JAOO conference 2006 October 1 - 6 Denmark

# Programming with Equinox

## The OSGi foundation for Eclipse

Jeff McAffer, IBM Rational
Tom Watson, IBM Lotus

# Contents


- Setup
- Introduction to OSGi
- Managing your Target Environment
- The Equinox/OSGi Development Model
- OSGi Basics
- Components
- Services
- Remoting
- Conclusion

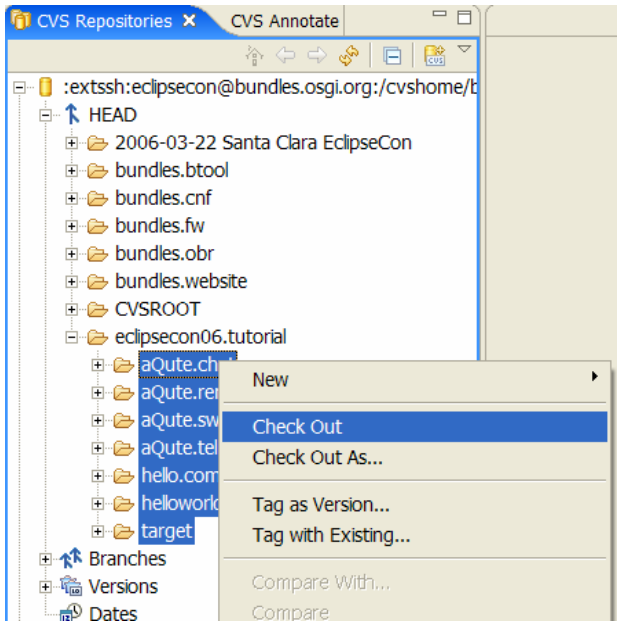
# Your Infrastructure

- You need to have the following software installed on your machine in a new workspace:
  - Eclipse SDK 3.2 (http://eclipse.org)
  - The tutorial projects from CVS:
    - Server: bundles.osgi.org
    - Repository /cvshome/bundles
    - User apachecon
    - Password 2006
    - Projects Select all projects under Tutorial

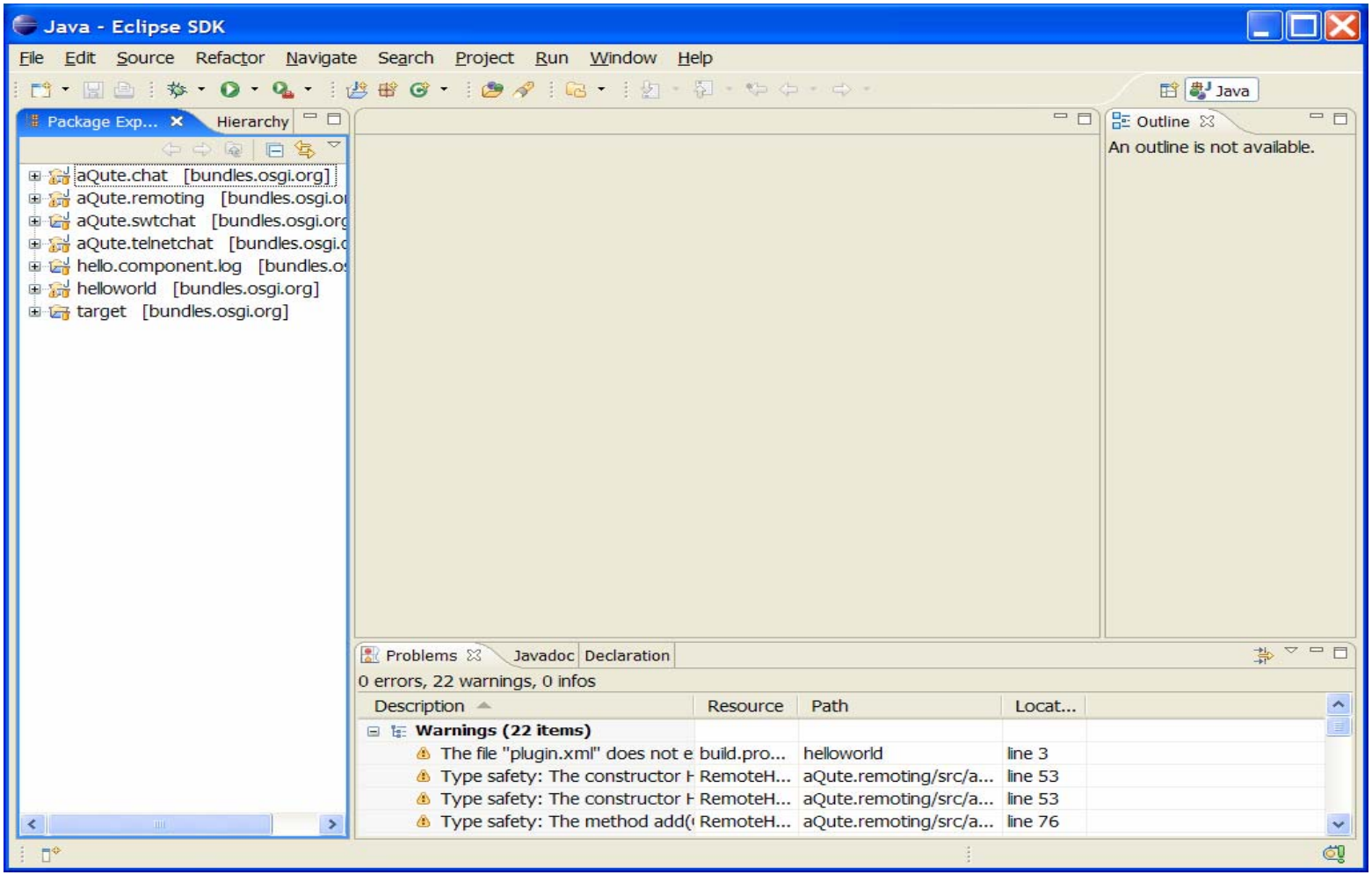
 

# Loading the tutorial projects from CVS

- Window > Open Perspective > Other > CVS Repository Exploring
- In CVS Repository view context menu: New > Repository Location
  - Fill in the necessary CVS Repository information
- In CVS Repositories view, expand: HEAD/Tutorial
- Select all projects under Tutorial and choose Check Out



# Your Workspace (more or less)



# Section I - OSGi Background

# What is the OSGi service platform?

- A Java™ framework for developing remotely deployed service applications, that require:
  - Reliability
  - Large scale distribution
  - Wide range of devices
  - Collaborative
- Created through collaboration of industry leaders
- Spec 4.0 publicly available at www.osgi.org …
- The Dynamic Modularity Layer for Java!
- Cool!

# Why the OSGi Service Platform?

- What problems does the OSGi Service Platform address?
- A unified software market:
  - The limited (binary) software portability problem
  - The complexity of building heterogeneous software systems
    - Supporting the myriad of configuration, variations, and customizations required by today's software and hardware
  - Managing the software life-cycle

# Limited Binary Software Portability

- Lack of portability causes
  - Market friction: No large market of reusable components and applications
  - Reduced quality
- Unnecessary constraints on hardware and software architectures
  - CPUs differ widely in cost and performance
  - Linux™ is nice, but it is sub-optimal for smaller devices
- Benefits of the OSGi Platform
  - Applications run unmodified on different hardware and software architectures

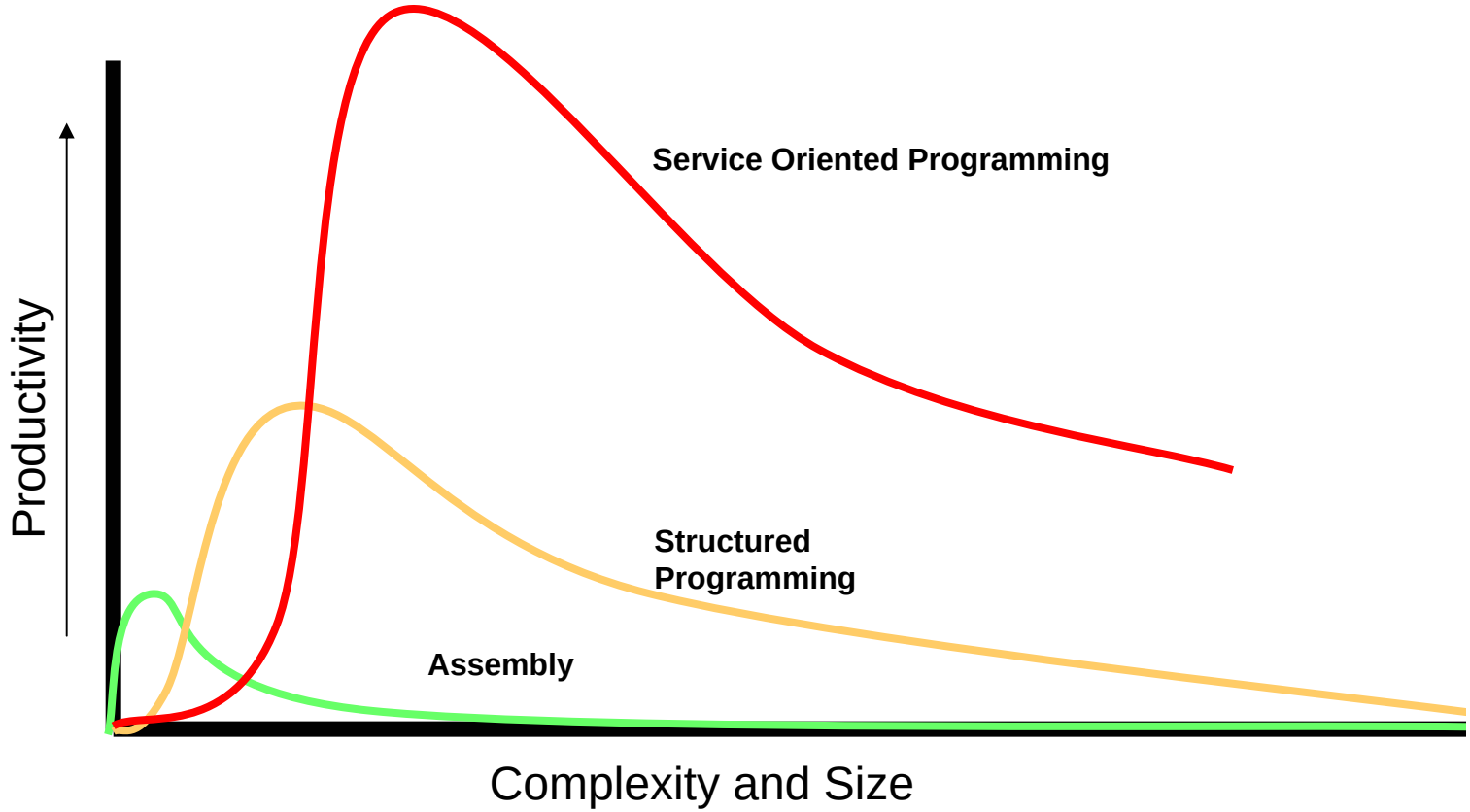
# Complexity of Software

- A DVD player can contain 1 Million lines of code
  - Comparison: Space Shuttle ~ 0.5 Million
- A BMW car can contain up to 50 networked computerized devices
- Eclipse contains 2.5 million lines of code
- An average programmer writes an average of 10 lines a day …

- Houston … we have a problem

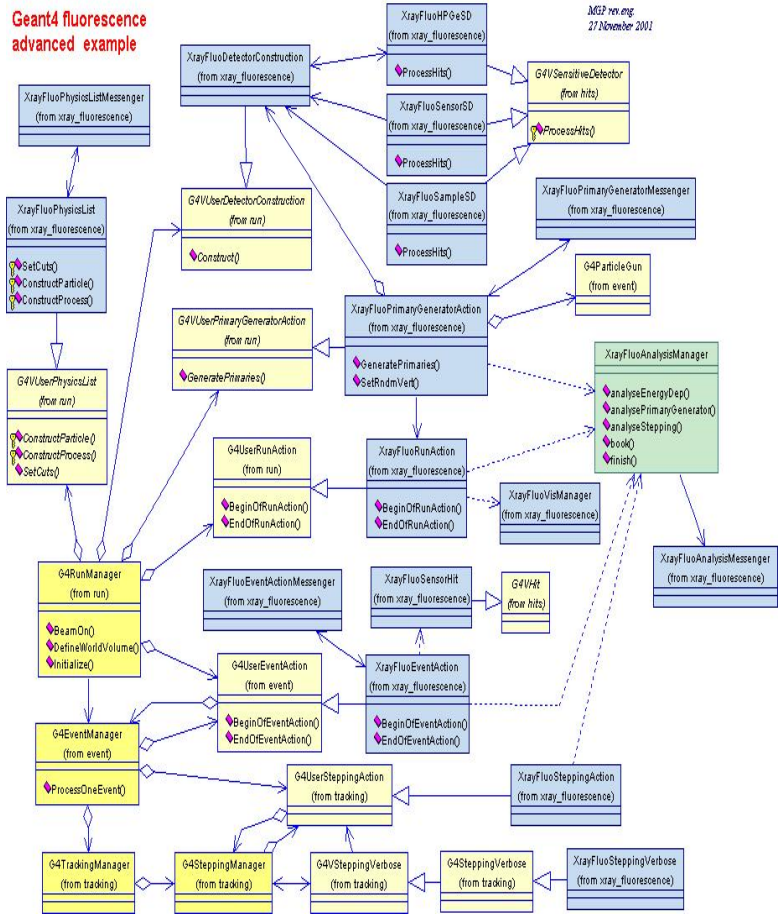
 

# Complexity of Software



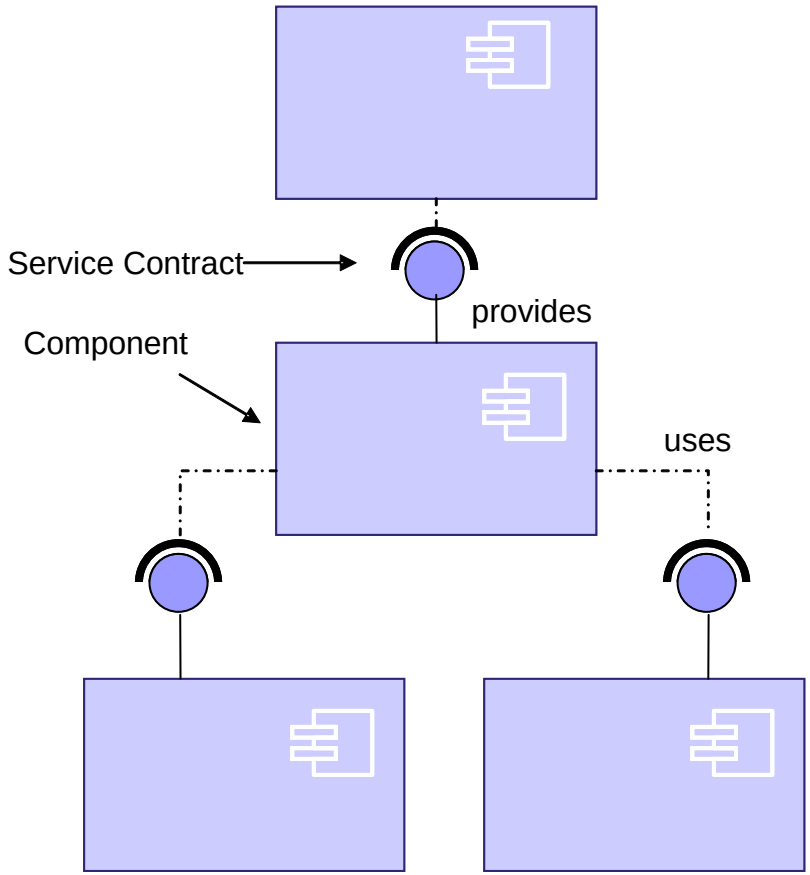
# Limits Object Oriented Technology

- Objects are great, but oh, the tangled webs we weaves …
- Coupling severely limits reusability
  - Using a generic object, can drag in a large number of other objects
- Creates overly large systems after a certain complexity is reached
- Flexibility must be built in by the programmer
  - *Plug-in* architectures



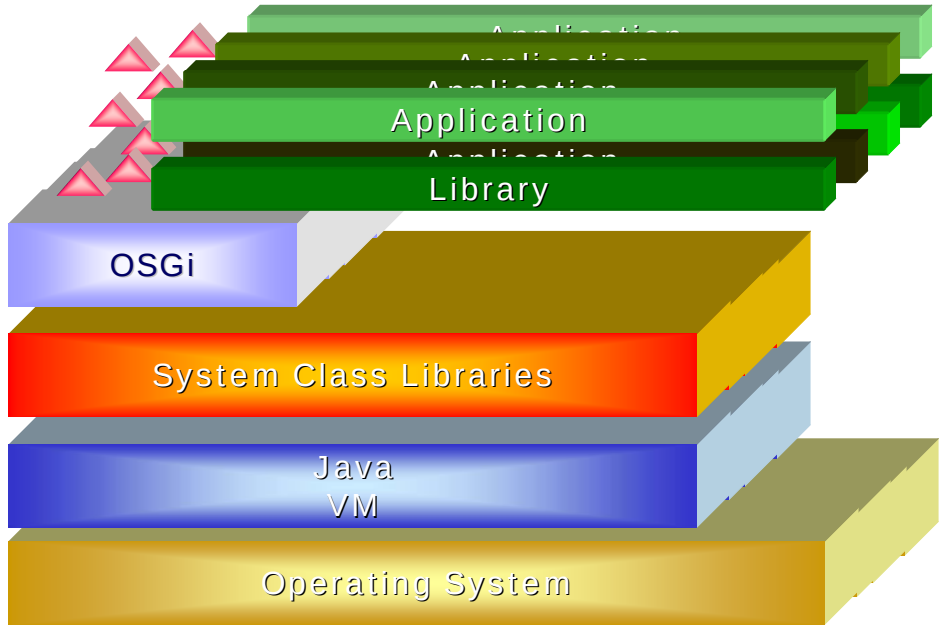
# Service Oriented Architectures

- Separate the contract from the implementation
- Allows alternate implementations
- Dynamically discover and bind available implementations
- Based on contract (interface)
- Components are reusable
- Not coupled to implementation details



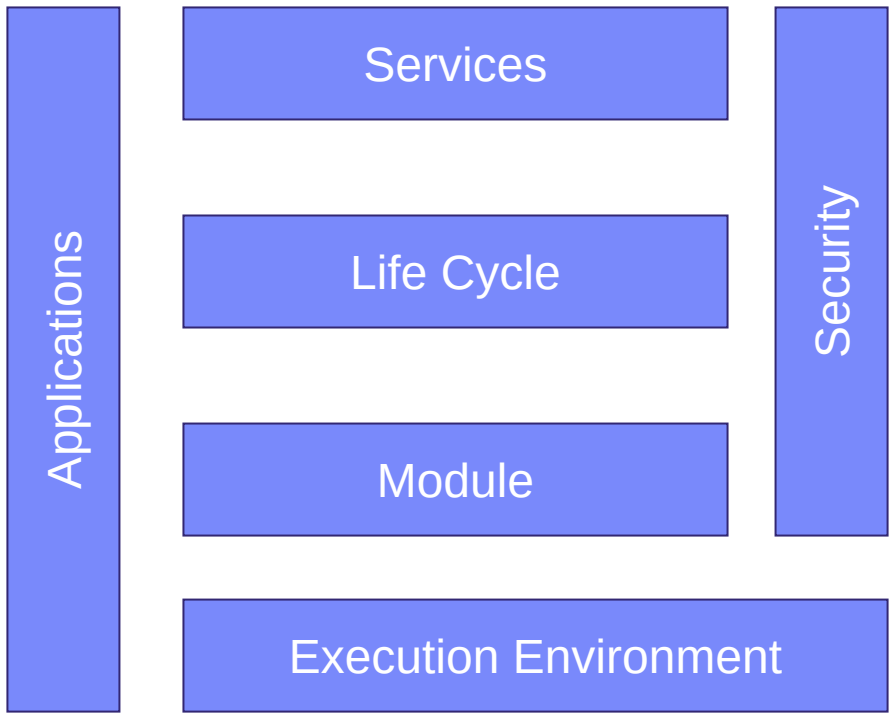
# Framework

- Allows applications to share a single Java VM
- Classloading
- Isolation/Security
- Communication `& Collaborations between applications
- Life cycle management
- Policy free
  - Policies are provided by bundles
- API is fully self managed



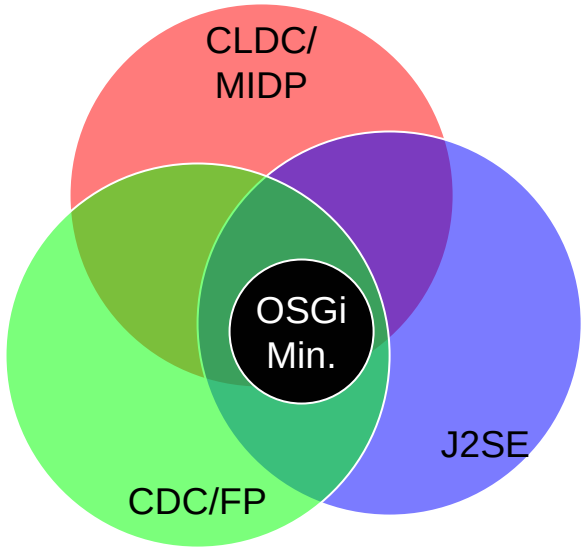
# Layering



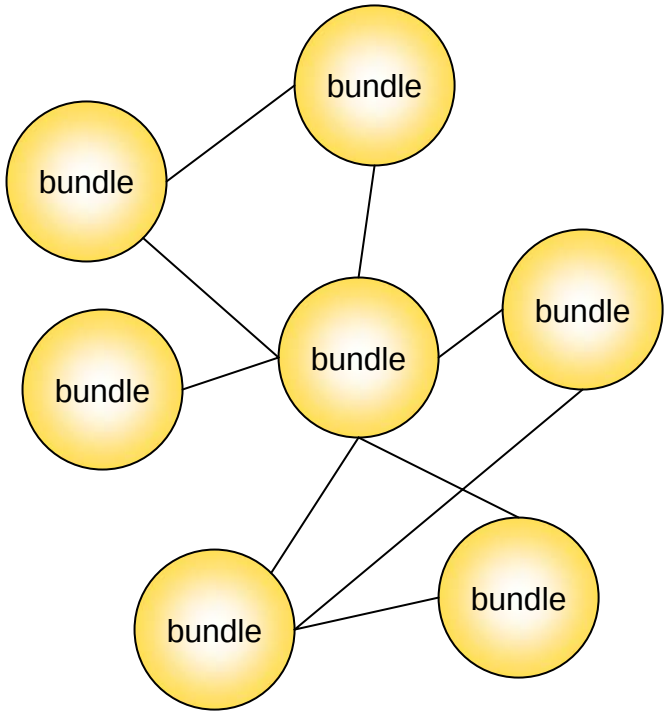
# Execution Environment

- OSGi APIs only use a subset of J2SE and J2ME CDC
  - OSGi Minimum EE
- Matches most profiles
- Implementations can use more than the OSGi Minimum EE
- Security is not mandatory
- CLDC is possible if class loaders are added in a device specific way



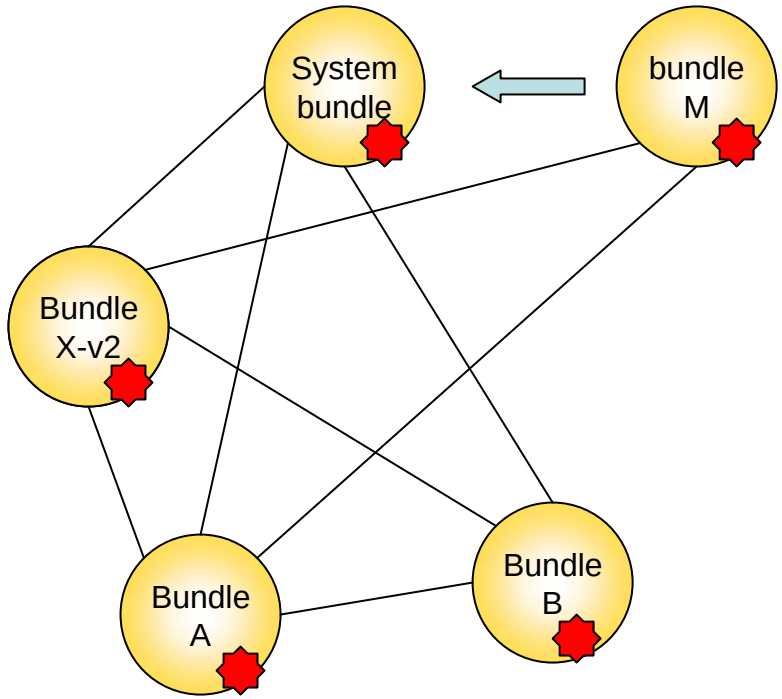
# Module Layer

- Packaging of applications and libraries in *Bundles*
  - Raw Java has significant deployment issues
- Class Loading modularization
  - Raw Java provides the Class Path as an ordered search list, which makes it hard to control multiple applications
- Protection
  - Raw Java can not protect certain packages and classes
- Versioning
  - Raw Java can not handle multiple versions of the same package



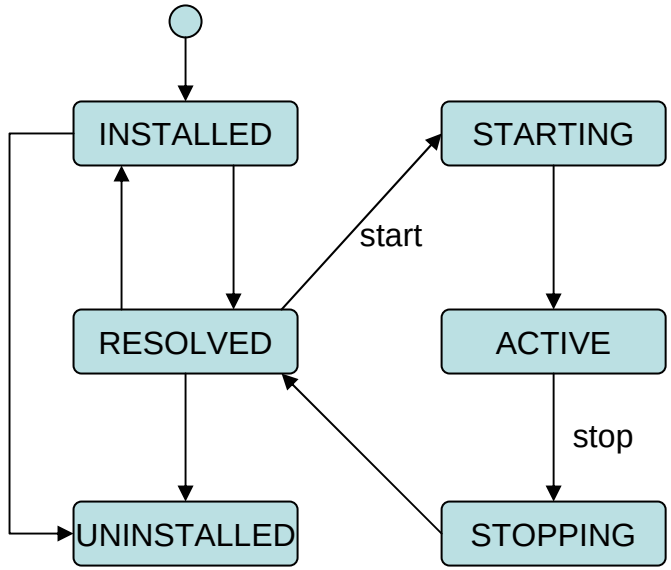
# Life Cycle Layer

- System Bundle represents the OSGi Framework
- Provides an API for managing bundles
  - Install
  - Resolve
  - Start
  - Stop
  - Refresh
  - Update
  - Uninstall
- Based on the module layer



# Life Cycle Layer

- Bundle is started by the Bundle Activator class
- Header in Manifest refers to this class
- Interface has 2 methods
  - Start: Initialize and return immediate
  - Stop: Cleanup
- The Activator gets a Bundle Context that provides access to the Framework functions
- Framework provides Start Level service to control the start/stop of groups of applications



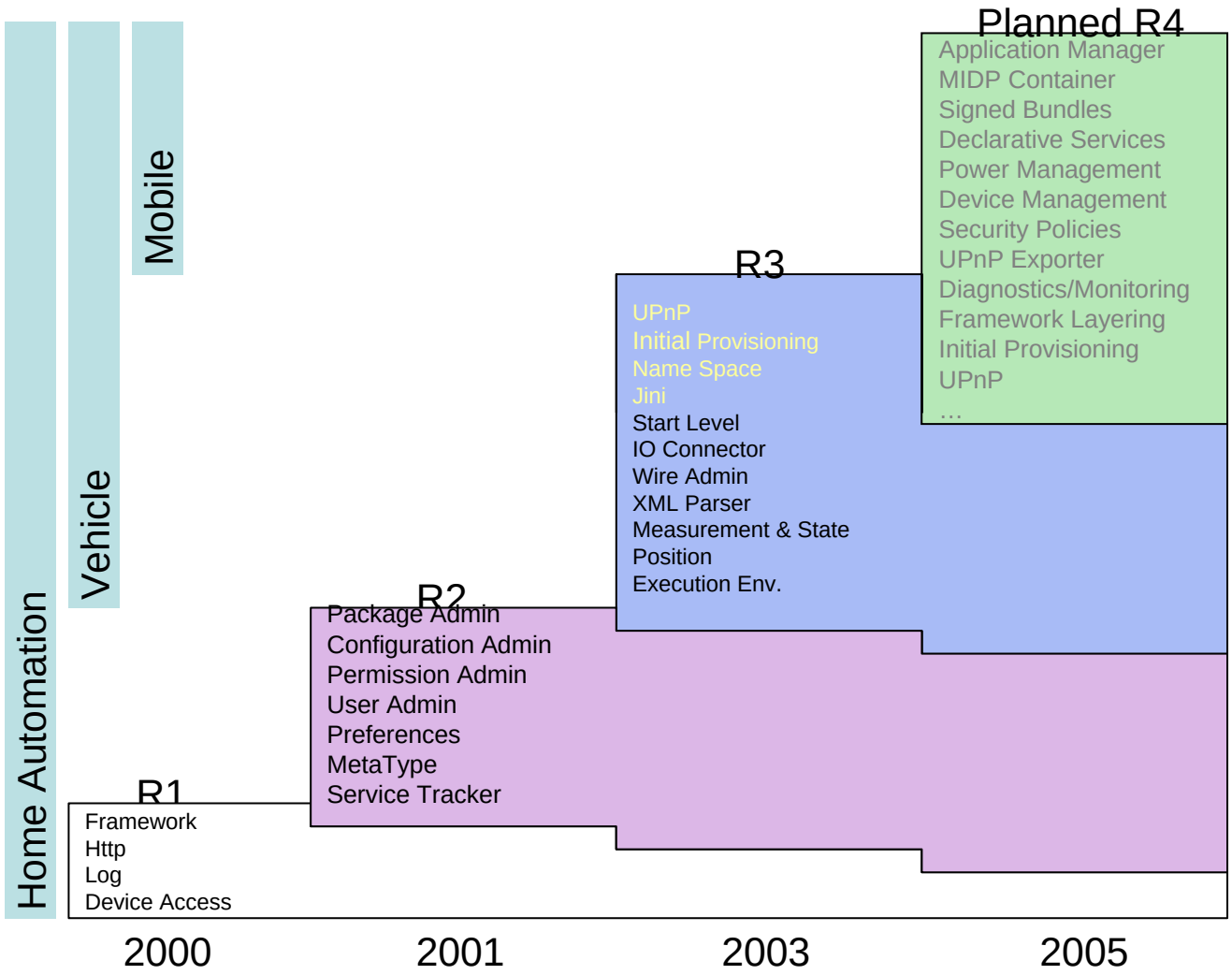

 

# Service Layer

- Provides an in-VM service model
  - Discover (and get notified about) services based on their interface or properties
  - Bind to one or more services by
    - program control,
    - default rules, or
    - deployment configuration
- SOA Confusion
  - Web services bind and discover over the net
  - The OSGi Service Platform binds and discovers inside a Java VM
- The OSGi Alliance provides many standardized services

# Evolution


Home Automation
Vehicle
Mobile
R1
Framework
Http
Log
Device Access
R2
Package Admin
Configuration Admin
Permission Admin
User Admin
Preferences
MetaType
Service Tracker
R3
UPnP
Initial Provisioning
Name Space
Jini
Start Level
IO Connector
Wire Admin
XML Parser
Measurement & State
Position
Execution Env.
Planned R4
Application Manager
MIDP Container
Signed Bundles
Declarative Services
Power Management
Device Management
Security Policies
UPnP Exporter
Diagnostics/Monitoring
Framework Layering
Initial Provisioning
UPnP
...
2000
2001
2003
2005

# Benefits of Using the OSGi Service Platform

- Components are smaller
  - Easier to make
- Components are not coupled to other components
  - Gives reusability
- Excellent model for the myriad of customizations and variation that are required of today's devices
- Collaborative model
  - Allows reuse of other components for most problems

# Section II – Equinox and Eclipse

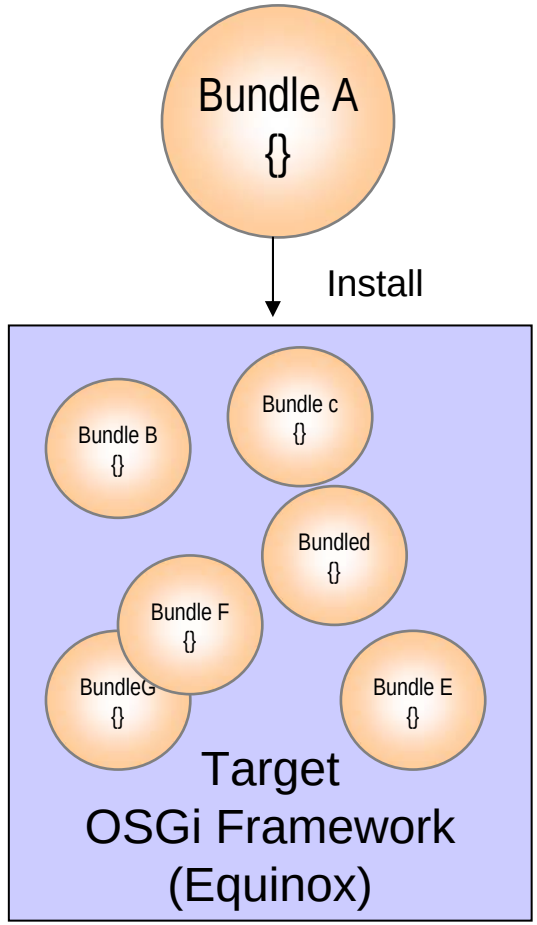
# What is Equinox ?

- An open source community focused on OSGi Technology
  - http://www.eclipse.org/equinox/
  - Develop OSGi specification implementations
  - Prototype ideas related to OSGi
- An OSGi Framework implementation
  - Core of the Eclipse runtime
  - Provides the base for Eclipse plug-in collaboration
  - Fully compatible with the OSGi R4 specification
- New for Eclipse 3.2 – Other specification implementations
  - Device Manager, Declarative Services, Event Admin, HTTP Service, Log Service, Metatype Service, Preferences Service, User Admin, Wire Admin – More on the way!!

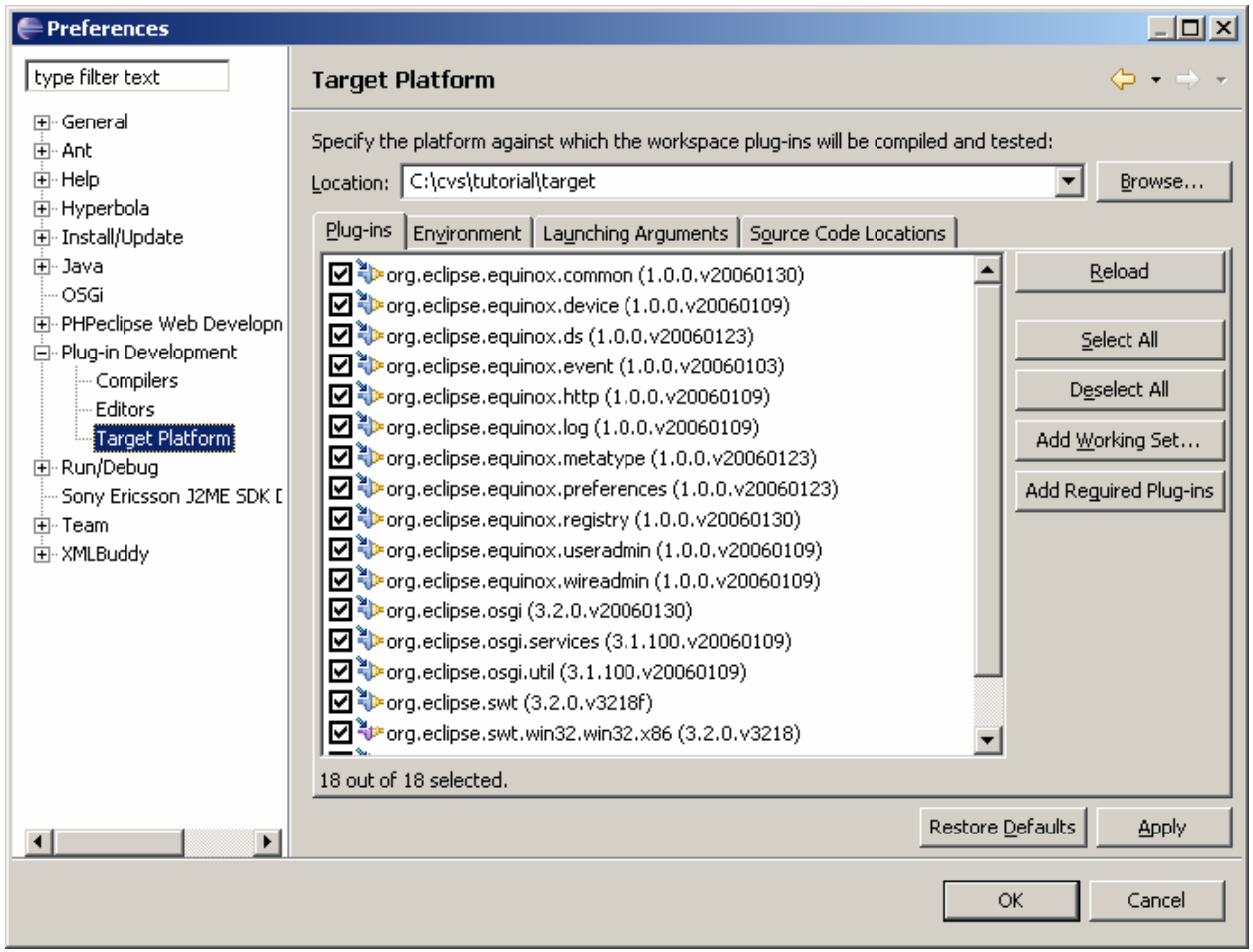
# The Equinox Target Environment

- Eclipse makes it *easy* to develop for all OSGi Service Platforms
- A *target platform*
  - Contains a set of bundles
  - Defines runtime parameters
- To Define the Target Platform, goto:
  - Preferences ->Plug-in Development ->Target Platform
  - Select the `target` project in your workspace as location
- Advanced target management using “Target Definitions” (New->Other->Plug-in Development->Target Definition)



# Setting up the Target Platform



# What Did We Learn

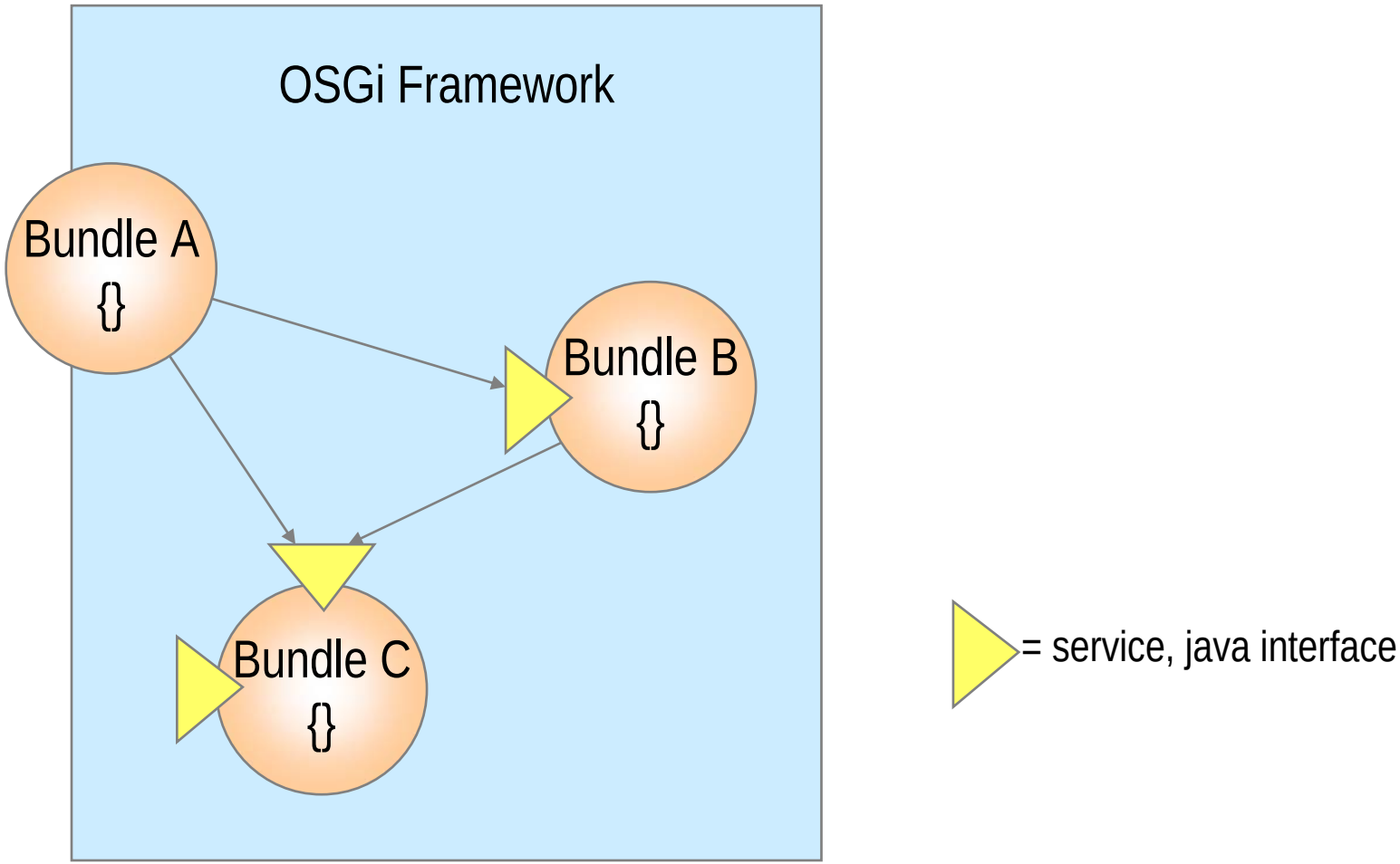
- The OSGi Service Platform is kind of a Java Operating System
- It simplifies:
  - Deployment Problems
  - Software composition
  - Software management
- Eclipse provides a development environment for OSGi Bundles
- Equinox provides open source implementations of the OSGi specifications in the Equinox project

# Section III - Fundamental OSGi concepts

# Framework Entities



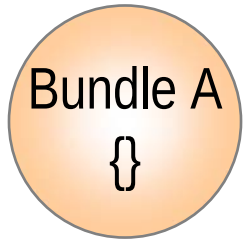
# Bundles

- A *bundle* is the deliverable application
  - Like a Windows™ EXE file
  - Content is a JAR file
- A bundle registers zero or more services
  - A service is specified in a Java interface and may be implemented by multiple bundles
  - Services are bound to the bundle life-cycle
- Searches can be used to find services registered by other bundles
  - Query language

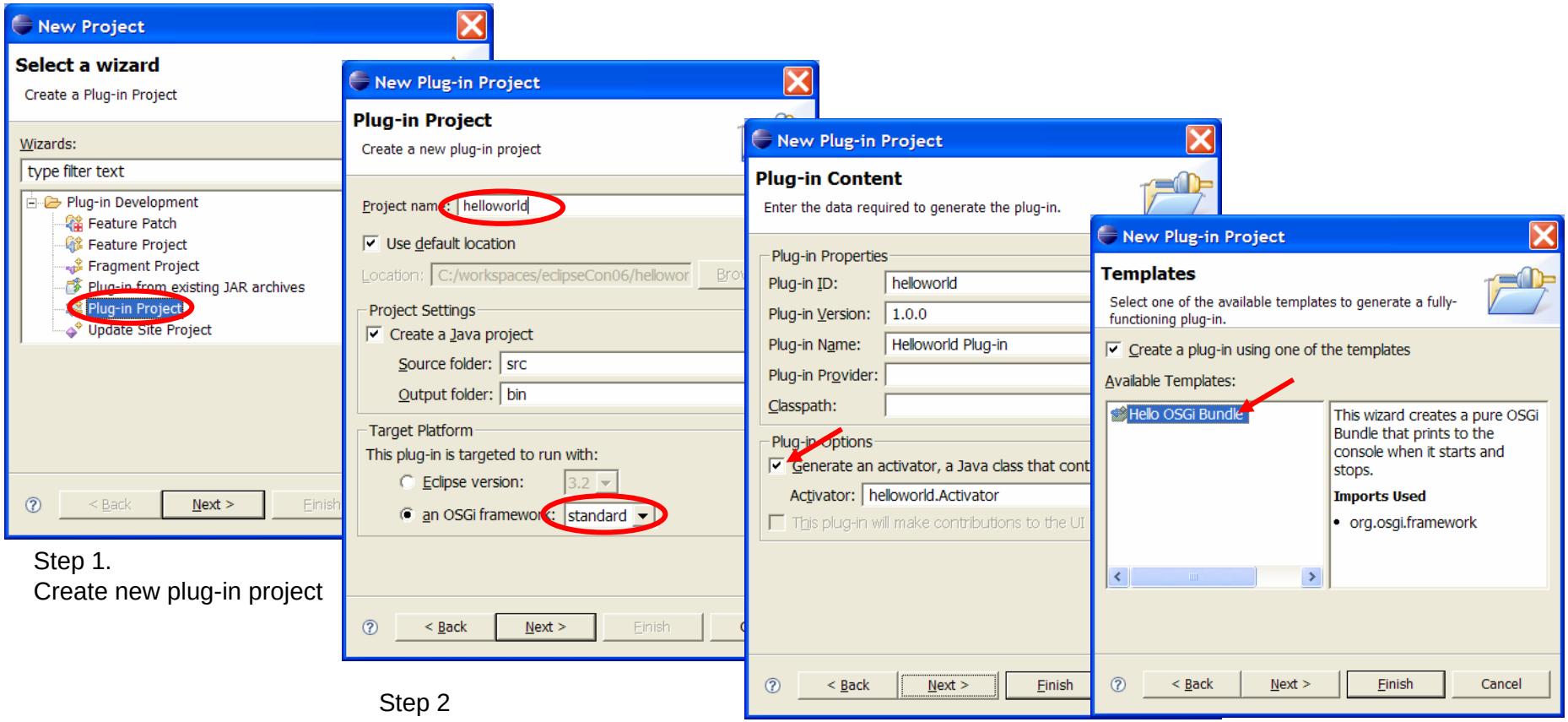
# What is in a Bundle?

- A Bundle contains (normally in a JAR file):
  - Manifest
  - Code
  - Resources
- The Framework:
  - Reads the bundle's manifest
  - Installs the code and resources
  - Resolves dependencies
- During Runtime:
  - Calls the Bundle Activator to start the bundle
  - Manages java classpath
  - Handles the service dependencies
  - Calls the Bundle Activator to stop the bundle



# Create the Hello World bundle




Step 1.
Create new plug-in project

Step 2
Project name: helloworld
an OSGi framework: standard

Step 3
Generate an activator

Step 4
Use the Hello OSGi Bundle template

# Real code! Hello World (and Goodbye)

- The wizard has generated the code on the left
- This class implements the BundleActivator so that the Framework can start/stop the class
- The activator is referenced in the manifest

HelloWorld.java

```
package helloworld
public class HelloWorld
   implements BundleActivator {
   public void start(
      BundleContext context)
      throws Exception{
      System.out.println(
           "Hello world!!");
   }

   public void stop(
      BundleContext context)
      throws Exception {
           System.out.println(
           "Goodbye world!!");
   }
}
```

# Real code! Hello World (and Goodbye)

- The Manifest (in META-INF/MANIFEST.MF) is also generated by the wizard
- Eclipse provides convenient editors for the manifest
  - For the source: click on MANIEST.MF
- Notice:
  - Bundle-Activator (used to notify the bundle of lifecycle changes)
  - Import-Package (dependencies)

META-INF/MANIFEST.MF

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Helloworld Plug-in
Bundle-SymbolicName: helloworld
Bundle-Version: 1.0.0
Bundle-Localization: plugin
Bundle-Activator:
helloworld.Activator
Import-Package:
 org.osgi.framework;version="1.3.0"
```

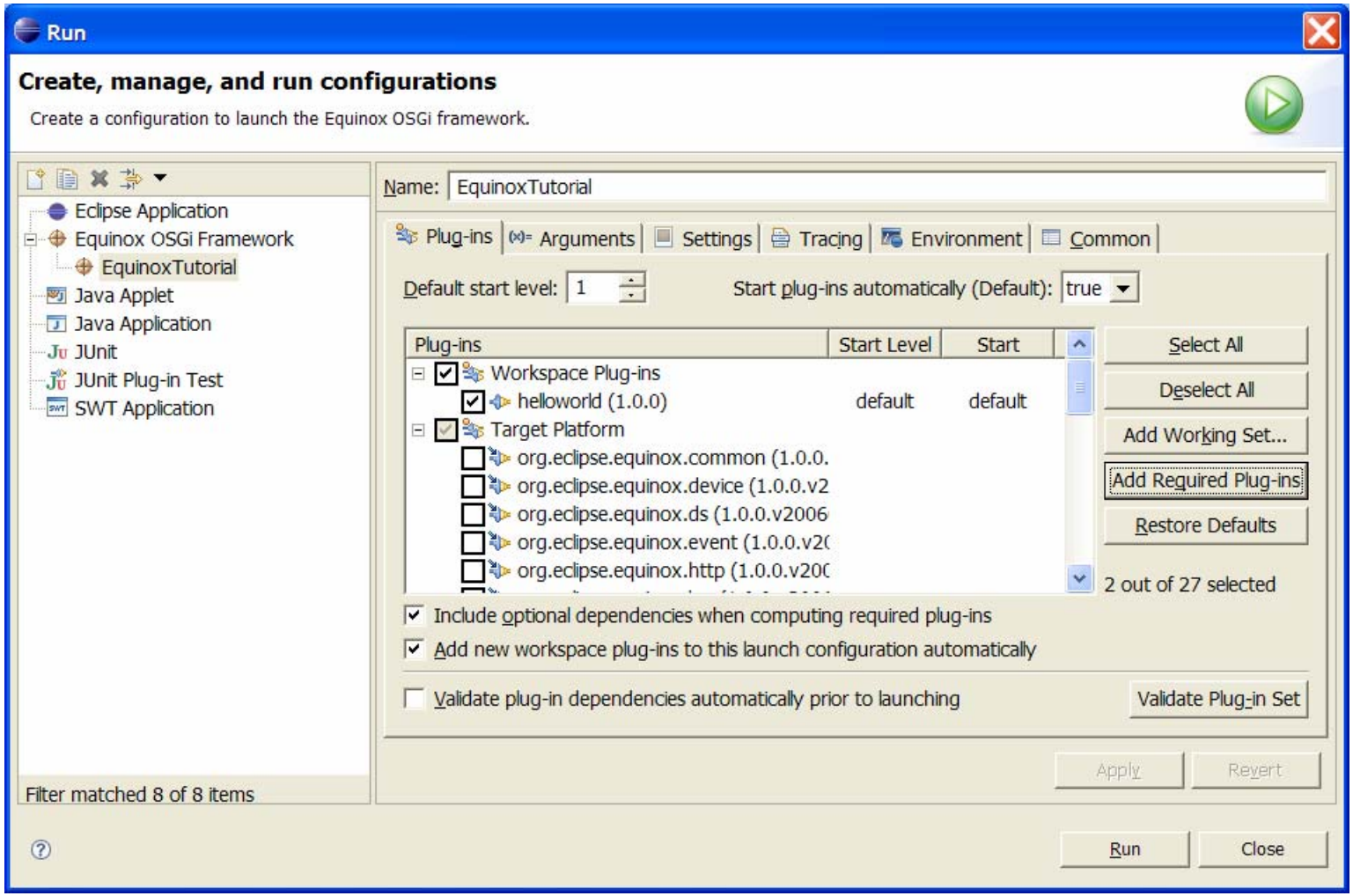
 

# Eclipse Launch Configuration

- The Launch Configuration is prepared for you
  - Run -> Run … -> EclipseTutorial
- Deselect “Workspace Plug-ins” and “Target Platform” checkbox
  - This removes all possible bundles from the launch configuration
- Select the helloworld bundle and Select “Add Required Plug-ins”
  - This calculates from the dependency information, which bundles are required to run our *helloworld* example

# Equinox Launch Configuration

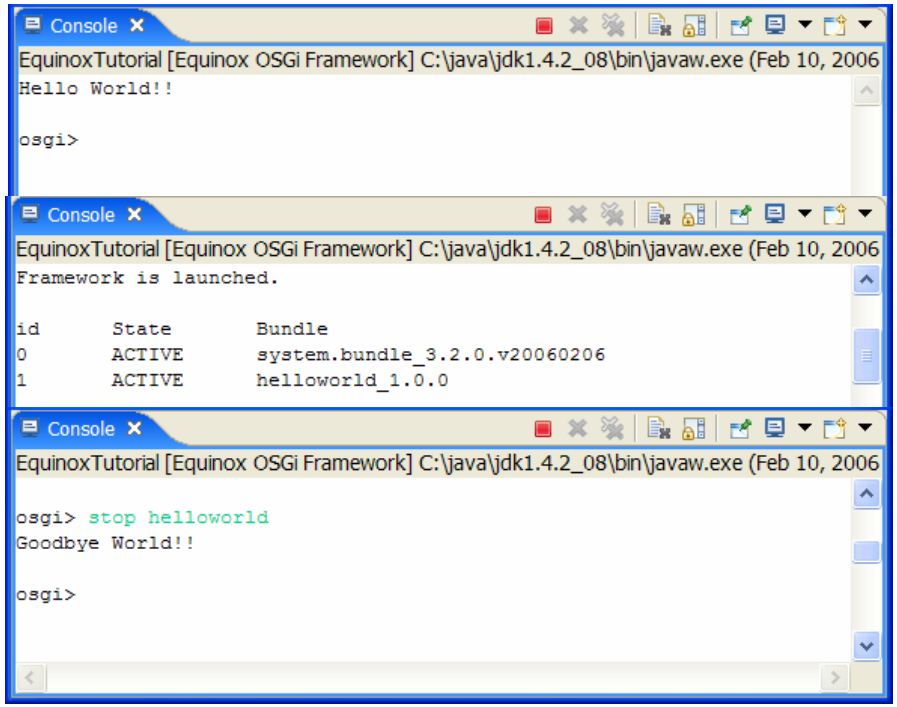




# Run the Hello World bundle

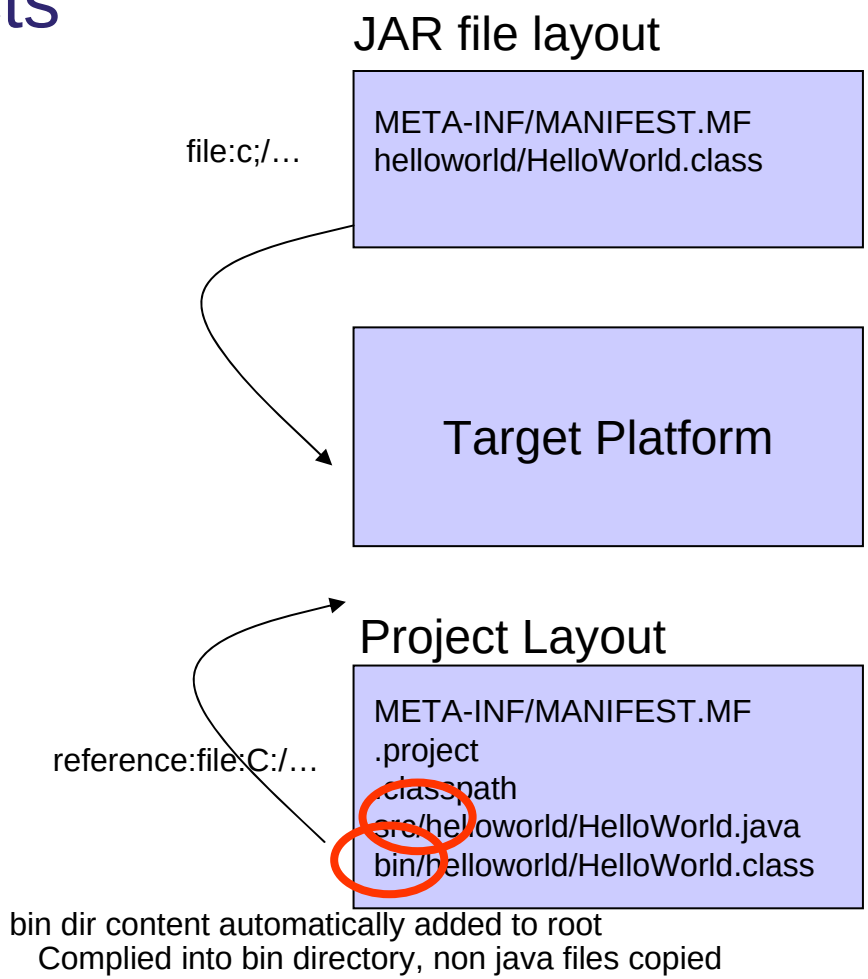
- Press Run
  - The Framework is a console application
- The Framework now runs the *helloworld* example
  - See the printed text
- It also runs a Framework console
  - Equinox specific
- Type “ss” (show status)
  - Look at the active bundles
  - Notice the number for the helloworld bundle. This is the bundle id.
- Type “stop <symbolic-name>”



# Self-Hosting Bundle Projects

- Normally, a bundle is packaged in a *JAR* file
  - The traditional edit-compile-debug cycle.
- Self-Hosting Allows for quick debugging of bundle code
  - No packaging steps
  - No deployment steps
  - Just code/save/run
- Some changes require update of the bundle in the Framework
  - Console:

    update <symbolic-name>



# Creating deployable bundles – how it works

- The build.properties file specifies the content of the bundle jar
  - Specifies the source and output folders of the different libraries
  - source.. – The source directory of the project. Used for compilation and resources.
  - output.. – The output directory where class files and resources are copied to
  - bin.includes – What is included in the JAR from the project directory

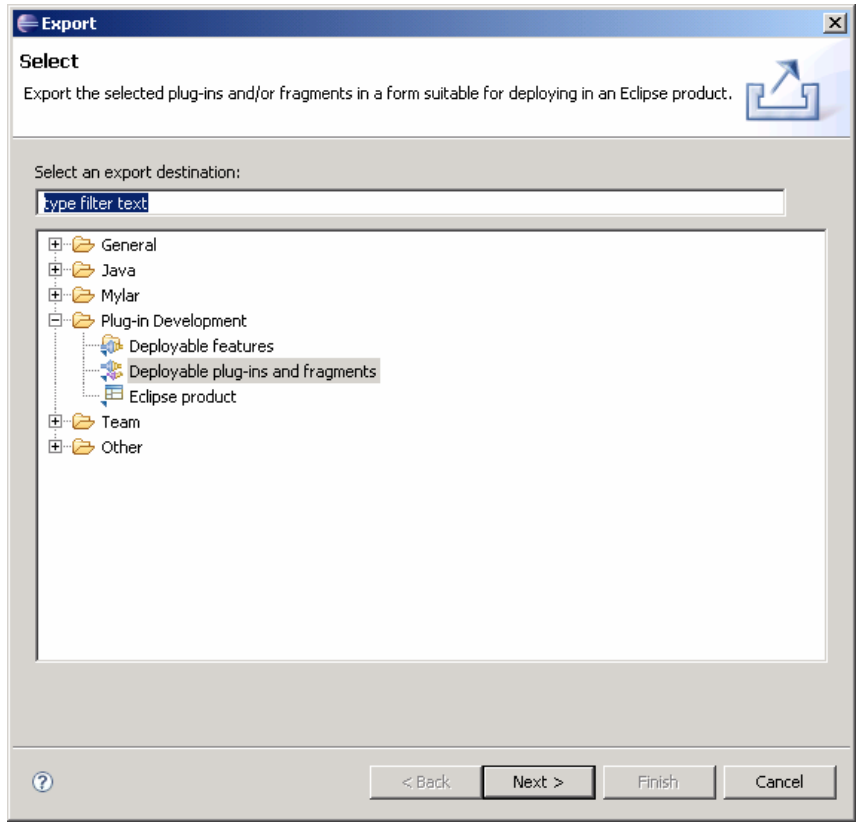
build.properties

```
source.. = src/
output.. = bin/
bin.includes = META-INF/,\
               .
```

# Export

- Export the content of a project into a bundle jar
  - Bundle jars can be installed across multiple OSGi Framework implementations
  - The *Deployable plug-ins and fragments* wizard can be used to generate a bundle jar from a project.
  - File -> Export -> Deployable plug-ins and fragments



# What Did We Learn

- The unit of deployment of an OSGi Service Platform
- The Eclipse Target Environment
- How to launch an Equinox environment with a defined set of bundles
- How to start/stop bundles
- How the Equinox console works
- How the classpath is managed for self hosted bundles

# Section IV – Component interaction and collaboration

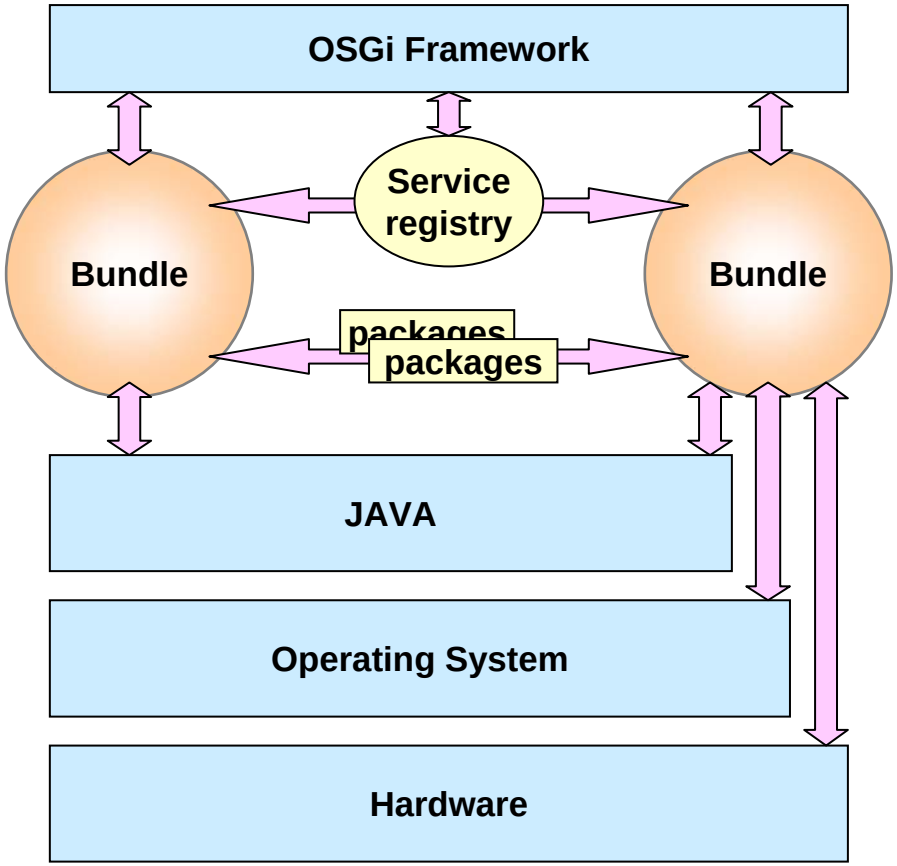
 

# Collaborative model

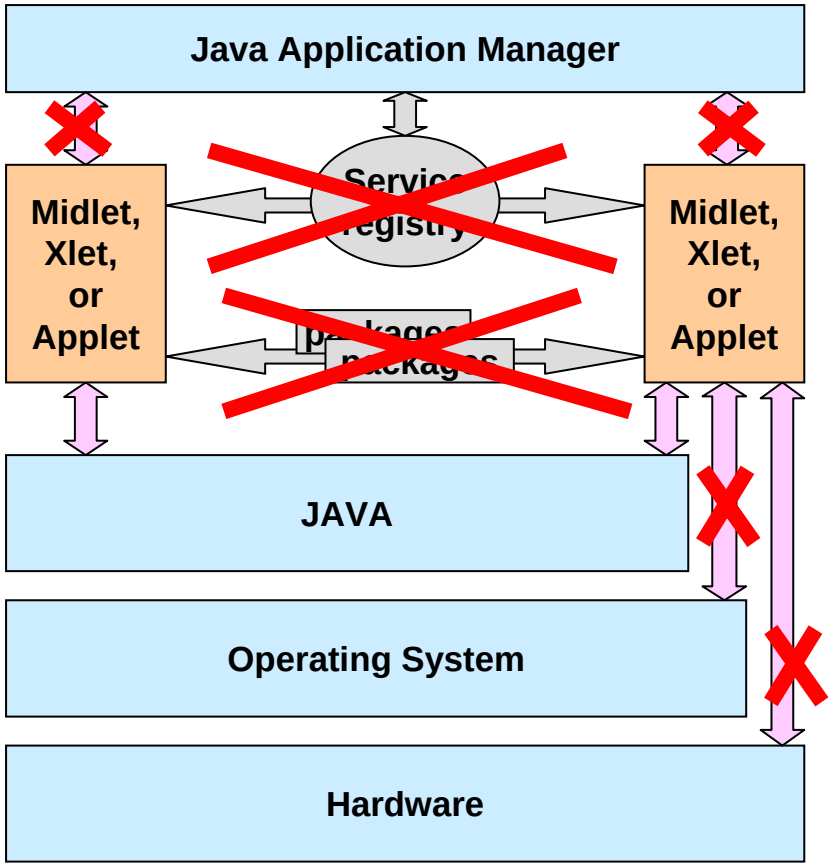
- *OSGi is more* than an Applet, MIDlet, Xlet runner
- Bundles can collaborate through:
  - *service objects*
  - *package sharing*
- A dynamic registry allows a bundle to find and track service objects
- Framework fully manages this collaboration
  - Dependencies, security

# Collaborative model

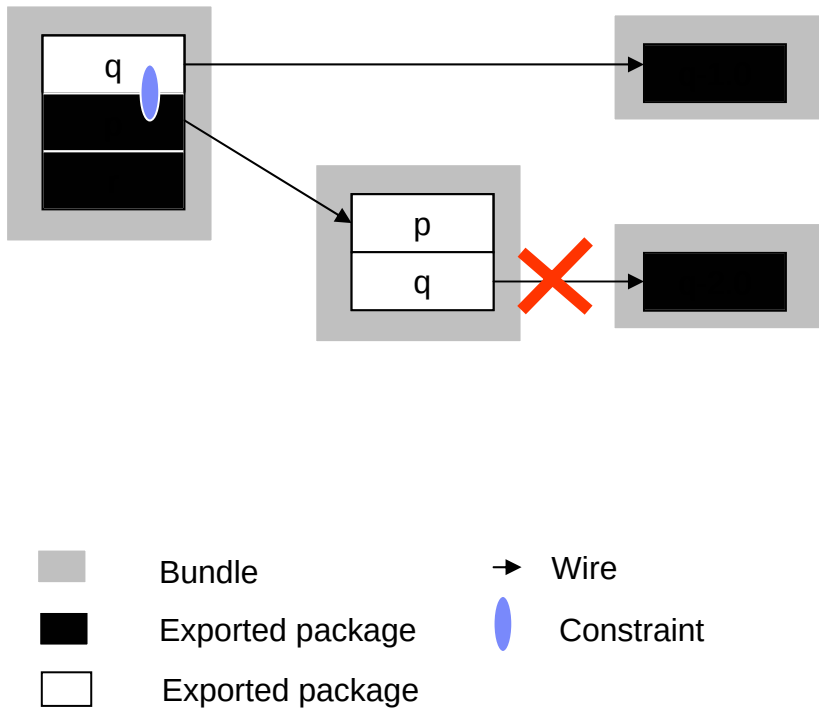


# Collaborative model



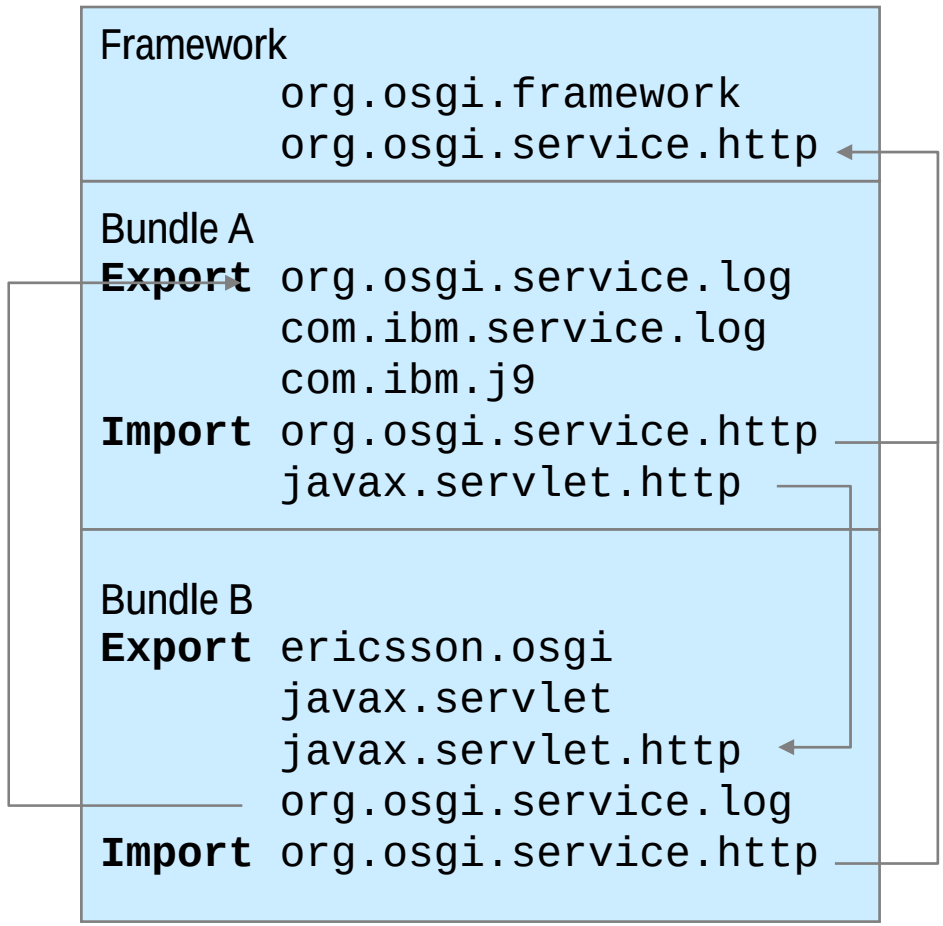
# Classpath issues

- Java applications consists of *classes* placed in *packages*
- Java searches for a package or class in different jar files and directories
  - These are usually specified in the CLASSPATH environment variable
- An OSGi Framework is a network of class loaders.
  - Parameterized by the Manifest headers
- Any dependencies between bundles are resolved by the Framework
- It is possible to fetch bundles on demand
- Complicated – But an OSGi Framework makes it painless to use



# OSGi dependency resolution



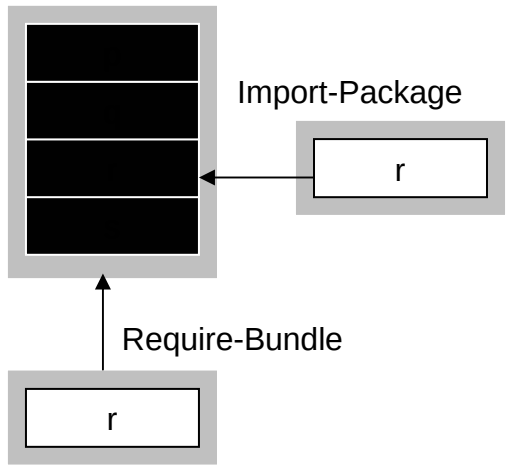

A resolved

B resolved

# Package or Bundle Dependencies?

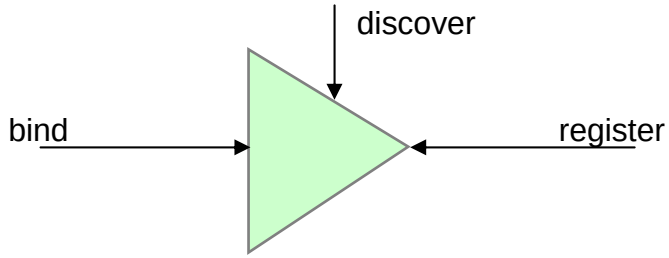
- The OSGi Specifications supports both Require-Bundle and Import-Package
- Require-Bundle creates a dependency on a complete bundle
  - Simple to use
  - Imports packages that are not used
- Import-Package creates a dependency on just a package
  - Creates less brittle bundles because of substitutability
  - More cumbersome to use (Tools!)
- In almost all cases, Import-Package is recommended because it eases deployment and version migration
- The specifications detail a number of additional problems with Require-Bundle



# Service Specifics

- A *service* is an object registered with the Framework by a bundle to be used by other bundles
- The semantics and syntax of a service are specified in a Java interface
- A bundle can register a service.
- A bundle can use a service (bind to)
  - 1..1
  - 0..1
  - 0..n
- A service can be discovered dynamically
- Services can go away at any time!




**service**

```
package org.osgi.service.log;
import org.osgi.framework.ServiceReference;
public interface LogService {
public static final int LOG_ERROR= 1;
public static final int LOG_WARNING= 2;
public static final int LOG_INFO= 3;
public static final int LOG_DEBUG= 4;
public void log(int level, String message);
public void log(int level,
  String message, Throwable exception);
public void log(ServiceReference sr,
  int level, String message);
public void log(ServiceReference sr,
  int level, String message,
  Throwable exception);
}
```

# Services continued

- The Framework Service Registry is available to all bundles to collaborate with other bundles
- Different bundles (from different vendors) can implement the same interface
  - Implementation is not visible to users
  - Allows operator to replace implementations without disrupting service
- OSGi defines a standard set of services
  - Other organizations can define more (AMI-C)
- Extensive notifications for service life cycles
- Services have a unique id
- Services require permission
  - Under Operator control
- Services are associated with properties
  - Query language to find appropriate service
  - Bundles can update the properties

# Manipulating Services

- The BundleContext provides the methods to manipulate the service registry
- Service registrations are handled by ServiceRegistration objects
  - They can be used to unregister a service or modify its properties
- Service References give access to the service as well as to the service's properties
- Access to service objects is through the getService method. These services should be returned with the ungetService method

```
ServiceRegistration registerService(
  String clazz,
  Object service,
  Dictionary properties)

ServiceReference[]
  getServiceReferences(
    String clazz,
    String filter)

Object getService(
  ServiceReference reference)

boolean ungetService(
  ServiceReference reference);
```

# What Did We Learn

- The OSGi Service Platform provides a collaboration model that is based on
  - Services
  - Package sharing
- Sharing is complicated, but the well defined specifications reduce the complexity for bundle developers
- Services provide a very powerful dynamic programming model

# Section V – Service Components

# Components Simplify Service Programming

- The dynamic nature of services make programming more complicated
- The declarative service model simplifies handling these dynamics
  - Dependencies are defined in an XML file
- Declarative Services:
  - Optionally Depend on one or more services
  - Optionally Provide a service
  - Optionally lazy initialized
  - Configurable
- Example shows a hello world bundle that logs Hello and Goodbye
- First add dependencies by selecting MANIFEST.MF, on the Dependencies tab
  - Add the component and log package

META-INF/MANIFEST.MF

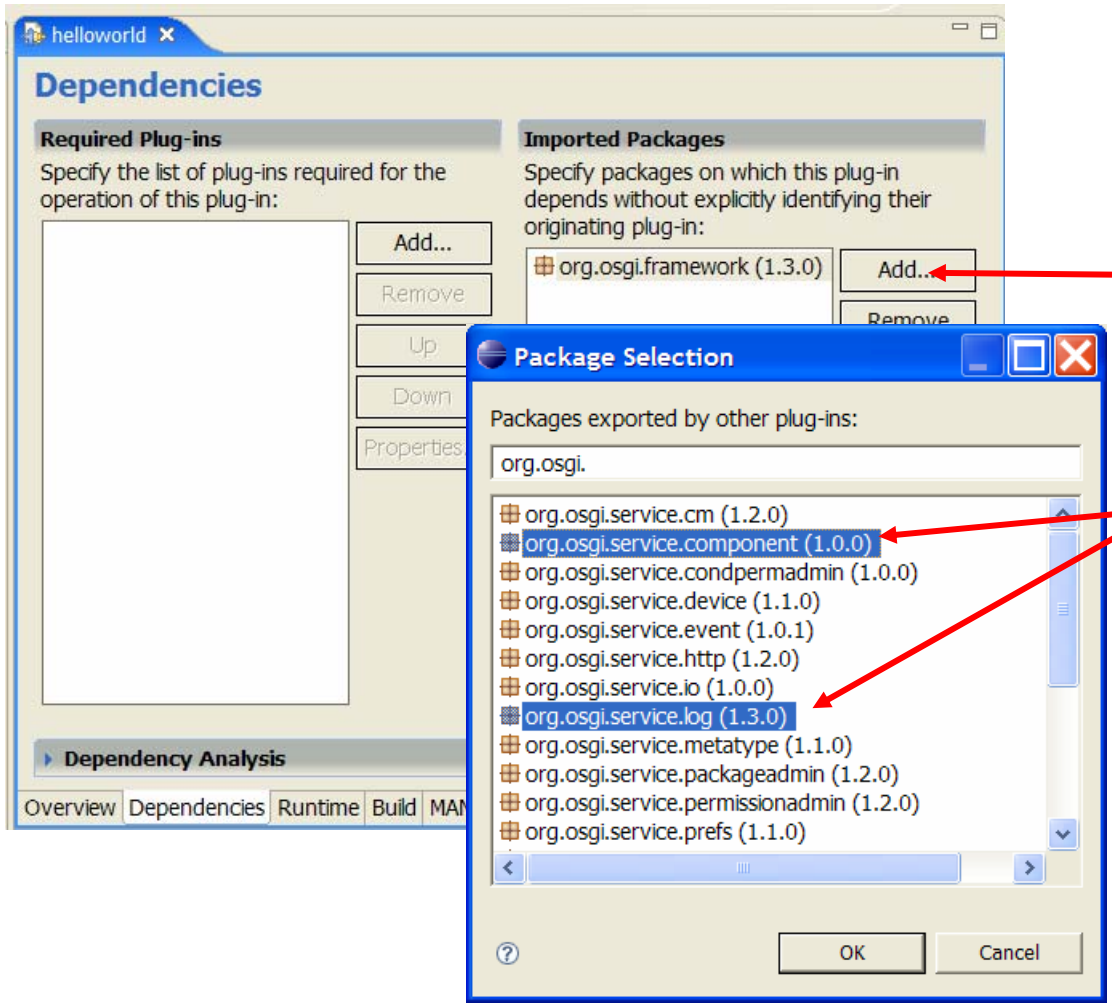
```
Manifest-Version: 1.0
…
Import-Package: org.osgi.framework;version,
 org.osgi.service.component,
 org.osgi.service.log
Service-Component:
 OSGI-INF/component.xml
```

OSGI-INF/component.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<component name="hello.component.log">
   <implementation
      class="hello.Component"/>
   <reference name="LOG" interface=
     "org.osgi.service.log.LogService"/>
</component>
```

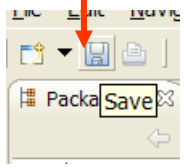
# Import the necessary packages




Step 1 – Add new Imported Packages

Step 2 – Select the necessary packages

Step 3 – Save the bundle manifest



# Login Component Source Code

- A component can be any class
  - No specific interface
- The activate and deactivate methods are called when the component is activated/deactivated
  - Dependencies must be resolved: Log Service
- The ComponentContext class provides access to the referenced services
  - The locateService methods finds a reference
- The component instance can be sure that at any moment in time between activate and deactivate there is a valid Log Service

OSGI-INF/component.xml

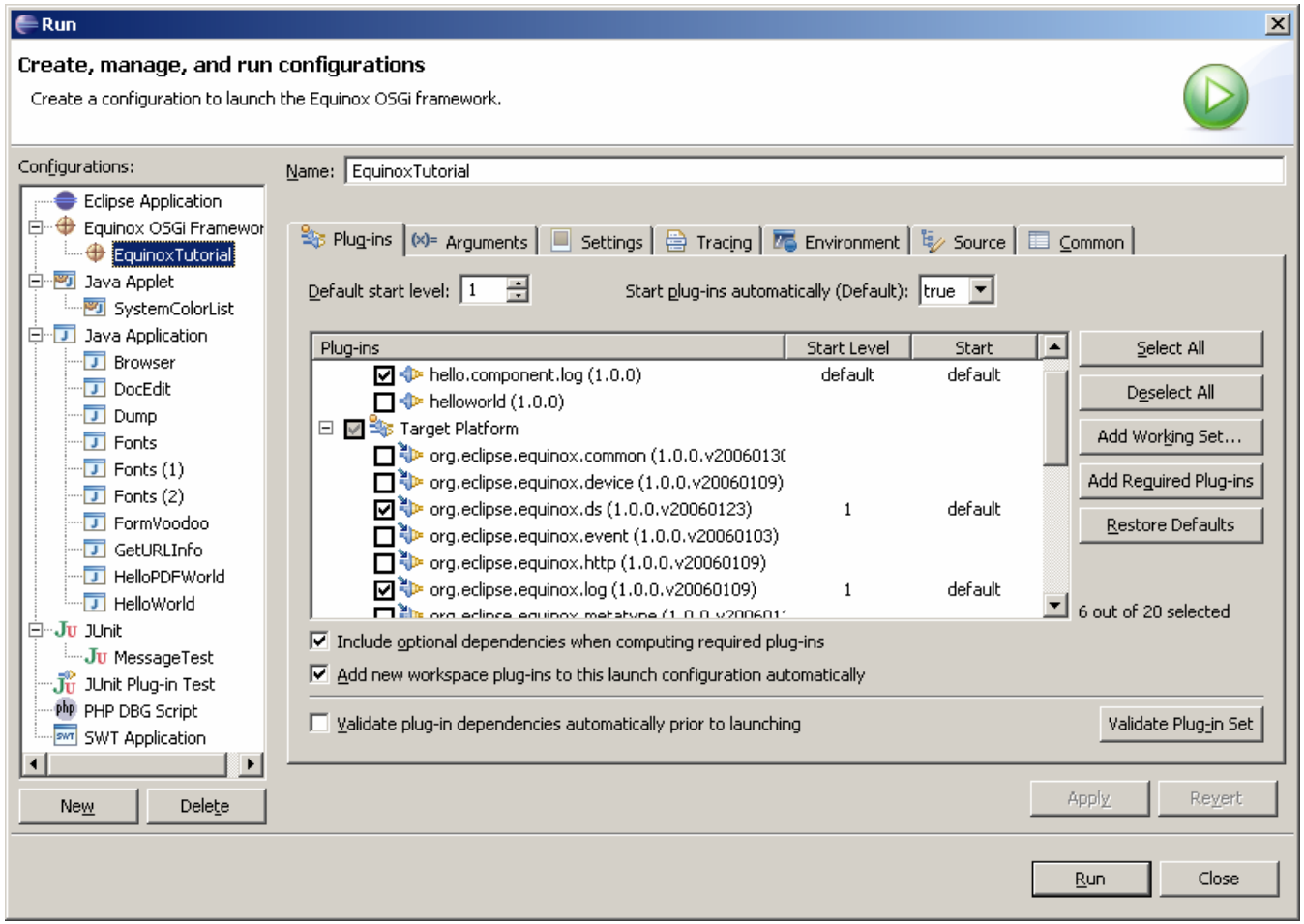
```
<?xml version="1.0" encoding="UTF-8"?>
<component name="hello.component.log">
  <implementation
    class="hello.Component"/>
  <reference name="LOG"
    interface="org.osgi.service.log.LogService"/>
</component>
```

Component.java

```
package hello;
import org.osgi.service.component.*;
import org.osgi.service.log.*;
public class Component  {
  LogService log;
  protected void activate(ComponentContext context){
    log = (LogService) context.locateService("LOG");
    log.log(LogService.LOG_INFO, "Hello World"); }
  protected void deactivate(ComponentContext context){
    log.log(LogService.LOG_INFO, "Goodbye World");
  }}
```

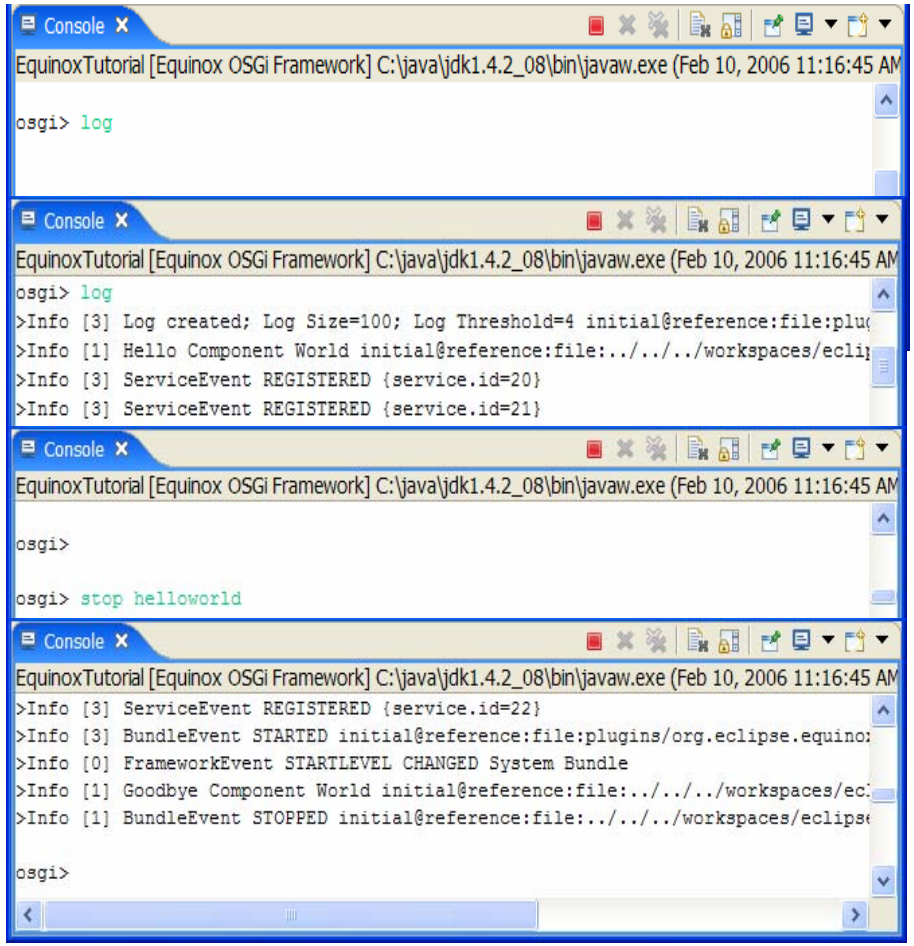
 

# Add Declarative Services, Log, and Component



# Launching

- Launch the EquinoxTutorial launch configuration
- You can look in the log with the log command
  - Last event is at bottom
- Stop the bundle
  - Stop <symbolic-name>
- Run log again



# What Did We Learn

- Programming with services is complicated
- The Declarative Services model makes service programming much simpler
- How the component XML is constructed
- We used the Log Service

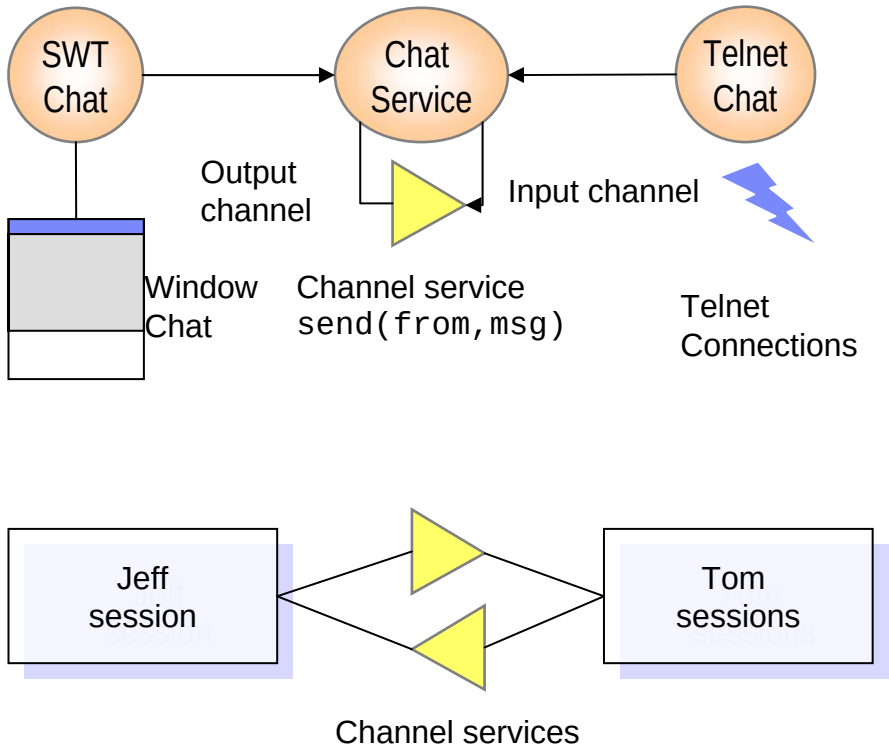
 

# Section VI – Use Case: Developing a Chat Service

# A Chat Service

- We will now design a service that simplifies Chat/Instant Messaging clients
  - We do the clients later, this is just intended to support clients.
- A Chat client should be able to communicate with a user through:
  - A window, telnet session, MSN, AOL, Skype, etc. interface.
- We base the communication between chat clients on a *Channel* interface.
  - We register a service we receive messages from
  - The registry contains other channel services we can send messages to
  - A property contains the user name
- For ease of use, we use a command based interface for login and listing buddies



# Channel Service Design

- Create a new project to hold our service interface
  - Call this project <myname>.chat
  - This is a Plug-in Project
- The Channel service is one way:
  - Each channel service represents on user
  - We only receive through a channel service
  - A client uses a channel service to send a message to a specific user
- The CH_NAME service property
  - This property must be registered with the service
  - The value is the name of the user, e.g. tom
- A single method *send* with the following arguments
  - from – The user name that sends the message
  - msg – The message to be send
- Export the service channel package

Channel.java

```
package aQute.service.channel;
import java.io.*;
public interface Channel {
  String CH_NAME="channel.name";

  void send(String from, String msg)
      throws IOException;
}
```

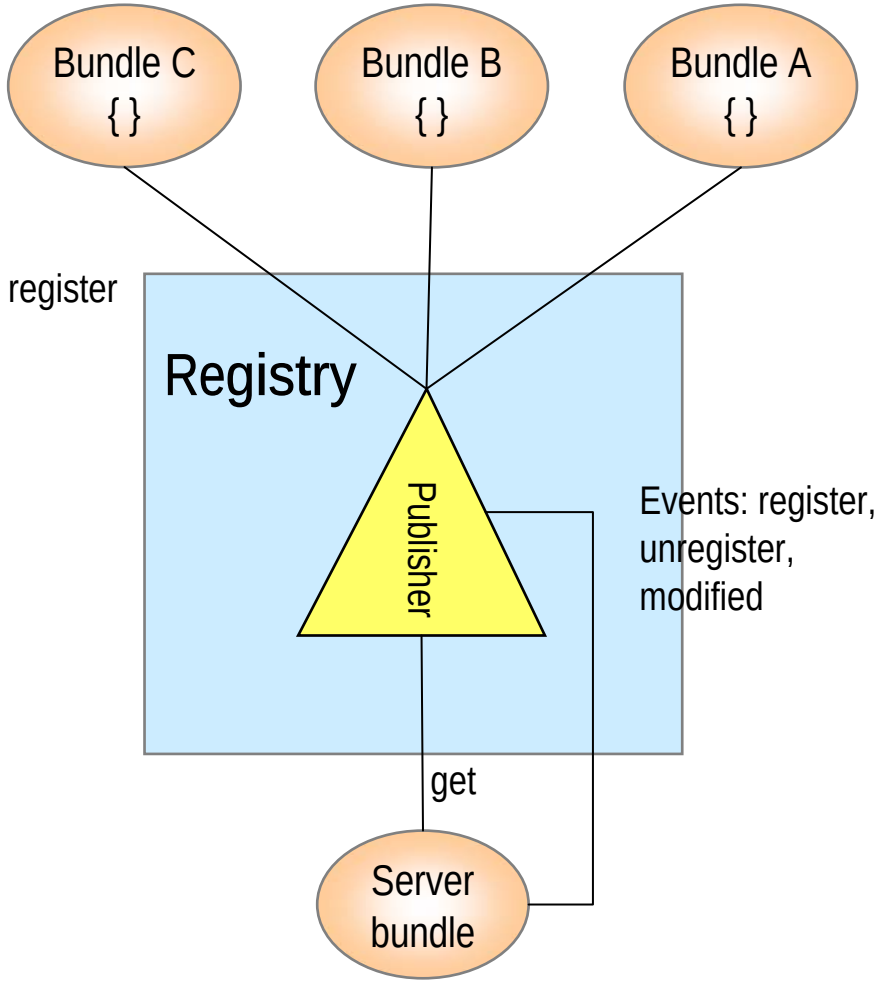
META-INF/MANIFEST.MF

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Chat Plug-in
Bundle-SymbolicName: aQute.chat
Bundle-Version: 1.0.0
Bundle-Localization: plugin
Import-Package:
 org.osgi.framework;version="1.3.0"
Export-Package: aQute.service.channel
```

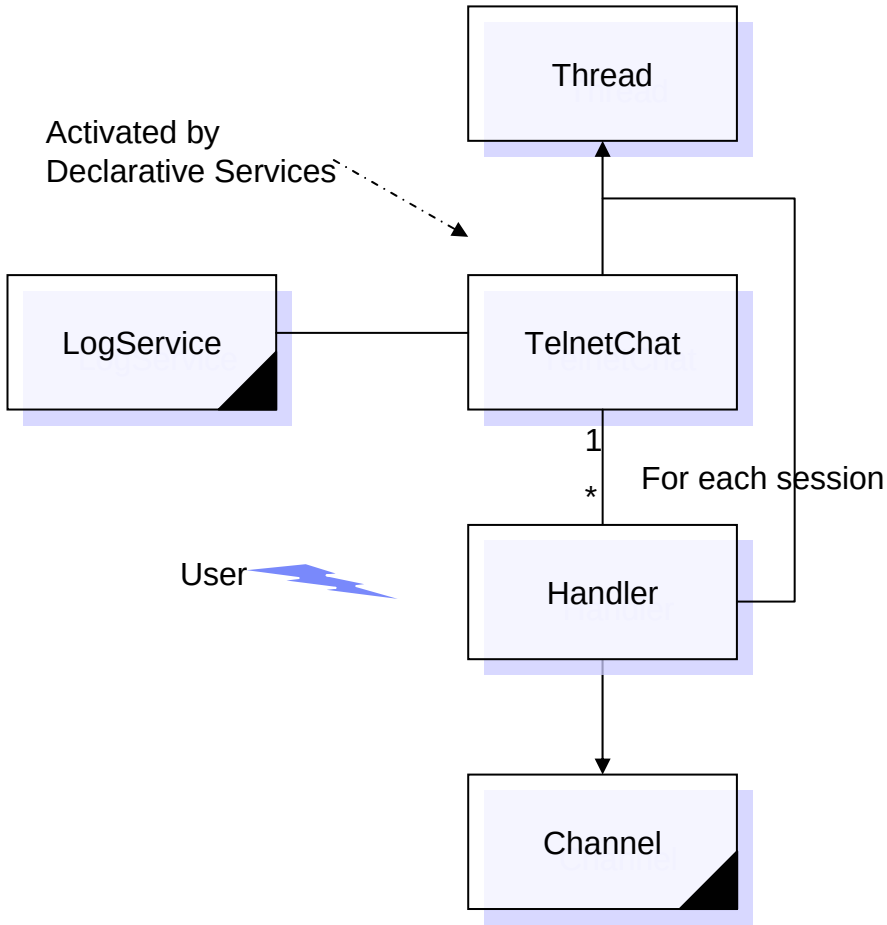
# White Board Approach

- The Channel Service uses the White Board Approach
- The White Board approach is:
  - Each Event Listeners (Channels) are registered as Services
  - Any interested client uses the service to send events (messages) to
- It is an effective approach to reduce the number of couplings between bundles
- There is a white-paper comparing a whiteboard approach with a non whiteboard approach.



# Telnet Based Chat Client

- The best way to start is to design a small test program.
- The easiest way to a “UI” is a telnet server that uses the Channel service to communicate with siblings
  - This also shows how Internet servers should be constructed
- The telnet Chat server will create a Handler for each opened session.
  - The Handler is a thread that waits for input from the user
  - The Handler registers a Channel.
- The Handler is stopped when the session closes.
  - This unregisters the Channel service



# The TelnetChat Manifest and component.xml

- Create a new project for a telnet chat
  - Call this project <myname>.telnetchat
  - This is a Plug-in Project
- Define the manifest and component definition
- Add the package import dependencies to the manifest. Either
  - Direct in the source code
  - Via the Dependencies tab
- Add the reference to the component.xml
- The component.xml must reside in OSGI-INF
- We only specify a reference to the Log Service

META-INF/MANIFEST.MF

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Telnetchat Plug-in
Bundle-SymbolicName: aQute.telnetchat
Bundle-Version: 1.0.0
Bundle-Localization: plugin
Service-Component: OSGI-INF/component.xml
Import-Package: aQute.chat,
 aQute.service.channel,
 org.osgi.framework;version="1.3.0",
 org.osgi.service.component,
 org.osgi.service.log
```

OSGI-INF/component.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<component name="aQute.telnetchat">
 <implementation class=
   "aqute.telnetchat.TelnetChat"/>
 <reference name="LOG" interface=
  "org.osgi.service.log.LogService"/>
</component>
```

# The TelnetChat Component code

- The code does not show the import packages and field definitions
  - The source code is provided for you to further check in aQute.telnetchat
  - Eclipse will tell you when they miss
- The activate method:
  - Remembers the context for later
  - Gets the log service
  - Starts the thread.
- The deactivate method:
  - Sets the quit flag so any loops in the started Thread will finally end
  - Closes all created Handlers
  - Exceptions are ignored because we are closing
  - And closes the server socket object, this will surely quit the main socket accept loop.

TelnetChat.java

```
protected void activate( ComponentContext context) {
  this.context = context;
  this.log = (LogService)
    context.locateService("LOG");
  start();
}

protected void deactivate(ComponentContext context)
    throws Exception {
  quit = true;
  for (Iterator i = handlers.iterator(); i.hasNext();)
    try {
      Handler h = (Handler)i.next();
      h.close();
    } catch (Throwable e) {
       // We are closing
    }
  server.close();
}
```

# The TelnetChat run method

- The run method creates a socket and accepts incoming connections.
- For bundles, it is crucial that this loop never quits, but also not overloads the system
  - There is usually no end user watching a server …
- The outer loop therefore catches errors, sleeps and try again
  - A lot of problems disappear over time. For example, an Internet connection can be temporarily be down
- The socket has a timeout to check the flag regularly
- The inner loop
  - Wait for an incoming socket
  - Creates a handler
  - And starts the handler's thread

TelnetChat.java

```
public void run() {
  while (!quit) try {
    server = new ServerSocket(2030);
    server.setSoTimeout(1000);
    loop();
  } catch (Exception e) {
    log.log(LogService.LOG_ERROR,
       "[TelnetChat] Inner loop", e);
    sleep(10000);
  }
}

void loop() throws Exception {
  while (!quit) try {
    Socket socket = server.accept();
    Handler handler = new Handler(
        context.getBundleContext(), socket, this);
    handlers.add(handler);
    handler.start();
  } catch (SocketTimeoutException e) {
    // Just for checking the quit flag
    // at a regular basis
  }
}
```

# TelnetChat convenience methods

- Convenience methods

TelnetChat.java

```
void remove(Handler handler) {
  handlers.remove(handler);
}

void sleep(int ms) {
  try {
     Thread.sleep(ms);
  } catch (InterruptedException e1) {}
}
```

# The Handler Source Code

- The constructor receives the socket for the session with the end user. It initializes:
  - The fields
  - A Writer object to send text to the end user
- The send method writes the message in the Write object and flushes it to ensure the user sees it
- The close method closes the different objects and quits the main loop:
  - By setting the quit flag
  - By closing the socket

Handler.java

```
public Handler(BundleContext context, Socket
    socket, TelnetChat activator) throws Exception {
  this.ctxt = context;
  this.socket = socket;
  this.parent = activator;
  writer = new PrintWriter(
    new OutputStreamWriter(socket.getOutputStream()));
}
public void send(String source,
    String msg) throws IOException {
  writer.println(source + "> " + msg);
  writer.flush();
}
void close() {
  try {
    quit = true;
    writer.close();
    socket.close();
  } catch (IOException e) {
    // Ignore in close
  }
}
```

# The Handler Source Code

- The run() method loops as long as there is input from the user. It quits when the socket is closed or an error occurs.
- Errors are only logged when the session has not quit because there are usually socket errors during closing
- The finally clause is used to guarantee that the handler is removed from the parent when it is closed.
- If a valid line is received, it is send to the process method

Handler.java

```
public void run() {
  writer.println("Welcome … Chat");
  writer.print("Enter name: ");
  writer.flush();
  try {
    BufferedReader rdr =
      new BufferedReader(
        new InputStreamReader(
          socket.getInputStream()));
    while (!quit && (line=rdr.readLine()) != null) {
      line = line.trim();
      process(line);
    }
  } catch (Exception e) {
    if (!quit)
      parent.log.log(
        LogService.LOG_ERROR,
        "reading user input", e);
  } finally {
    if (channel != null)
      channel.unregister();
    parent.remove(this);
    parent = null;
    close();
  }
}
```

# The Handler Source Code

- The process method handles a line of input from the user
- If we did not have a login name yet, we assume it is the given line
- If the line starts with /quit, we quit the program
- Otherwise we assume it is a line we need to send to another user, which is handled in the dispatch method

Handler.java

```
void process(String line) throws IOException,
    Exception {
  if (user == null) {
    user = line;
    Hashtable props = new Hashtable();
    props.put(Channel.CH_NAME, user);
    channel =ctxt.registerService(
        Channel.class.getName(), this, props);
    send("info", "User set to: " + user);
  } else {
    if (line.startsWith("/quit"))
      writer.println("bye ");
    else
      dispatch(line);
  }
}
```

# The Handler Source Code

- The dispatch method parses the destination name from the input
- For this name , finds a Channel service
- It then sends the remainder of the line to that service

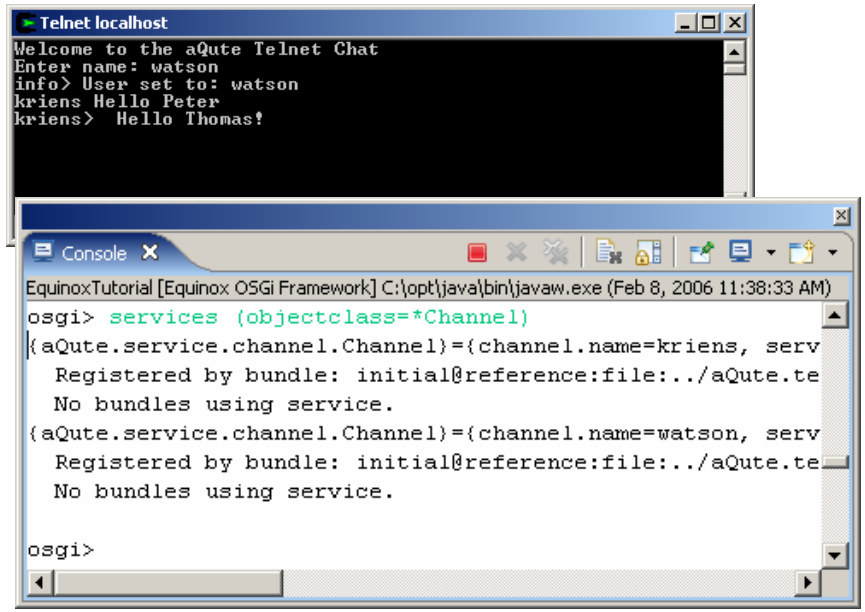
Handler.java

```
void dispatch(String line) throws Exception {
  String parts[] = line.split("\\W");
  ServiceReference channels[] =
    ctxt.getServiceReferences(
      Channel.class.getName(),
      "(" + Channel.CH_NAME + "=" + parts[0] + ")");
  if (channels != null) {
    Channel to = (Channel)ctxt.getService(channels[0]);
    to.send(user, line.substring(parts[0].length()));
    ctxt.ungetService(channels[0]);
  } else {
    send("error", "no such user: " + parts[0]);
  }
}
```

# Run the Telnet Chat

- Launch the EquinoxTutorial
- Do not forget to check the launch configuration
  - The TelnetChat bundle included?
  - Includes all required bundles from the Target environment?
  - Do not forget to start the bundle via the console
- Create 2 telnet sessions:
  - Open a telnet session into port 2030
  - Login with your last name
  - Open a second telnet session into port 2030
  - Login with another name
- See if you can send messages Check the services, and see that two channel services are registered
  - Services (objectclass=*Channel)

# What Did We Learn

- How services are designed
  - White board approach
- Developed a simple telnet chat application
  - Chat sessions use the white board approach to find Channel services
  - The Channel service is used to send messages

# Section VII – Service Tracking

# Finishing the Chat Service

- The TelnetChat contains code that must be repeated between different clients
- A Chat library that captures the shared code would be useful
  - As a bundle, this could run on phones, Eclipse, etc.
- For this example, we create a Chat class that works on a command line basis
  - The Chat class will be added to the <myname>.chat bundle
  - /xxx are commands
- This bundle will therefore act as a library and exports the chat package
- Not all code is shown, however, this is available in the aQute.chat project

Chat.java

```
Chat(BundleContext cntxt,Channel user);

void execute(String ln) throws IOException;

String[] getBuddies();

void close();

String getName();

void login( String user, String passwd);
```

META-INF/MANIFEST.MF

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Chat Plug-in
Bundle-SymbolicName: aQute.chat
Bundle-Version: 1.0.0
Bundle-Localization: plugin
Import-Package:
 org.osgi.framework;version="1.3.0"
Export-Package: aQute.service.channel,
 aQute.chat
```

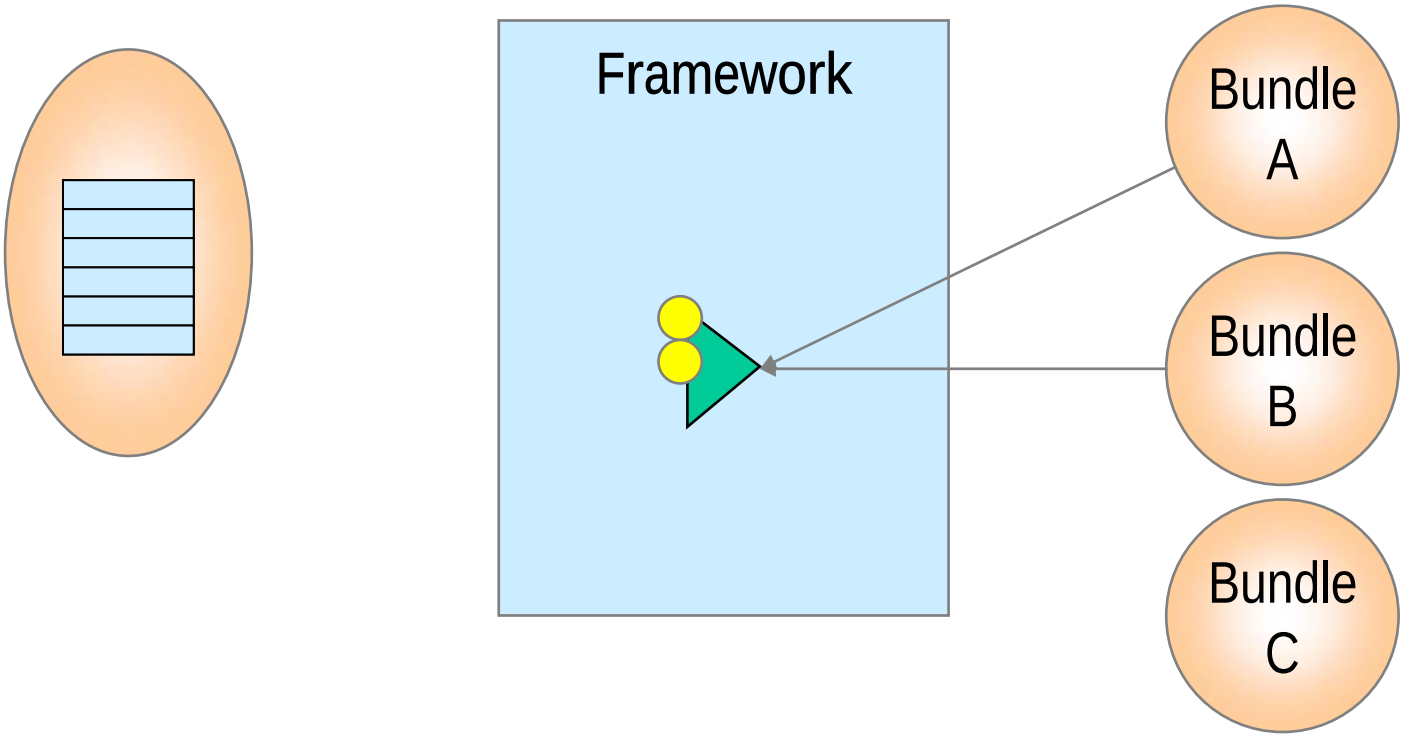
 

# The Case for the ServiceTracker

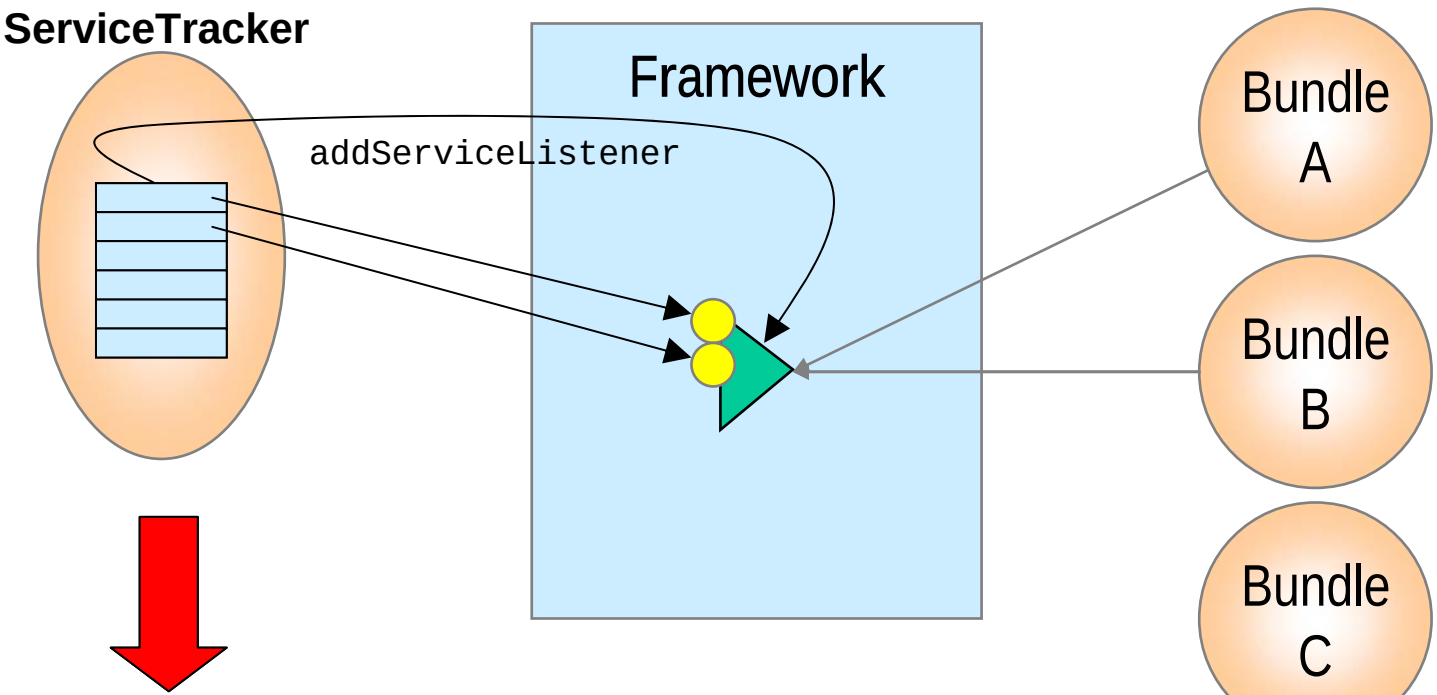
- Finding services for each message is kind of expensive.
- The ServiceTracker in org.osgi.util.tracker package is intended to simplify this task
- A service tracker maintains a list of services based on:
  - A filter
  - A specific class
- It reports any existing or new services as well as any services that become unregistered
  - Object addingService
  - void modifiedService
  - void removedService
- The service tracker is used to track channels and store them in a Map

# ServiceTracker: create



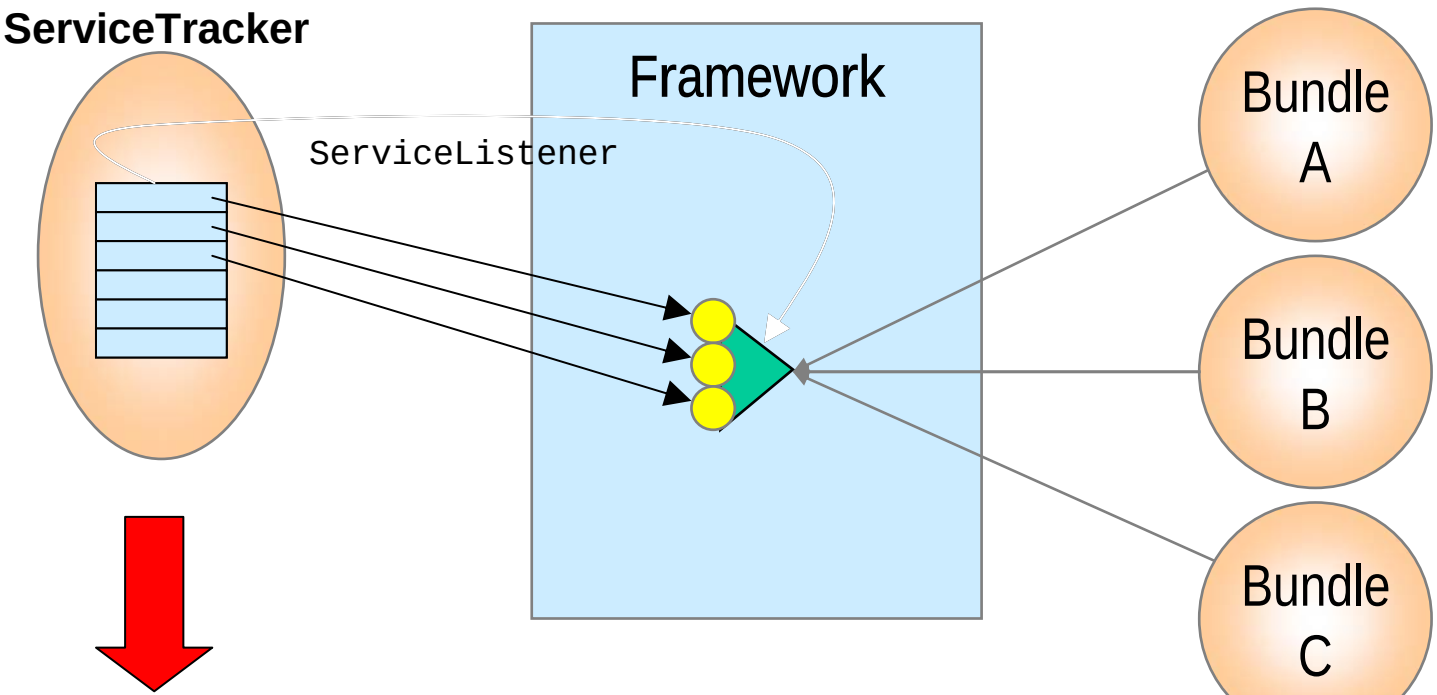
# ServiceTracker: open




```
Object addingService(ServiceReference r){…}
```

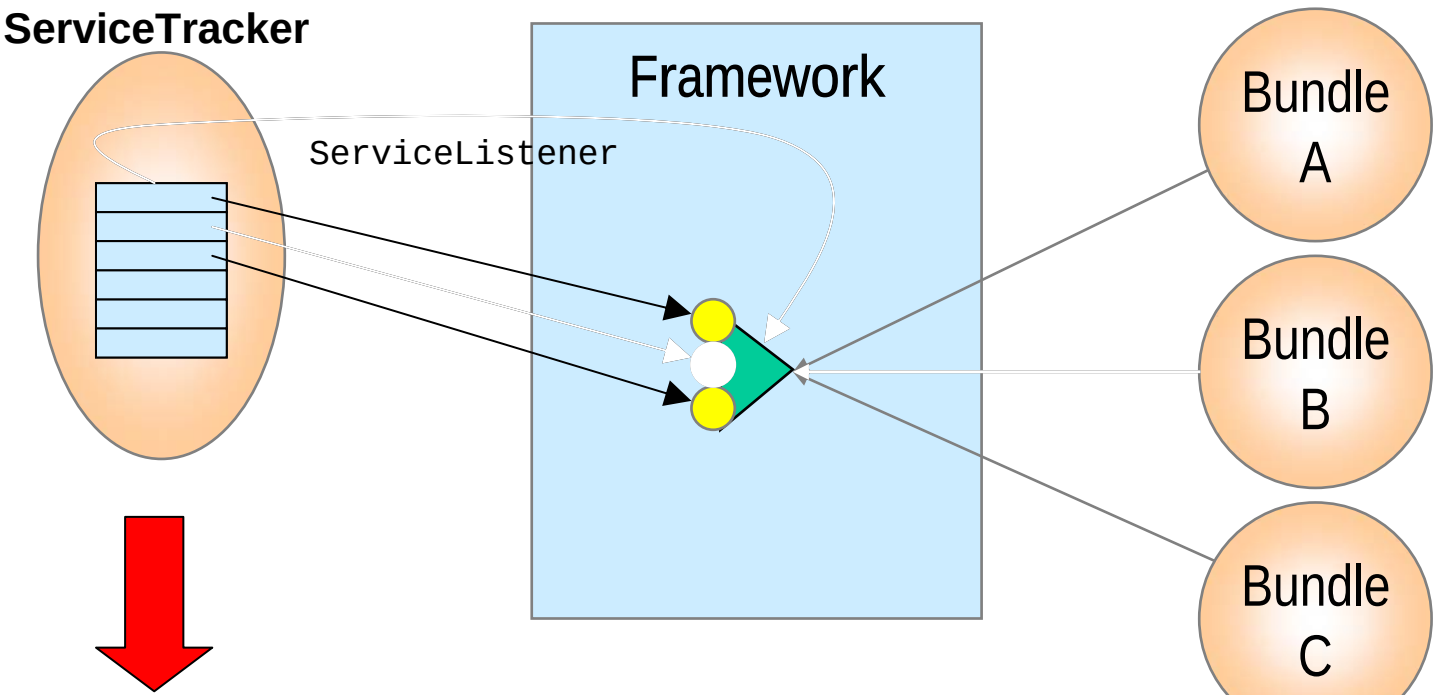
# ServiceTracker: adding




```
Object addingService(ServiceReference r){…}
```

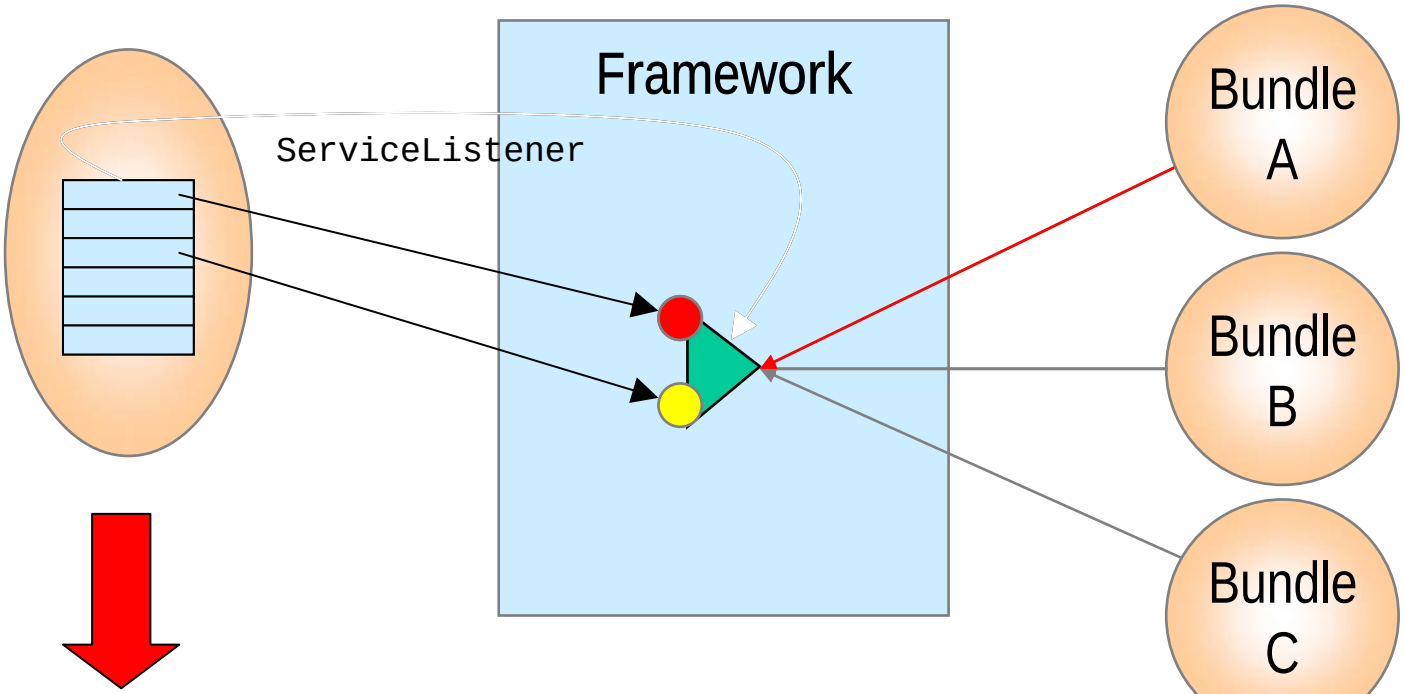
 

# ServiceTracker: removing




```
void removedService(ServiceReference r,Objecto){…}
```

# ServiceTracker: modified




```
void modifiedService(ServiceReference r, Object o){…}
```

# The Chat Library

- The Constructor initializes some fields and creates the service tracker
- The user is given as a Channel. The user of this library must implement this service to get a callback with any incoming messages.

Chat.java

```
public Chat(BundleContext context, final
    Channel user) {
  this.user = user;
  this.cntxt = context;
  channels = doChannelTracker(context, user);
  channels.open();
}
```

# The Chat Library Service Tracker

- The Service Tracker is used to track Channel services
  - The addingService method gets the Channel and puts the Channel in a Map under the given name
  - The removedService method just cleans up the Map when a Channel service is removed

Chat.java

```
ServiceTracker doChannelTracker(
    BundleContext cntxt, final Channel user) {
  return new ServiceTracker(
    cntxt,
    Channel.class.getName(), null) {
      public Object addingService(ServiceReference ref) {
        Channel buddy = (Channel)context.getService(ref);
        if (buddy != user) {
          String name =
              (String)ref.getProperty(Channel.CH_NAME);
          String rn = name;
          int n = 0;
          synchronized (bdds) {
            while(bdds.containsKey(rn))
              rn = name + "-" + n++;
            bdds.put(rn, buddy);
          }
        }
        return buddy;
      }

      public void removedService(
          ServiceReference ref,Object buddy){
        bdds.remove(ref.getProperty(Channel.CH_NAME));
      }
  });
}
```

# The Chat Library

- The login method registers a new Channel service
- The password is ignored
- The result is sent as a message to the user

Chat.java

```
public void login(String name,
  String password) throws IOException {
  if (registration != null)
    registration.unregister();
  registration = null;
  this.name = name;
  Hashtable properties = new Hashtable();
  properties.put(
    Constants.SERVICE_PID,
    "pid:chat[" + InetAddress.getLocalHost()
    + "]:" + name);
  properties.put(Channel.CH_NAME, name);
  registration = cntxt.registerService(
    Channel.class.getName(), user, properties);
  user.send("", "Logged in as " + name);
}
```

# The Chat Library: execute

- The execute method looks at the command line and scans for '/' characters, which are commands
  - /help – Show short help
  - /buddies – List the buddies
  - /login – Login
  - /<buddy> to send to a buddy
- If no / is given the message is sent to the last used buddy

Chat.java

```
public void execute(String line) throws IOException {
  if (!line.startsWith("/"))
    send(line);
  else {
    String ws[]=line.split("\\s+");
    if ("/buddies".startsWith(ws[0]))
      doBuddies();
    else if("/help".startsWith(ws[0]))
      doHelp();
    else if("/login".startsWith(ws[0]))
      login(ws[1], null);
    else {
      lastTo = ws[0].substring(1);
      send(line.substring(ws[0].length() + 1));
    }
  }
}
```

# The Chat Library: send

- The send method must transfer the message to the lastTo buddy.
- We maintain all the buddy Channel services in the bdds Map field, so it is easy to find them

Chat.java

```
void send(String line) throws IOException {
  if (lastTo == null)
    user.send("", "No buddy");
  else {
    Channel channel = (Channel) bdds.get(lastTo);
    if (channel != null) {
      channel.send(name, line);
      user.send(name, line);
    } else
      user.send("?", "Can't find " + lastTo);
  }
}
```

# The Chat Library: doBuddies, getBuddies

- The doBuddies method sends the list of currently logged in buddies to the user
- The getBuddies method returns the buddies as a String[]

Chat.java

```
void doBuddies() throws IOException {
  StringBuffer sb = new StringBuffer();
  String del = "";
  for(Iterator i =
    bdds.keySet().iterator();
    i.hasNext();) {
      sb.append(del);
      sb.append(i.next());
      del = ", ";
  }
  user.send("", sb.toString());
}

public synchronized String[] getBuddies() {
  return (String[])
    bdds.keySet().toArray(new String[0]);
}
```

# The Chat Library: utiltities

- The doHelp method sends some help text to the user
- The getName method returns the currently logged name
- The close method uses a careful way to unregister the associated Channel service only once
  - Often it is not easy to control how many time close is called

Chat.java

```
void doHelp() throws IOException {
  user.send("", "…n as: " + name);
  user.send("", "/bdds …");
  user.send("", "/help …");
  user.send("", "/login …");
  user.send("", "/<name> … ");
  user.send("", "…");
}

public String getName() {
  return name;
}

public void close() {
  ServiceRegistration reg;
  synchronized (this) {
    reg = registration;
    if (reg == null)
      return;
    registration = null;
  }
  reg.unregister();
}
```

# What Did We Learn

- How to track services and react appropriately on their arrival and departure
- How to use the Service Tracker
- The white board pattern as a solution to many dynamic problems

# Section VIII – Finishing Touch

# Using the Chat Library

- We now have a library bundle that is easy to use
- We could adapt the Telnet Chat, but that is old news
- Lets make a small SWT program that shows a simple chat window
  - Call this project <myname>.swtchat
  - This is a Plug-in Project
- Such a program is provided in the aQute.swtchat bundle

META-INF/MANIFEST.MF

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Swtchat Plug-in
Bundle-SymbolicName: aQute.swtchat
Bundle-Version: 1.0.0
Service-Component:
 OSGI-INF/component.xml
Bundle-Localization: plugin
Import-Package: aQute.chat,
 aQute.service.channel,
 org.eclipse.swt,
 org.eclipse.swt.events,
 org.eclipse.swt.layout,
 org.eclipse.swt.widgets,
 org.osgi.framework,
 org.osgi.service.component
```

OSGI-INF/component.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<component name="aQute.swtchat.component">
  <implementation class=
    "aqute.swtchat.ChatWindow"/>
</component>
```

# Using the Chat Library

- The activate method creates a new Chat instance and starts the thread
- The deactivate method sets a quit flag and interrupts the running thread so that it can exit
- The run method creates and opens a new window
  - Reads quit flag to determine when to close the application

ChatWindow.java

```
protected void activate(
    ComponentContext context)
    throws Exception {
  this.chat = new Chat(
  context.getBundleContext(), this);
  start();
}
protected void deactivate(
    ComponentContext context)
  throws Exception {
  quit = true; interrupt();
}
public void run() {
  createShell();
  shell.open();
  display = shell.getDisplay();
  while (!shell.isDisposed() && !quit)
  try {
    if (!display.readAndDispatch())
      display.sleep();
  } catch (Exception e) {
    error(e);}
  if (!shell.isDisposed())
     shell.dispose();chat.close();
}
```

# Using the Chat Library

- The usual window verbosity …
- The window creates:
  - A shell (the window itself),
  - A text field that will contain the chat output, and
  - A line field that will contain the chat input

```
void createShell() {
  shell = new Shell();
  shell.setText("SWT Chat");
  GridLayout layout = new GridLayout();
  layout.numColumns = 1;
  shell.setSize(500, 300);
  shell.setLayout(layout);
  text = new Text(shell,
    SWT.MULTI|SWT.BORDER|SWT.MULTI|SWT.V_SCROLL
|SWT.READ_ONLY);
  GridData spec = new GridData();
  spec.horizontalAlignment = GridData.FILL;
  spec.grabExcessHorizontalSpace = true;
  spec.verticalAlignment = GridData.FILL;
  spec.grabExcessVerticalSpace = true;
  text.setLayoutData(spec);
  line = new Text(shell,
   SWT.MULTI|SWT.BORDER|SWT.V_SCROLL);
  spec = new GridData();
  spec.horizontalAlignment = GridData.FILL;
  spec.grabExcessHorizontalSpace = true;
  spec.verticalAlignment = GridData.FILL;
  spec.grabExcessVerticalSpace = false;
  spec.heightHint = 40;
  line.setLayoutData(spec);
  line.addKeyListener(this);
}
```

# Using the Chat Library

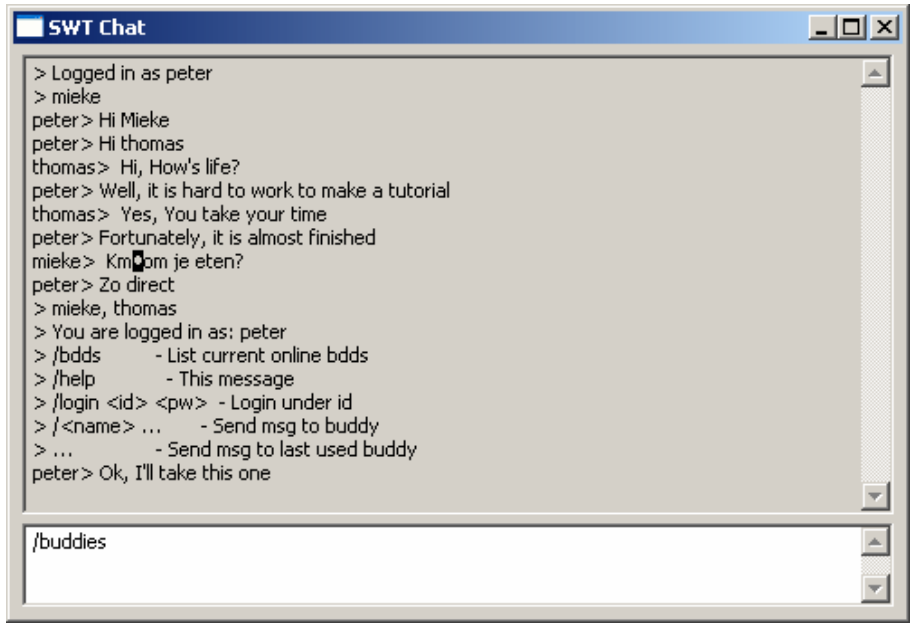
- The keyReleased method processes when the new line is entered
- The error method displays an error to the user
- The send method displays a message to the user from another client

```
public void keyPressed(KeyEvent ev){}
public void keyReleased(KeyEvent ev){
  switch (ev.keyCode) {
    case SWT.CR : try {
      String txt = line.getText();
      chat.execute(txt.trim());
      line.setText("");
    } catch (IOException e1) {
      error(e1);
    }
    break;
  }
}

void error(Exception e) {
  if (!quit) {
    text.append("error> ");
    text.append(e + "");}}

public void send(String from, String str) {
  display.asyncExec(new Runnable() {
    public void run() {
      text.append(from + "> " + str
      + "\r\n");
    }
  });
}
```

# Running the SWT Chat

- Run the SWT Chat in the normal way
  - Verify the target bundles



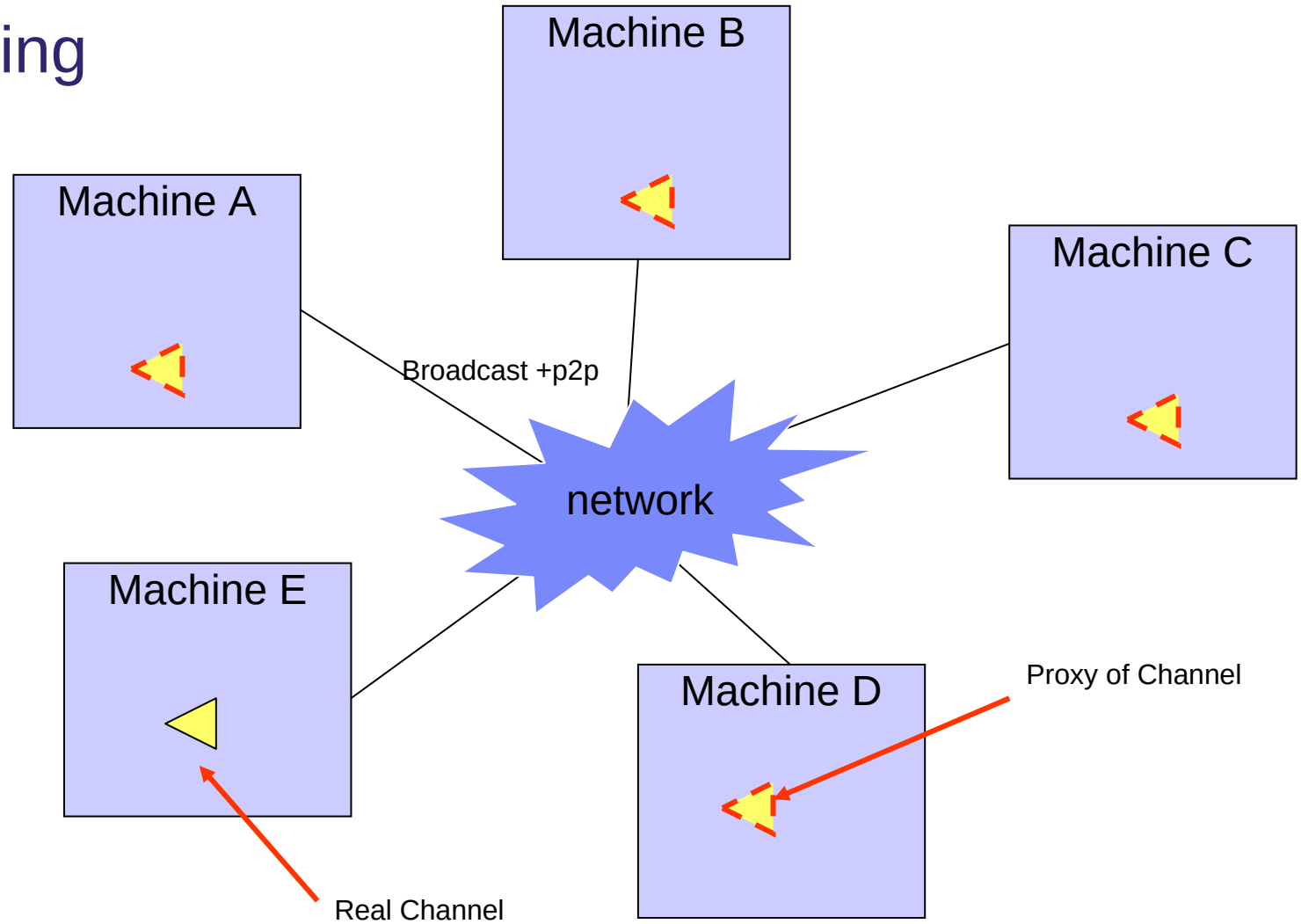
# Remoting

- The chat is kind of *boring* because it only works on our own laptop. Missing is discovery of each other's bundles! Rescue is on the way …
- Enable the aQute.remoting project
  - This project can export services to other participating machines
- The only requirement is that you have a property remote=* on your service. See code
- The aQute.remoting project will then export this service to any participating machine
- Our Channel objects will therefore be spread all over the place
- Enable the aQute.remoting bundle, launch, and test with your buddies.

```
public void login(String name,
  String password) throws IOException{
   if (registration != null)
    registration.unregister();
   registration = null;
   this.name = name;
   Hashtable properties =
     new Hashtable();

   properties.put("remote", "*" );
   properties.put(
    Constants.SERVICE_PID,
    "pid:chat[" +
    InetAddress.getLocalHost()
    + "]:" + name);
   properties.put(
     Channel.CH_NAME, name);
  registration = cntxt.registerService(
    Channel.class.getName(),
     user, properties);
  user.send("", "Logged in as "
    + name);
}
```

# Remoting



# What did we learn?

- That the OSGi Service Registry is a surprisingly powerful model for collaboration
- The decoupling that it promotes allows additional functionality without influencing existing functions

# What We Did Not Learn

- Security Architecture
- Permission Management
- Signed Bundles
- Package Management
- Bundle Life Cycle Management
- Configuration Management and Preferences
- Servlet Support/Web Server
- Device Access
- Event Manager

- UPnP
- User Admin
- Wire Admin
- Application Model
- Deployment Admin and Autoconf
- Device Management Tree
- Initial Provisioning
- Position, Measurement, State
- MetaType
- And much, much, more

# Conclusion

- The OSGi R4 Specifications consists of considerable more details than elucidated in this tutorial
- There are many independent OSGi implementations on the market, both commercial and open source
  - Apache Felix, Atinav, Eclipse Equinox, Espial, IBM® SMF, Knopflerfish/Ubiserv of Gatespace, ProSyst, …
- The OSGi specifications are running today on mobile phones, PDAs, embedded computers, desktops, and mainframes
- Both in managed and unmanaged configurations
- The OSGi specifications solve real world problems
- The OSGi Alliance is working on making the OSGi specifications *the* standard for portable applications. Join us!

# The End

Further reading:

http://www.eclipse.org/equinox/
http://www.osgi.org
http://bundles.osgi.org
http://www.aqute.biz

# Legal Notices

- IBM is a registered trademark of International Business Machines Corp. in the United States, other countries, or both.
- Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.
- Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.
- Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.
- Other company, product, or service names may be trademarks or service marks of others.