Cruft!

Paris, France

Peter Kriens, CEO, aQute
• *cruft* is redundant, old or improperly written code which needs to be fixed, but tends to stick around.
Why Is Cruft Bad!

• More work to write
• Adds redundancy
• More potential for errors
• Harder to read and understand
• Harder to extend
• Harder to optimize
Low-Cruft Languages

- **APL**
  - $\text{PRIMES} : (\sim R \in R \cdot \times R) / R \leftarrow 1 \cdot R$
  - Maybe a trifle too condense

- **Smalltalk**
  - (Integer primesUpTo: 100)
  - max
  - Extremely low cruft syntax without becoming unreadable

- **Compare this to**

```java
package primes;
public class Primes {
    public static void main(String[] args) throws Exception {
        int max = 10000000;
        boolean[] isprime = new boolean[max + 1];
        for (int i = 0; i <= max; i++)
            isprime[i] = true;
        isprime[0] = isprime[1] = false;
        int n = (int) Math.ceil(Math.sqrt(max));
        for (int i = 0; i <= n; i++) {
            if (isprime[i])
                for (int j = 2 * i; j <= max; j = j + i)
                    isprime[j] = false;
        }
        int largest;
        for (largest = max; !isprime[largest]; largest--)
            ; // empty loop body
        System.out.println(largest);
    }
}
```
High Cruft Languages

- Java is a very verbose language
  - Simple things require an amazing amount of code
  - Type safe languages have this tendency. Sometimes one wonders if the type safety is needed to handle the cruft that it causes …

- OSGi adds additional cruft
  - Service dependencies
  - Bundle dependencies

- Using OSGi is very intrusive in the application code
  - High learning threshold
  - Programmers are initially appalled when they have not yet seen the advantages
So What?

- Let's analyze how much cruft is added by creating an OSGi service
package aQute.opm.echo;

public interface Echo {
    String echo(String msg);
}

A Logging Echo Service
package aQute.opm.echo.plain;
import java.util.*;
import org.osgi.framework.*;
import org.osgi.service.cm.*;
import aQute.opm.echo.Echo;

public class Activator implements BundleActivator, ManagedService {
    PlainEcho echo;
    public void start(BundleContext context) throws Exception {
        Dictionary map = new Hashtable();
        map.put("service.pid", "aQute.opm.echo.plain");
        echo = new PlainEcho(context);
        context.registerService(Echo.class.getName(), echo, map);
        map.put("service.pid", "aQute.opm.echo.plain.ms");
        context.registerService(ManagedService.class.getName(), this, map);
    }

    public void stop(BundleContext context) throws Exception {
    }

    public void updated(Dictionary map) throwsConfigurationException {
        if (map == null) return;
        echo.toupper = "true".equals(map.get("toupper"));
    }
}
package aQute.opm.echo.plain;
import java.util.*;
import org.osgi.framework.*;
import org.osgi.service.cm.*;
import aQute.opm.echo.Echo;
public class Activator implements BundleActivator, ManagedService {
    PlainEcho echo;
    public void start(BundleContext context) throws Exception {
        Dictionary map = new Hashtable();
        map.put("service.pid", "aQute.opm.echo.plain");
        echo = new PlainEcho(context);
        context.registerService(Echo.class.getName(), echo, map);
        map.put("service.pid", "aQute.opm.echo.plain.ms");
        context.registerService(ManagedService.class.getName(), this, map);
    }
    public void stop(BundleContext context) throws Exception {}
    public void updated(Dictionary map) throws ConfigurationException {
        if (map == null) return;
        echo.toupper = "true" .equals(map.get("toupper"));
    }
}
package aQute.opm.echo.plain;

import org.osgi.framework.BundleContext;
import org.osgi.service.log.LogService;
import org.osgi.util.tracker.ServiceTracker;

import aQute.opm.echo.Echo;

public class PlainEcho implements Echo {
    boolean toupper;
    ServiceTracker logTracker;

    PlainEcho(BundleContext context) {
        logTracker = new ServiceTracker(context, LogService.class.getName(), null);
    }

    public String echo(String msg) {  
        if (toupper)             
            msg = msg.toUpperCase();
        LogService log;
        try {
            log = (LogService) logTracker.waitForService(10000);
        } catch (InterruptedException e) { return msg; }
        if (log != null) log.log(LogService.LOG_INFO, msg);
        return msg;
    }
}
package aQute.opm.echo.plain;

import org.osgi.framework.BundleContext;
import org.osgi.service.log.LogService;
import org.osgi.util.tracker.ServiceTracker;
import aQute.opm.echo.Echo;

public class PlainEcho implements Echo {
  boolean toupper;
  ServiceTracker logTracker;

  PlainEcho(BundleContext context) {
    logTracker = new ServiceTracker(context, LogService.class.getName(), null);
  }

  public String echo(String msg) {
    if (toupper)
      msg = msg.toUpperCase();
    LogService log;
    try {
      log = (LogService) logTracker.waitForService(10000);
    } catch (InterruptedException e) { return msg; }
    if (log != null)   log.log(LogService.LOG_INFO, msg);
    return msg;
  }
}
Declarative Services?

• In R4 we added Declarative Services for exactly this reasons. And Service Binder exists for some time. Are those not the answer?

• Lets try …
package aQute.opm.echo.declarative;

import org.osgi.service.component.ComponentContext;
import org.osgi.service.log.LogService;
import aQute.opm.echo.Echo;

public class DeclarativeComponent implements Echo {
    boolean toupper;
    ComponentContext context;

    protected void activate(ComponentContext context) {
        this.context = context;
    }

    public String echo(String msg) {
        if (toupper)
            msg = msg.toUpperCase();
        LogService log = (LogService) context.locateService("log");
        log.log(LogService.LOG_INFO, msg);
        return msg;
    }
}
package aQute.opm.echo.declarative;

import org.osgi.service.component.ComponentContext;
import org.osgi.service.log.LogService;
import aQute.opm.echo.Echo;

public class DeclarativeComponent implements Echo {
    boolean toupper;
    ComponentContext context;

    public void activate(ComponentContext context) {
        this.context = context;
    }

    public String echo(String msg) {
        if (toupper)
            msg = msg.toUpperCase();
        LogService log = (LogService) context.locateService("log");
        log.log(LogService.LOG_INFO, msg);
        return msg;
    }
}
<?xml version="1.0" encoding="UTF-8"?>
<component
    name="aQute.opm.echo.declarative"
    xmlns="http://www.osgi.org/xmlns/scr/v1.0.0">
    <implementation class="aQute.opm.echo.declarative.DeclarativeComponent"/>
    <service>
        <provide interface="aQute.opm.echo.Echo"/>
    </service>
    <reference name="log"
        interface="org.osgi.service.log.LogService"/>
</component>
<?xml version="1.0" encoding="UTF-8"?>
<component
  name="aQute.opm.echo.declarative"
  xmlns="http://www.osgi.org/xmlns/scr/v1.0.0">
  <implementation class="aQute.opm.echo.declarative.DeclarativeComponent"/>
  <service>
    <provide interface="aQute.opm.echo.Echo"/>
  </service>
  <reference name="log"
    interface="org.osgi.service.log.LogService"/>
</component>
• Lets analyze the problem

• How would this example look like with only the Java cruft?
package aQute.opm.echo.opm;

import org.osgi.service.log.LogService;

import aQute.opm.echo.Echo;

public class Component implements Echo {

  private LogService log;

  boolean toUpper;

  public String echo(String message) {
    if (toUpper)
      message = message.toUpperCase();
    log.log(LogService.LOG_INFO, message);
    return message;
  }
}
What is Missing?

• What would a handler need to run this example?
  – The name of the class
  – The service that must be registered for this component: Echo
  – The dependency on the Log Service with the log variable

• Lets design a manifest header …
Manifest Header

Manifest-Version: 1.0
Bundle-Name: aQute.opm.echo.opm
Bundle-Activator: scr.ComponentRuntime
Component: aQute.opm.echo.opm.Component;
  mandatory=log;
  objectclass=aQute.opm.echo.Echo;
  config=toupper
$(IMPORT-PACKAGE)
$(EXPORT-PACKAGE)
Manifest Header

Manifest-Version: 1.0
Bundle-Name: aQute.opm.echo.opm
Bundle-Activator: scr.ComponentRuntime
Component: aQute.opm.echo.opm.Component;
  mandatory=log;
  objectclass=aQute.opm.echo.Echo;
  config=toupper
$(IMPORT--PACKAGE)
$(EXPORT--PACKAGE)
package aQute.opm.echo.opm;

import org.osgi.service.log.LogService;
import aQute.opm.echo.Echo;

public class Component implements Echo {
    private LogService log;
    boolean toUpper;

    public String echo(String message) {
        if ( toUpper )
            message = message.toUpperCase();
        log.log(LogService.LOG_INFO, message);
        return message;
    }
}

Manifest-Version: 1.0
Bundle-Name: aQute.opm.echo.opm
Bundle-Activator: scr.ComponentRuntime
Component: aQute.opm.echo.opm.Component;
    mandatory=log;
    objectclass=aQute.opm.echo.Echo;
    config=toupper
$(IMPORT-PACKAGE)
$(EXPORT-PACKAGE)
The Component Handler Structure

- opm
  - Get the manifest info
  - Register service as a ServiceFactory
  - Create service when needed

- echo
  - Activator

OSGi Alliance
How Are the Dependencies Handled?

- The Component uses the log instance variable without any concerns for dependencies or initialization.
- The handler only gets the name of the fields, not the types.
- The Handler should be careful not to set the log field when it is never needed.
  - Lazy initialization, do not consume services when not needed.
Byte Code Weaving

- The handler can do its magic if it could change/read the byte codes of the class
  - Types of variables are defined in the byte code
  - References to dependents can be changed to a callback

- The OPM prototype:
  - Replaces all service references with its own object, a ComponentDependency
  - All references to the service are replaced with a callback to the ComponentDependency
Implementation

• The bundle-activator is loaded from the OPM bundle.
  – Reads manifest
  – Analyzes Component class for missing information
  – Starts dependency checking
  – Activates Components when dependencies are met

• The Component class is analyzed and rewritten using a custom class loader
  – Gets the bytes from the class loader of the application bundle
  – Uses ASM from Objectweb

• If the Framework standardized the custom hook for the Declarative Service, then OPM could use it as well
  – No OPM Activator would then be necessary
public class Component implements Echo{
private LogService  log;
boolean toUpper;
public Component();
  Code:
    0:   aload_0
    1:   invokespecial
    4:   return
public String echo(String);
  Code:
    0:   aload_0
    1:   getfield #toupper
    4:   ifeq 12
    7:   aload_1
     8:   invokevirtual #toUpperCase()
   11:  astore_1
   12:  aload_0
     13:  getfield #log
   16:  iconst_3
   17:  aload_1
    18:  invokeinterface #log()
   21:  aload_1
   22:  invokevirtual #get
   25:  invokeinterface #LogService
   28:  areturn
}

public class Component implements Echo{
private ComponentDependency log;
boolean toUpper;
public Component();
  Code:
    0:   aload_0
    1:   invokespecial
    4:   return
public String echo(String);
  Code:
    0:   aload_0
    1:   getfield #toupper
    4:   ifeq 12
    7:   aload_1
     8:   invokevirtual #toUpperCase()
   11:  astore_1
   12:  aload_0
     13:  getfield #log
   16:  invokevirtual #get
   19:  checkcast #LogService
   22:  iconst_3
   23:  aload_1
    24:  invokevirtual #log()
   27:  areturn
}
Issues

- **Performance**
  - Access to services is slightly slower
- **More memory usage**
  - Tracking the dependencies requires a number of objects per service dependency
- **Change in byte code**
  - Debuggers, though so far it looks ok
  - Some developers do not like changes in their byte codes
- **It is uncommon for developer that their private variables change from the outside**

---

*OSGi Alliance*
• Configuration Data
  – Instance variables can be declared to come from configuration data
  – Automatic type coercion

• Use of Managed Service Factories to drive the 0..n component creations
  – Very useful for services that can be instantiated multiple times with different configurations

• Use of Java annotations instead of manifest header

• Preprocessing during development

• Use of standard AOP syntax

• Use implemented interfaces as services

• Big trick is to keep it simple …
Conclusion

- The OPM uses a very specific form of byte code weaving that is popularized by Aspect Oriented Programming (AOP)
- AOP turns out to be extremely useful to remove cruft
- The OPM prototype shows that it is possible to remove all OSGi related aspects from code that runs on an OSGi Service Platform
  - Without losing any of the dynamics
For a Cruft Free World