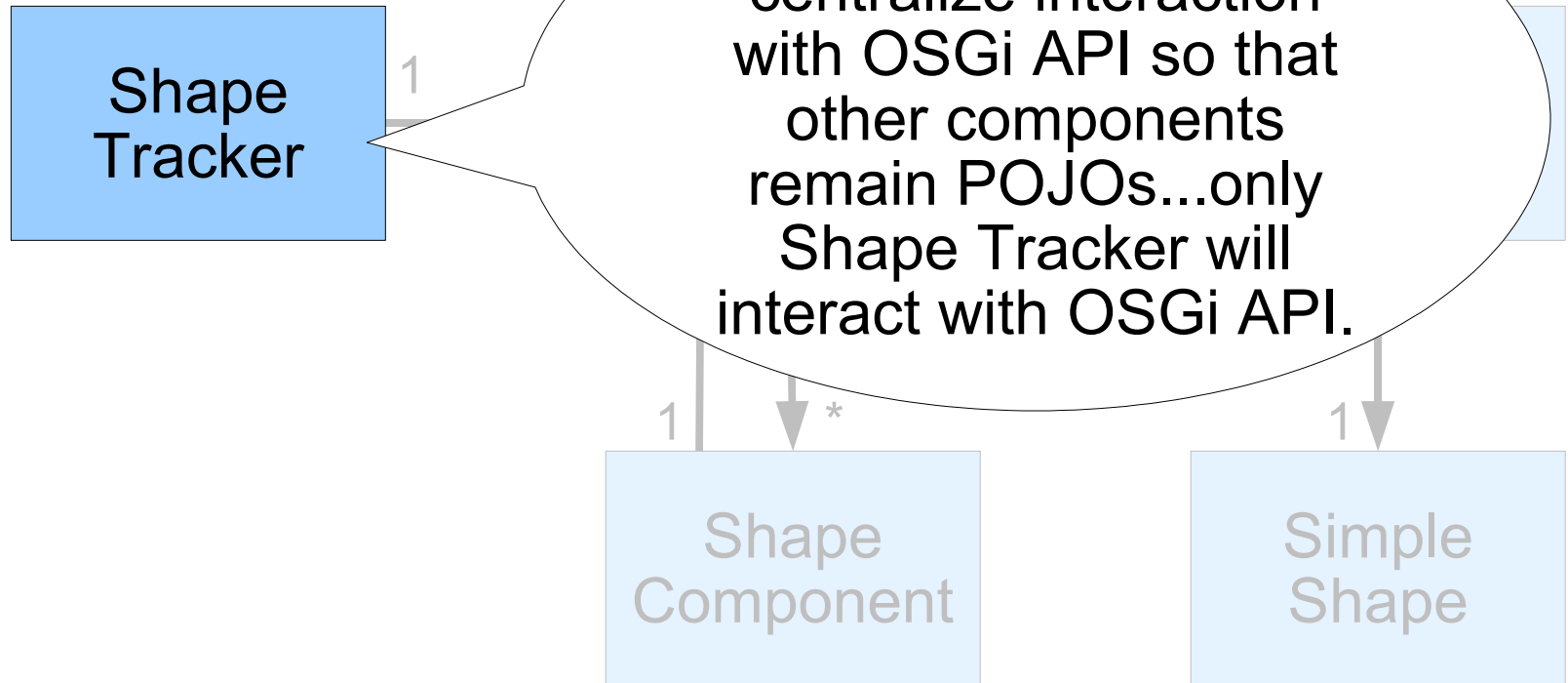
Paint Program Mock Up



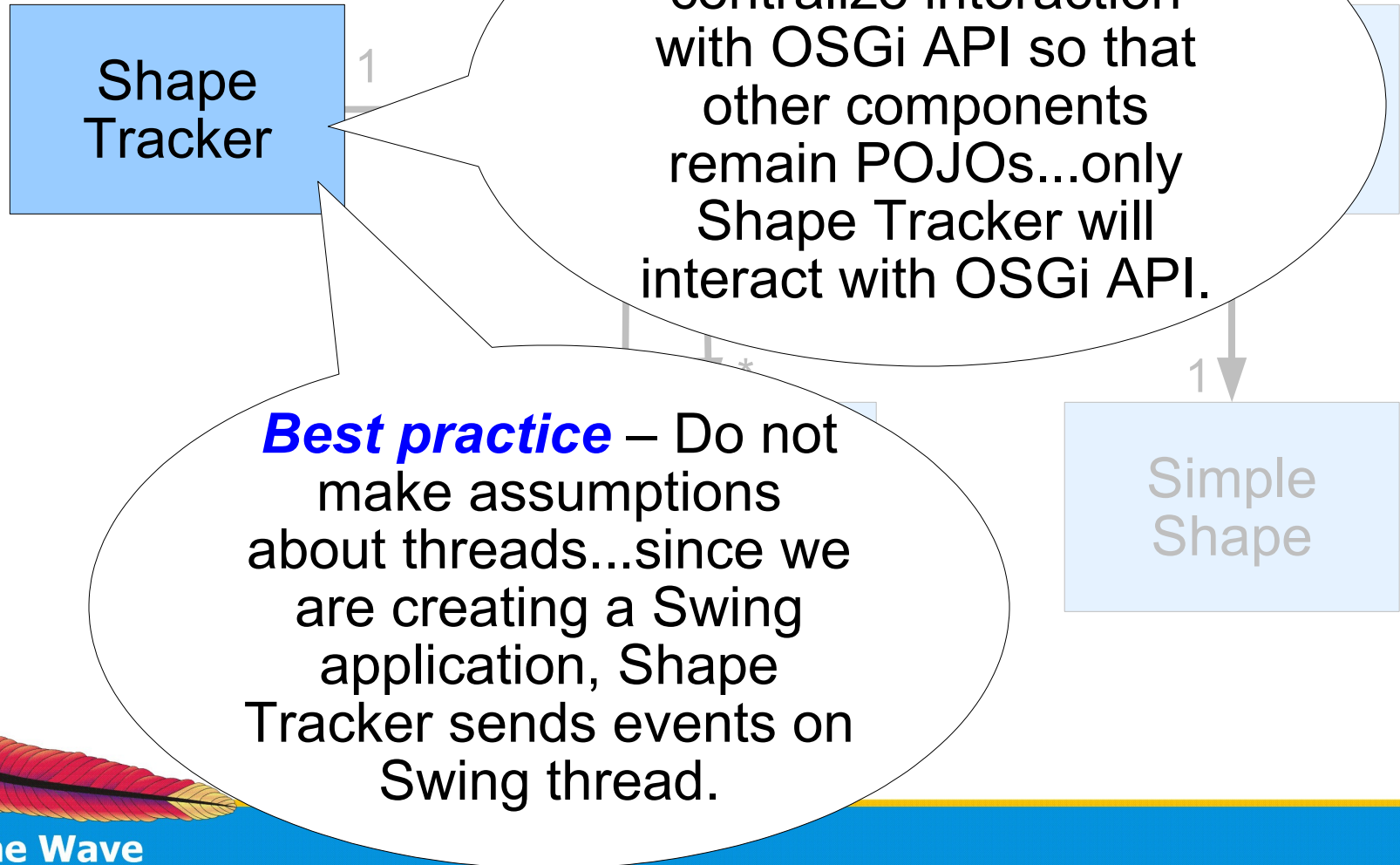
High-Level Architecture



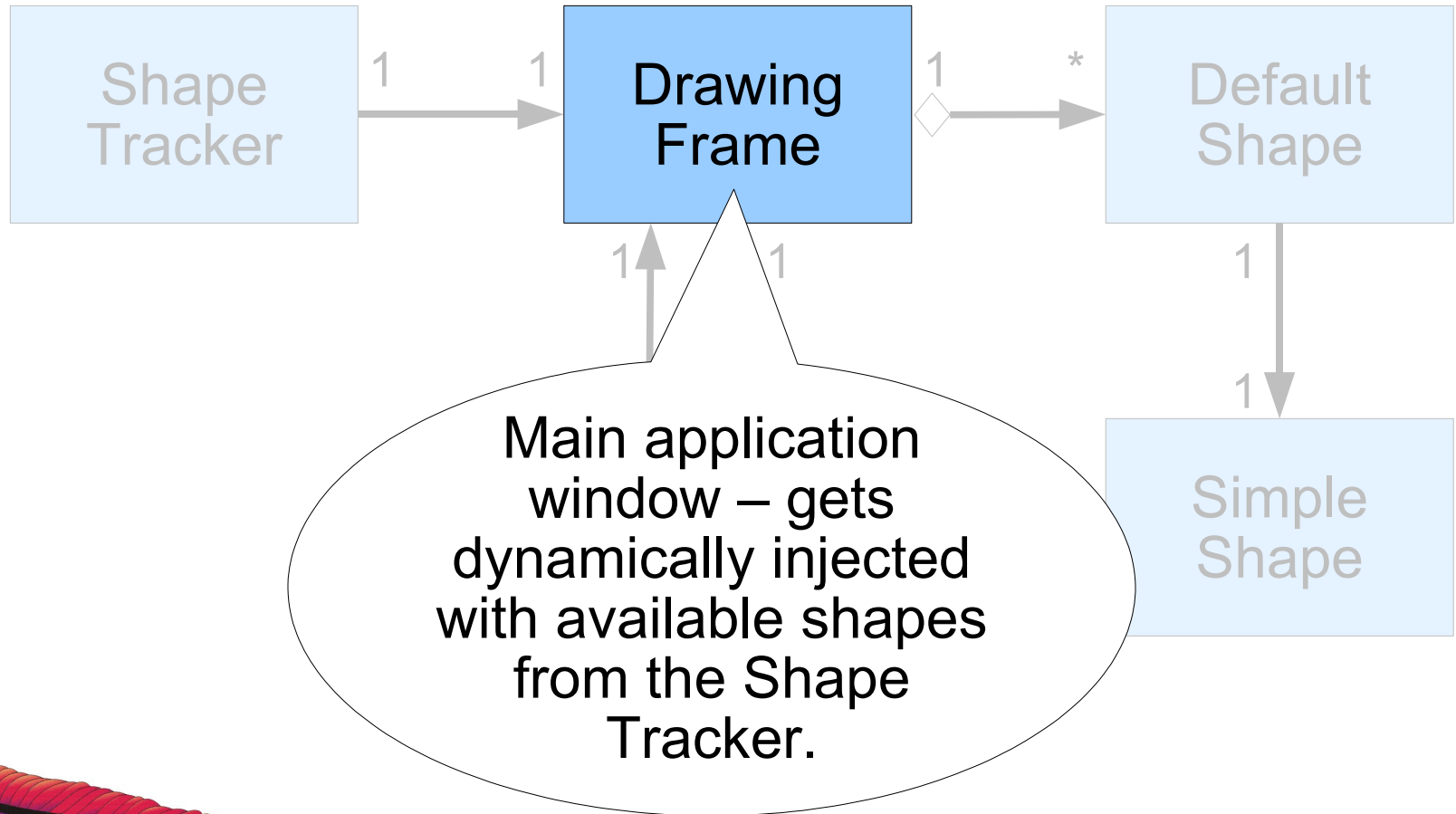
High-Level Architecture



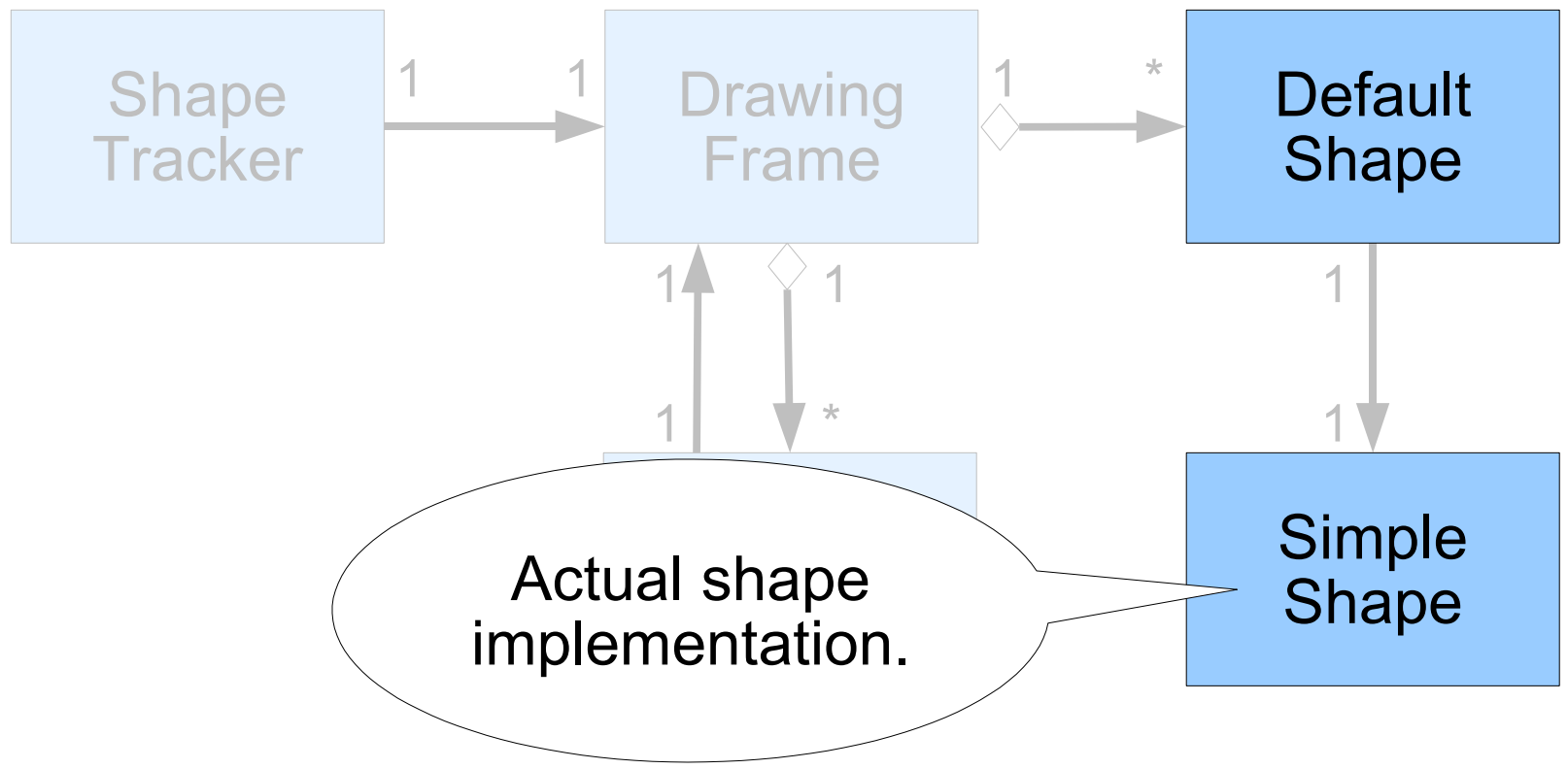
High-Level Architecture



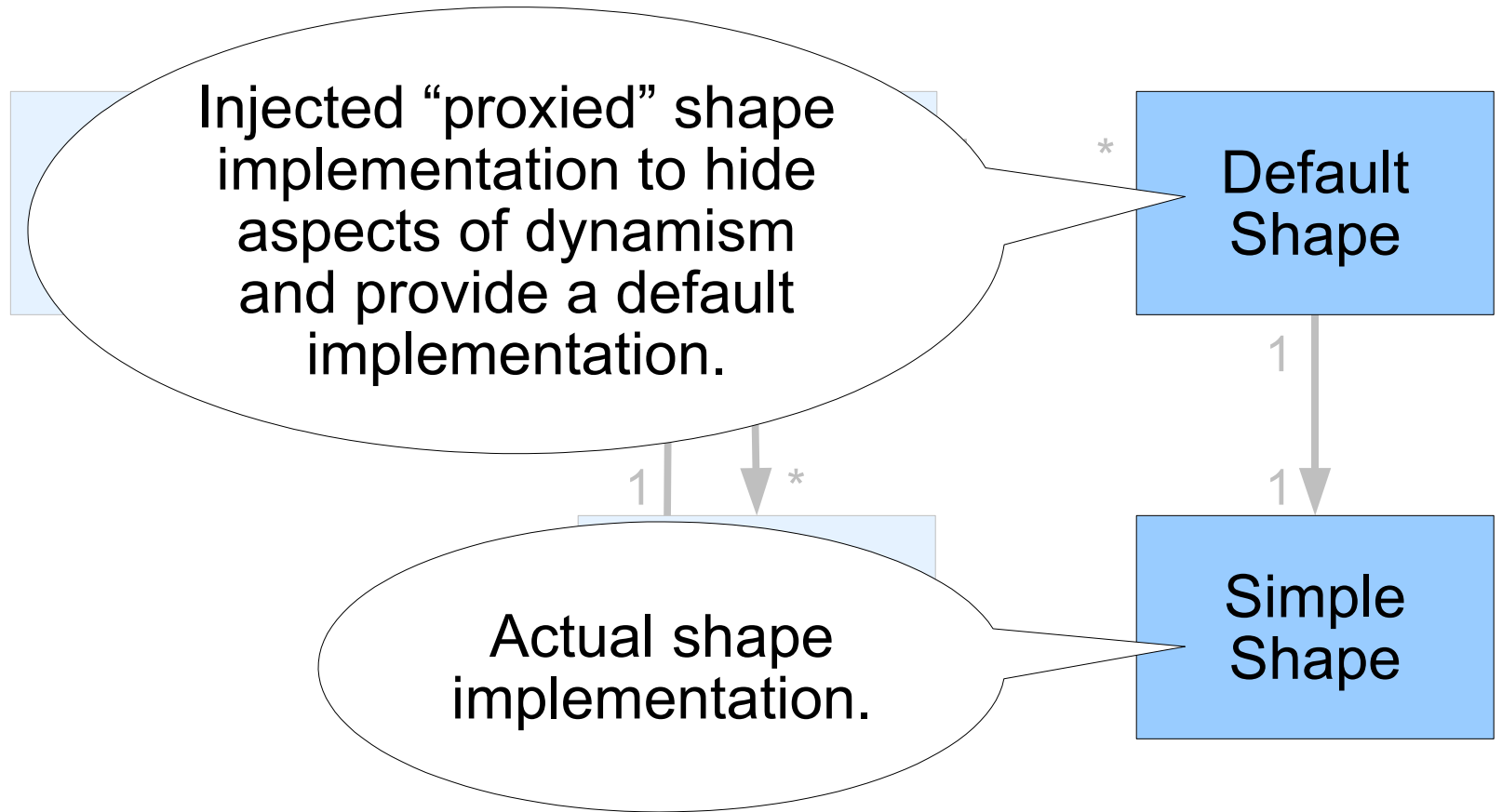
High-Level Architecture



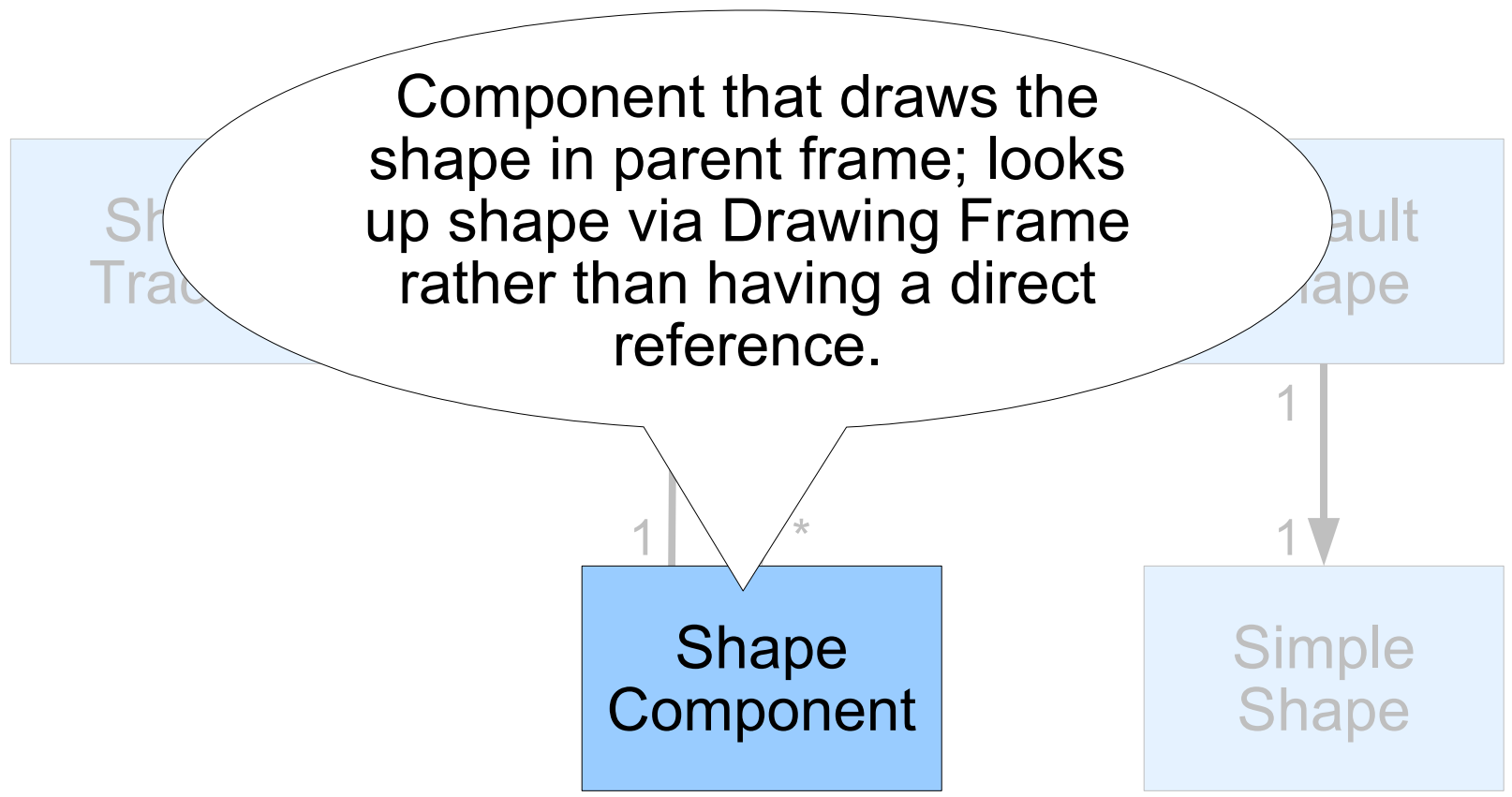
High-Level Architecture



High-Level Architecture



High-Level Architecture



Implementing the Design

- The design is reasonably complete, but what is the precise approach for implementation?
 - It depends...
 - There are a few approach options when building OSGi-based applications...



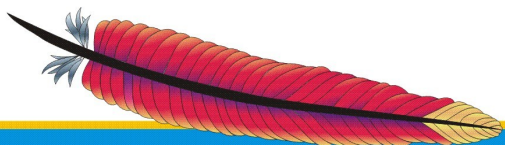
⑤

OSGi Application Approaches

⑥

⑦

⑧



OSGi Application Approaches

- When creating an OSGi-based application there are two main orthogonal issues to consider
 - *Service model* vs. *extension model*
 - *Bundled application* vs. *hosted framework*
- The first issue is related to choosing an OSGi extensibility mechanism
- The second issue is more of an advanced topic to be discussed later, but is related to who is in control of whom



Service vs. Extender Models

- Two different approaches for adding extensibility to an OSGi-based application
 - The **service-based** approach uses the OSGi service concept and the service registry as the extensibility mechanism
 - The **extension-based** approach uses the OSGi installed bundle set as the extensibility mechanism
- Advantages and disadvantages for each
- Can be used independently or together

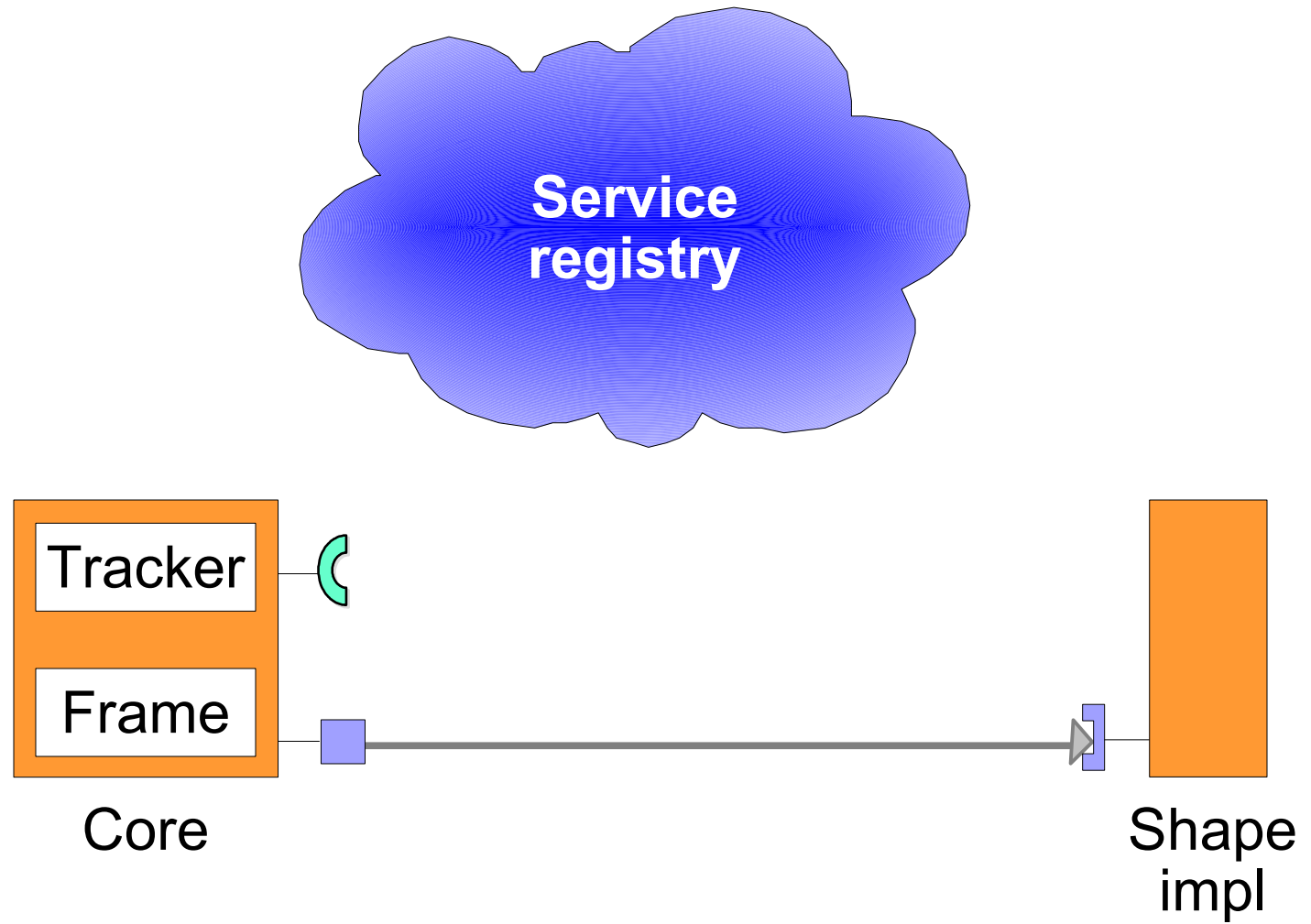


Service-Based Approach

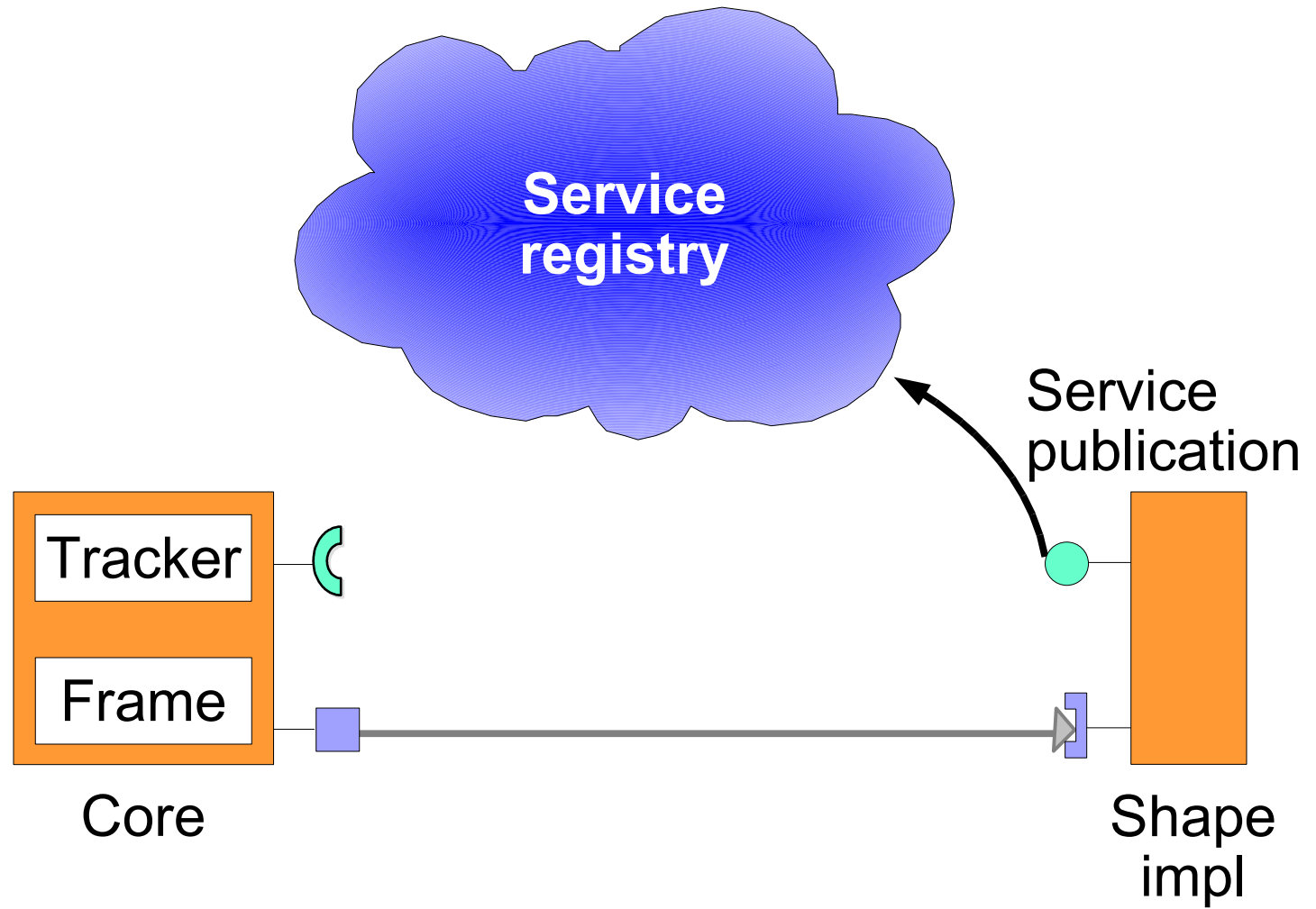
*First, we will examine
the service-based approach*



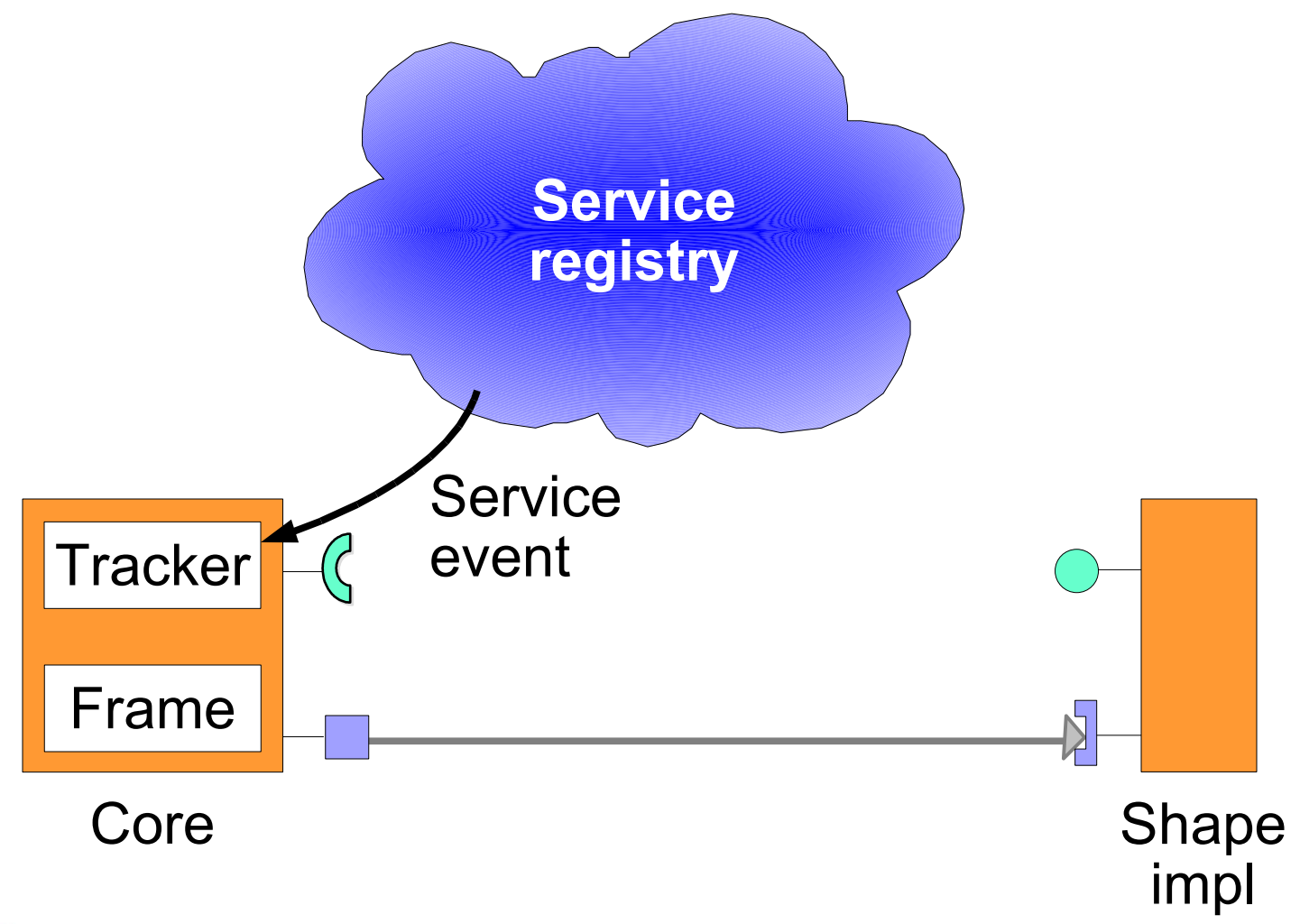
Service-Based Approach



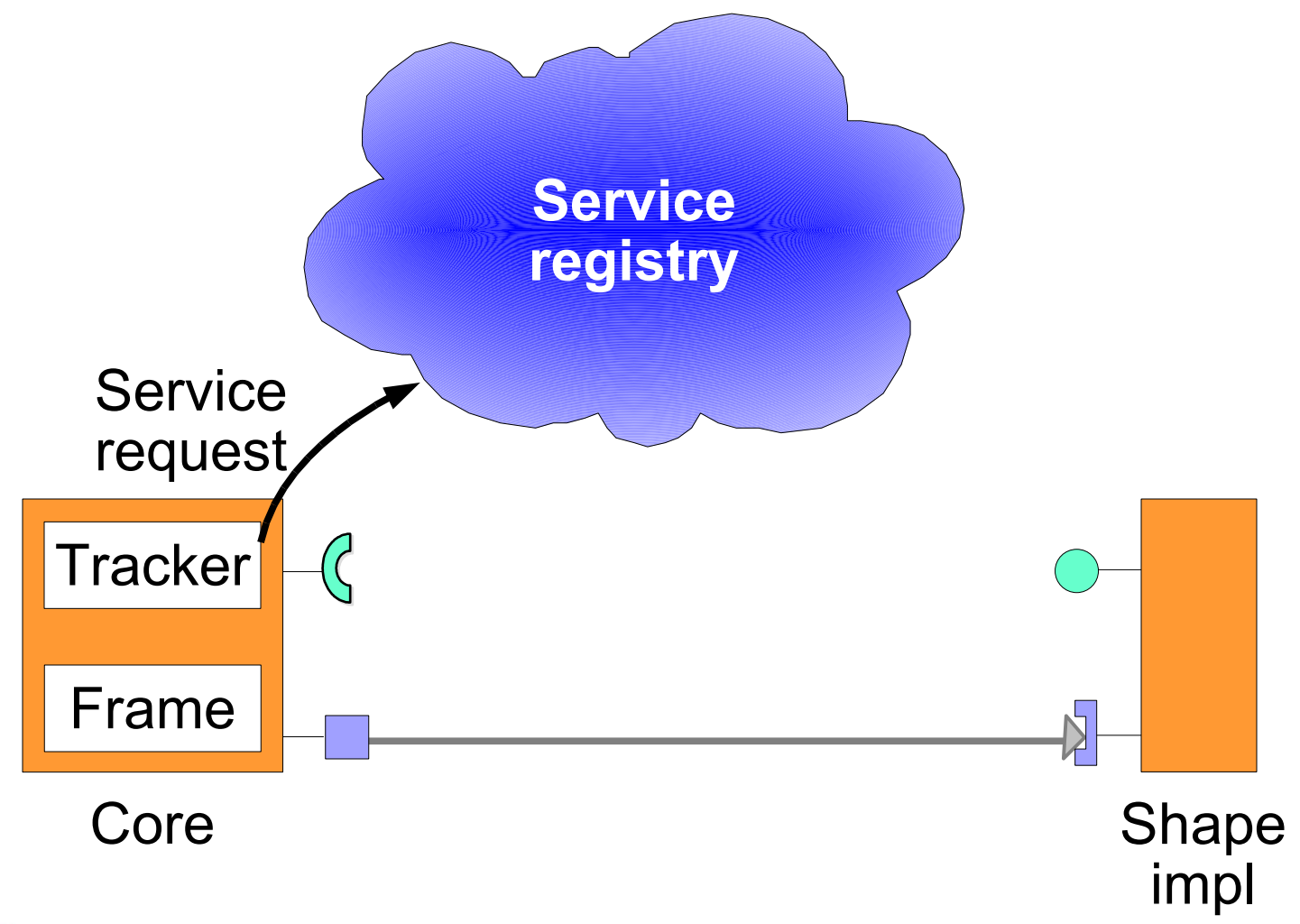
Service-Based Approach



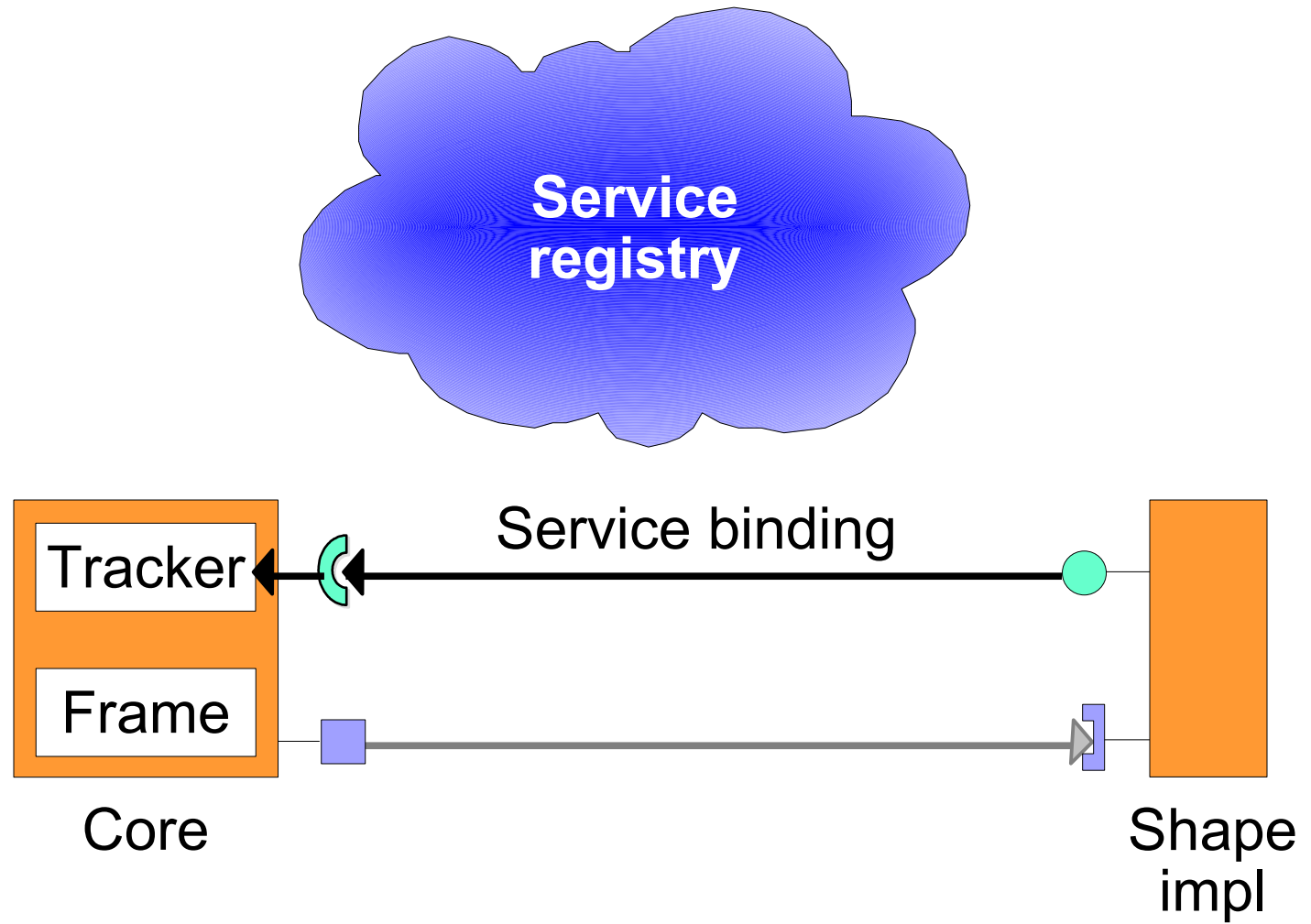
Service-Based Approach



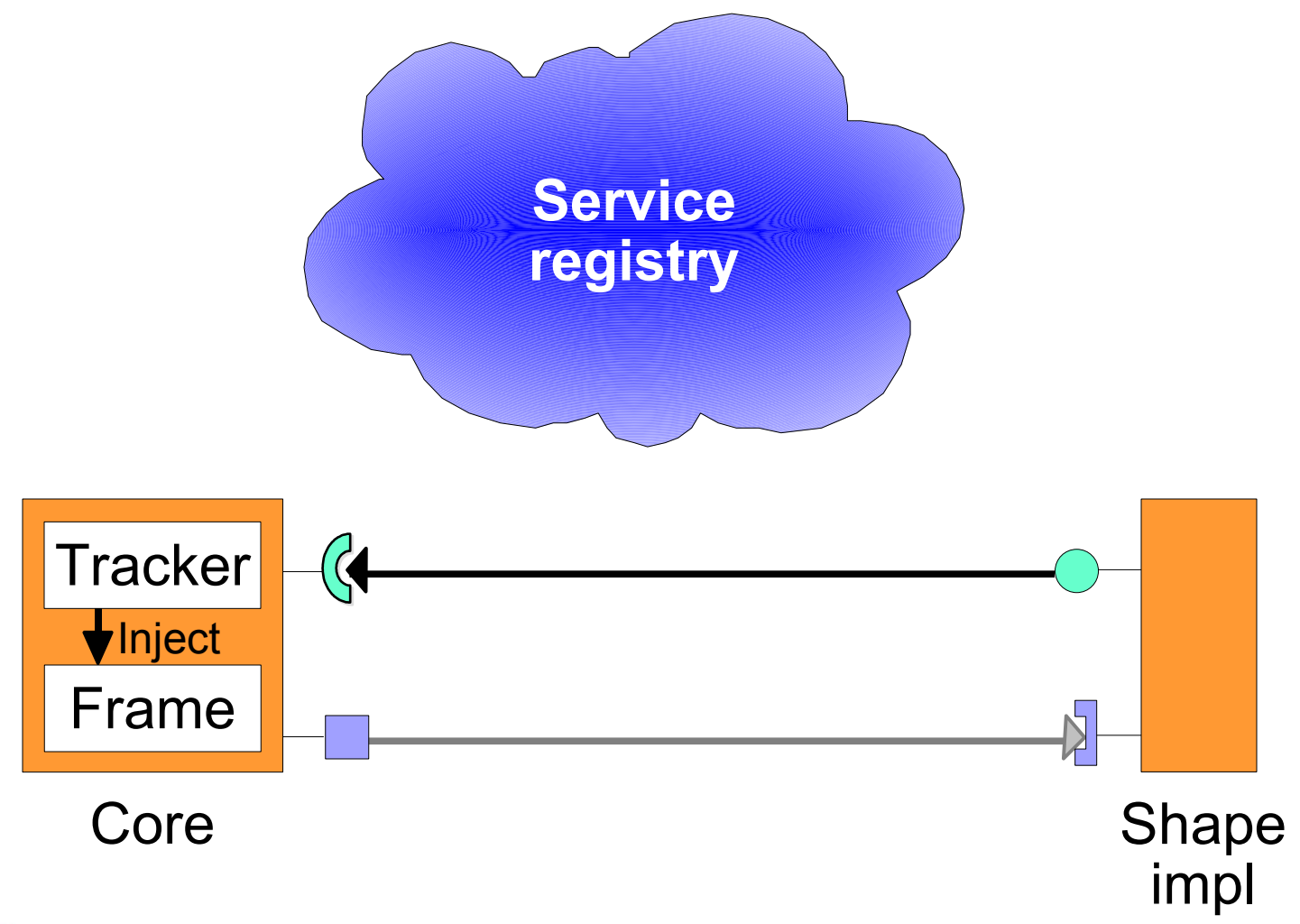
Service-Based Approach



Service-Based Approach

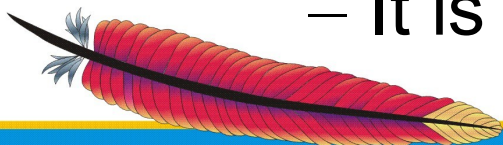


Service-Based Approach



Service Whiteboard Pattern

- **Best practice**
 - Instead of having clients look up and use a service interface, have clients register a service interface to express their interest
 - A component can then track the registered service interfaces and call them when appropriate
- Simple, more robust, leverages the OSGi service model
- This is called the **Whiteboard** pattern
 - It is an *Inversion of Control* pattern

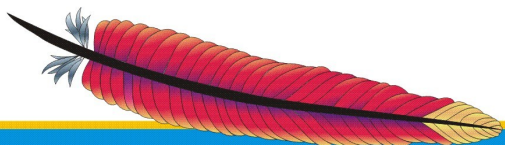


Service-Based Paint Program

- SimpleShape service interface

```
public interface SimpleShape
{
    // A service property for the name of the shape.
    public static final String NAME_PROPERTY
        = "simple.shape.name";
    // A service property for the icon of the shape.
    public static final String ICON_PROPERTY
        = "simple.shape.icon";

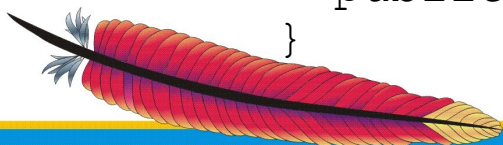
    // Method to draw the shape of the service.
    public void draw(Graphics2D g2, Point p);
}
```



Service-Based Paint Program

- Shape service bundles have an activator to register their service
 - for example...

```
public class Activator implements BundleActivator
{
    public void start(BundleContext context)
    {
        Hashtable dict = new Hashtable();
        dict.put(SimpleShape.NAME_PROPERTY, "Circle");
        dict.put(SimpleShape.ICON_PROPERTY,
            new ImageIcon(
                this.getClass().getResource("circle.png")));
        m_context.registerService(
            SimpleShape.class.getName(), new Circle(), dict);
    }
    public void stop(BundleContext context) { ... }
}
```



Service-Based Shape Tracker

- Recall goal of the Shape Tracker
 - Use IoC principles to inject shapes into application
 - Puts tracking logic in one place
 - Isolates application from OSGi API
- Implemented as an OSGi Service Tracker subclass
 - Uses whiteboard pattern for services
 - Listens for `SimpleShape` service events
 - Resulting from service publications into OSGi service registry



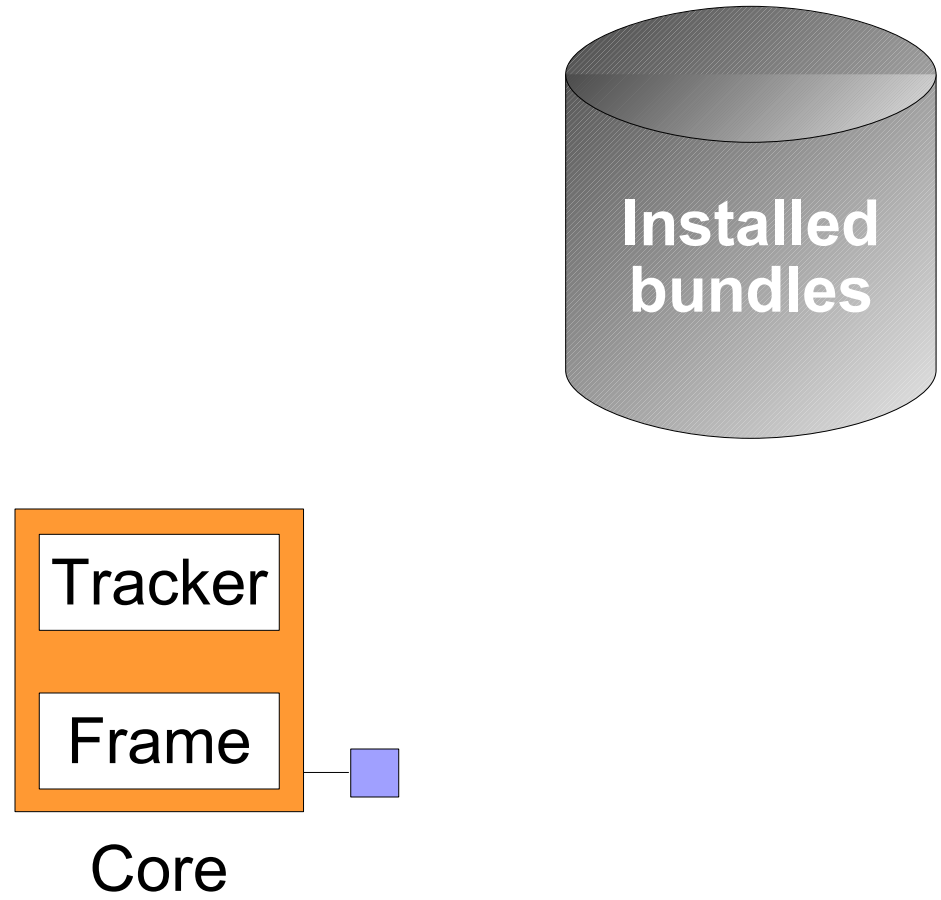
Extension-Based Approach

*Now, we will examine
the extension-based approach*

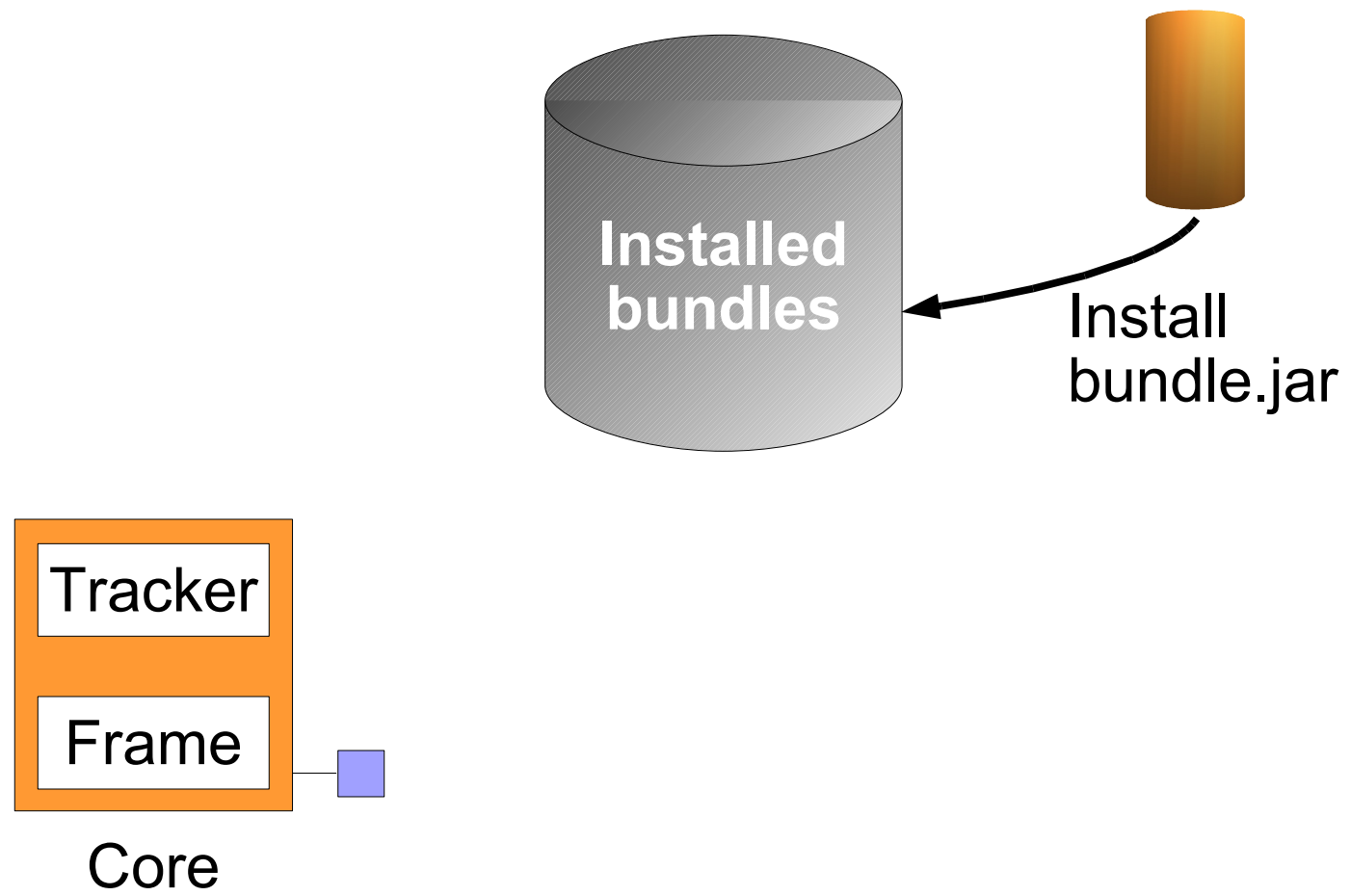
Apache
Con



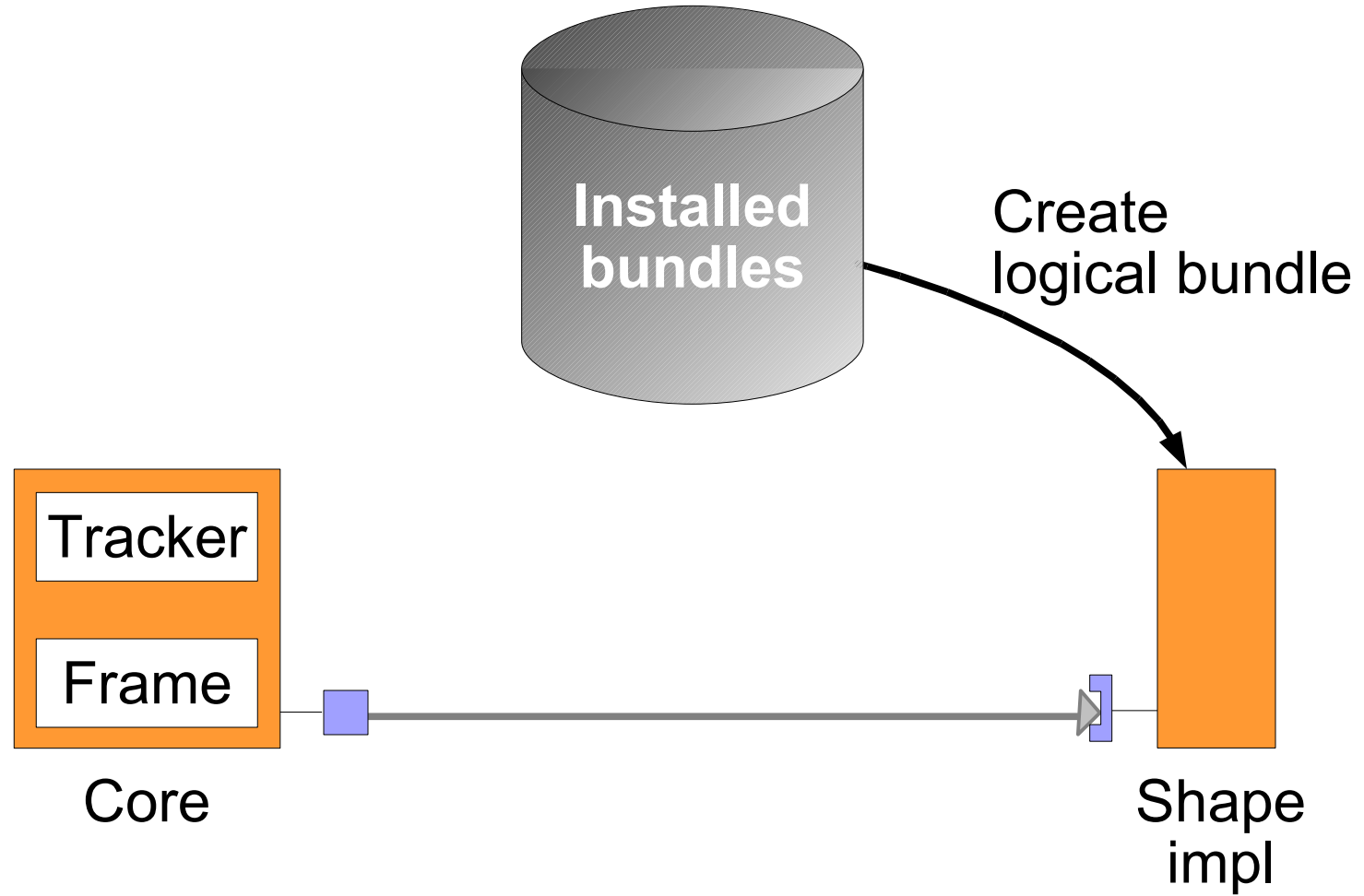
Extension-Based Approach



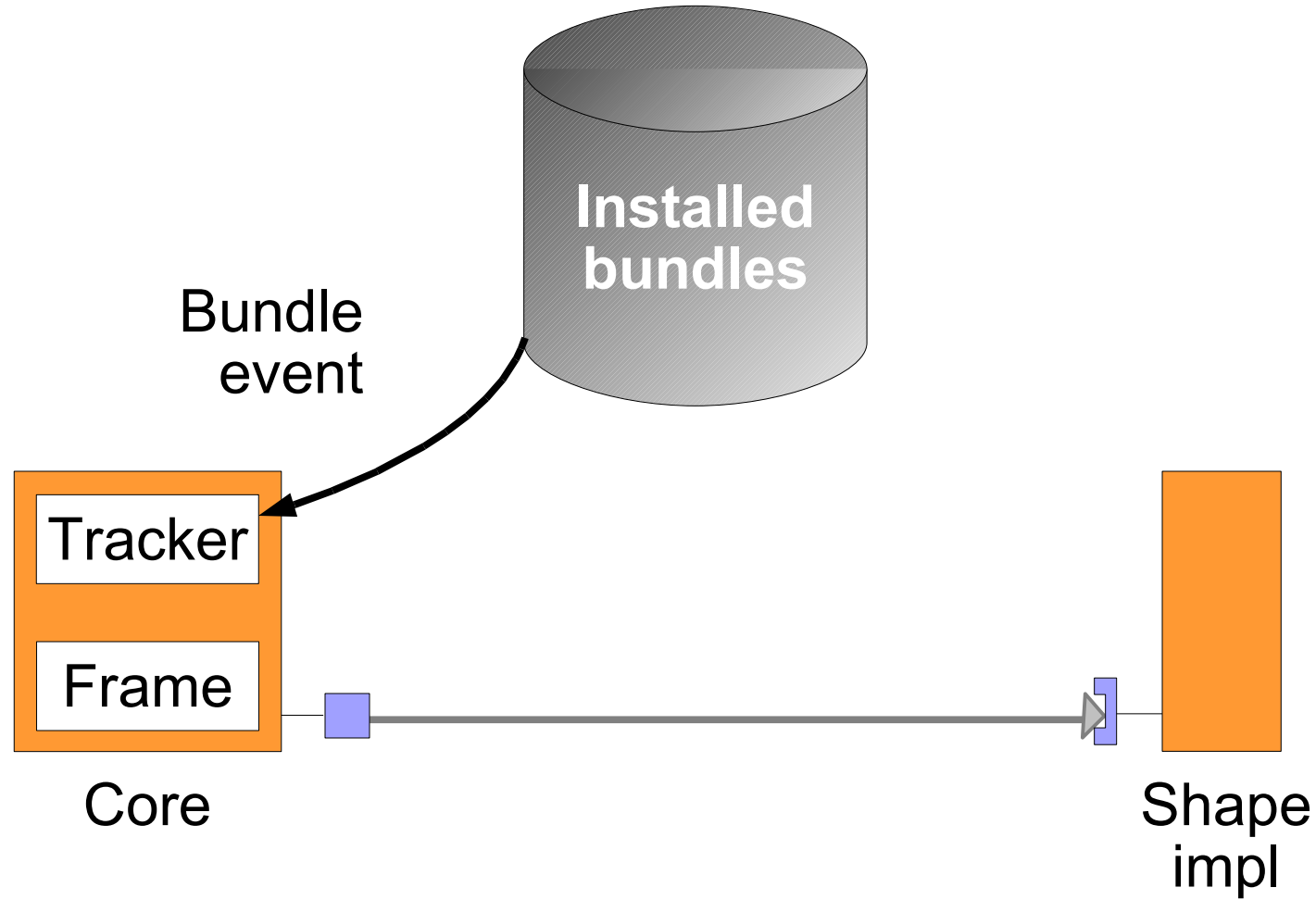
Extension-Based Approach



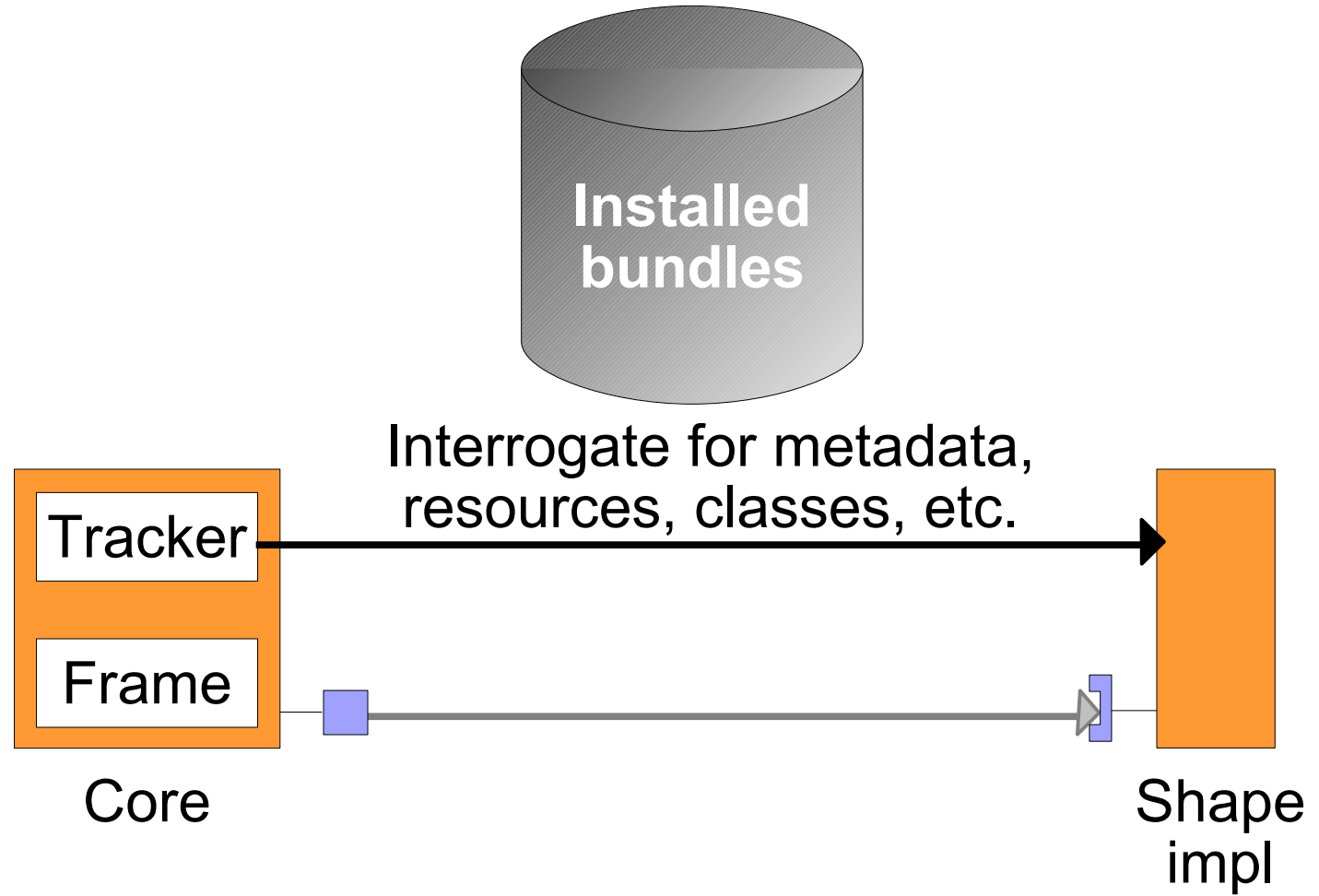
Extension-Based Approach



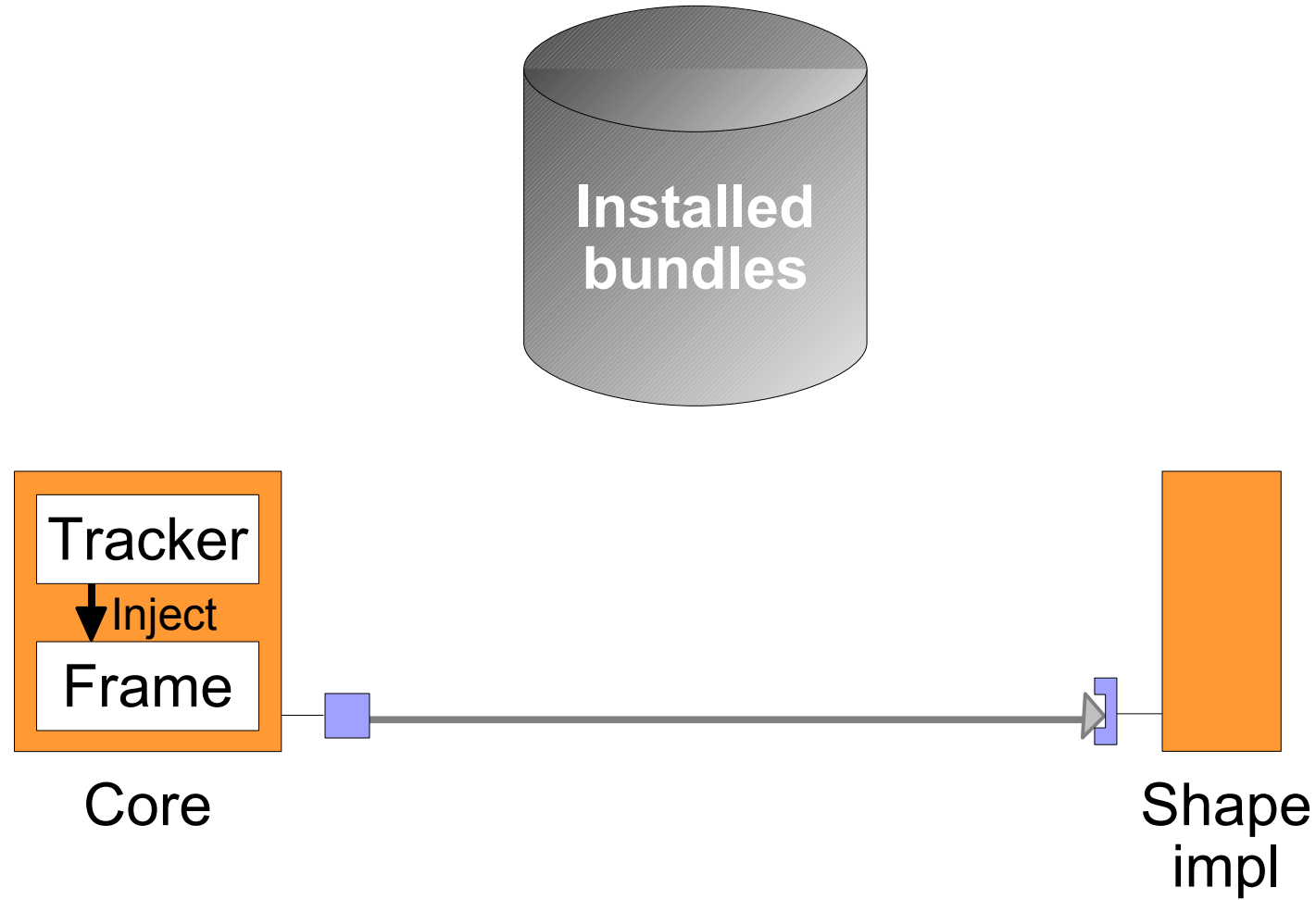
Extension-Based Approach



Extension-Based Approach



Extension-Based Approach

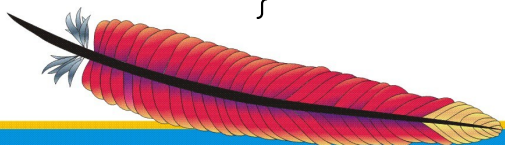


Extension-Based Paint Program

- SimpleShape extension interface

```
public interface SimpleShape
{
    // A property for the name of the shape.
    public static final String NAME_PROPERTY
        = "Extension-Name";
    // A property for the icon of the shape.
    public static final String ICON_PROPERTY
        = "Extension-Icon";
    // A property for the class of the shape.
    public static final String CLASS_PROPERTY
        = "Extension-Class";

    // Method to draw the shape of the extension.
    public void draw(Graphics2D g2, Point p);
}
```



Extension-Based Paint Program

- Shape extension bundles have extension-related metadata in their JAR manifest
 - for example...

...

Extension-Name: Circle

Extension-Icon: org/apache/felix/circle/circle.png

Extension-Class: org.apache.felix.circle.Circle

...



Extension-Based Shape Tracker

- Recall goal of the Shape Tracker
 - Use Inversion of Control principles to inject shapes into application
 - Puts tracking logic in one place
 - Isolates application from OSGi API
- Implemented as custom “bundle tracker”
 - Uses pattern similar to whiteboard, but for installed bundles instead of services
 - Listens for bundle events
 - Specifically, `STARTED` and `STOPPED` events
 - Probes bundle manifests to see if bundles provide shape extensions



Paint Program Implementations

- Both approaches implemented for demonstration
- Bulk of application is the same
 - The main difference is in the implementation of the shaper trackers
- Shape implementations are the same
 - The only difference is that service-based shapes need activators to register their associated services
 - Could be eliminated using DS or IPOJO



Paint Program Implementations

- To be clear here, there is *no magic*
 - You cannot make an application extensible if it is not designed to be so
 - Likewise, it cannot be extended in ways that it does not expect
 - However, it is possible to make expected extensions further extensible (e.g., a plugin could have plugins)
 - In this example, the core paint program was designed for shape extensibility
 - However, it does not dictate how this extensibility is achieved
 - Thus, we were able to use OSGi technology and two different approaches to achieve it



Packaging the Paint Program

- Shape implementations and the core application are packaged in a similar fashion for both service and extension approaches
 - As a bundle JAR file with metadata
- ***Best practice***
 - Separate public API into separate packages
 - Only export public API packages in your metadata

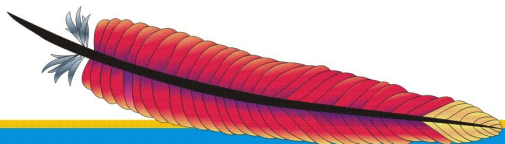


⑥

Example Application Demo

⑦

⑧



7

Advanced Issues

8



Bundled vs. Hosted

- Applications can leverage OSGi functionality in two ways
 - ***Bundled application***
 - Build entire application as a set of bundles that will run on top of a framework instance
 - ***Hosted framework***
 - Host a framework instance inside the application and externally interact with bundles/services



Bundled vs. Hosted

- Building your application as a set of bundles is the ***preferred approach***
 - Allows all parts of application to benefit from OSGi modularity and dynamism
 - Allows application to run on any framework
 - However, it is not always possible to bundle application, e.g., legacy situations
- Hosted framework approach allows piecemeal OSGi adoption
 - Will likely tie application to a framework implementation



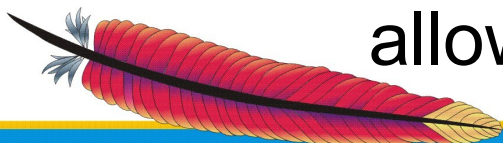
Hosted Framework

- More complicated due to external/internal gap between application and framework
 - e.g., unlike bundles, the host application does not have a bundle context by which it can access framework services
- Required host/framework interactions
 - Accessing framework functionality
 - Providing services to bundles
 - Using services from bundles



Hosted Framework

- Felix tries to simplify hosted framework scenarios
 - All configuration data is passed into framework constructor
 - Felix framework implements `Bundle` interface and acts as the “system bundle”
 - Gives the host application an intuitive way to access framework functionality
 - Felix constructor also accepts “constructor activators” to extend system bundle
 - Felix tries to multiplex singleton resources to allow for multiple framework instances



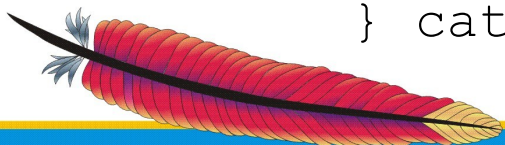
Hosted Framework

```
// Define configuration properties
Map configMap = new StringMap(false);
configMap.put(..., ...);

...
// Create application activators
List list = new ArrayList();
list.add(new Activator());

try {
    // Create a framework instance
    Felix felix = new Felix(configMap, list);
    // Start framework instance
    felix.start();

    ...
    // Stop framework instance
    felix.stop();
} catch (Exception ex) { ... }
```



Hosted Framework

- Providing a host application service

```
BundleContext bc = felix.getBundleContext();  
bc.registerService(Service.class, svcObj, null);
```



Hosted Framework

- Providing a host application service

```
BundleContext bc = felix.getBundleContext();  
bc.registerService(Service.class, svcObj, null);
```

- Accessing internal bundle services

```
BundleContext bc = felix.getBundleContext();  
ServiceReference ref =  
    bc.getServiceReference(Service.class);  
Service svcObj = (Service) bc.getService(ref);
```



Hosted Framework

- Providing a host application service

```
BundleContext bc = felix.getBundleContext();
bc.registerService(Service.class, svcObj, null);
```

- Accessing internal bundle services

```
BundleContext bc = felix.getBundleContext();
ServiceReference ref =
    bc.getServiceReference(Service.class);
Service svcObj = (Service) bc.getService(ref);
```

- Better approach is to use a constructor activator since it is integrated with system bundle (i.e., framework) starting and stopping



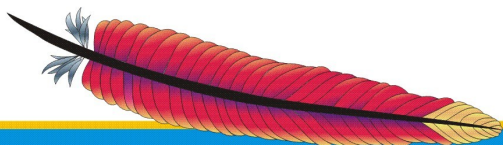
Hosted Framework

- Classes shared among host application and bundles **must** be on the application class path
 - Disadvantage of hosted framework approach, which limits dynamics
 - Use of reflection by host to access bundle services can eliminate this issue, but it is still not an optimal solution
- In summary, better to completely bundle your application if possible



Hosted Framework Paint Program

- This approach was also implemented for the paint program
 - For both approaches (i.e., service-based and extender-based)
- Simply added a static main to the activator of the main application bundle
 - This means they can be used as either a bundled application or a hosted framework



Custom Life Cycle Layer

- As mentioned previously, it is possible to use the module layer without the service layer
 - Some projects have their own component models with different life cycle and interaction mechanisms
- The extension model can be used to create custom life cycle and component interaction mechanisms
 - The Eclipse plugin model is actually an example of this



Custom Life Cycle Layer

- OSGi framework still manages bundle resolution
 - Bundle activation becomes a no-op
- Special extension bundle must implement custom life cycle layer
 - Probes bundles to discover custom life cycle interfaces/entry points
 - Exposes how to initialize, start, stop, custom components
 - Injects custom bindings and/or container context



8

Conclusion



Conclusion (1/2)

- Today's applications require high degrees of modularity and dynamic extensibility
 - Java provides infrastructure, but no direct support
- OSGi technology addresses Java's limitations in these areas
 - Available today and growing in importance
- Developing OSGi-based applications is straightforward and provides immediate benefits



Conclusion (2/2)

- OSGi technology is flexible and can be leveraged in different ways to meet your requirements
- Apache Felix is ready when you are
 - Stable and in regular use
- Paint Program application is available
 - <http://cwiki.apache.org/FELIX/apache-felix-application-demonstration.html>



ApacheCon

Questions?



Leading the Wave
of Open Source