Paremus

Just-in-Time Java EE:

Provisioning Runtimes for Enterprise Applications

Tim Ward

http://www.paremus.com
info@paremus.com

Who is Tim Ward?
@TimothyWard

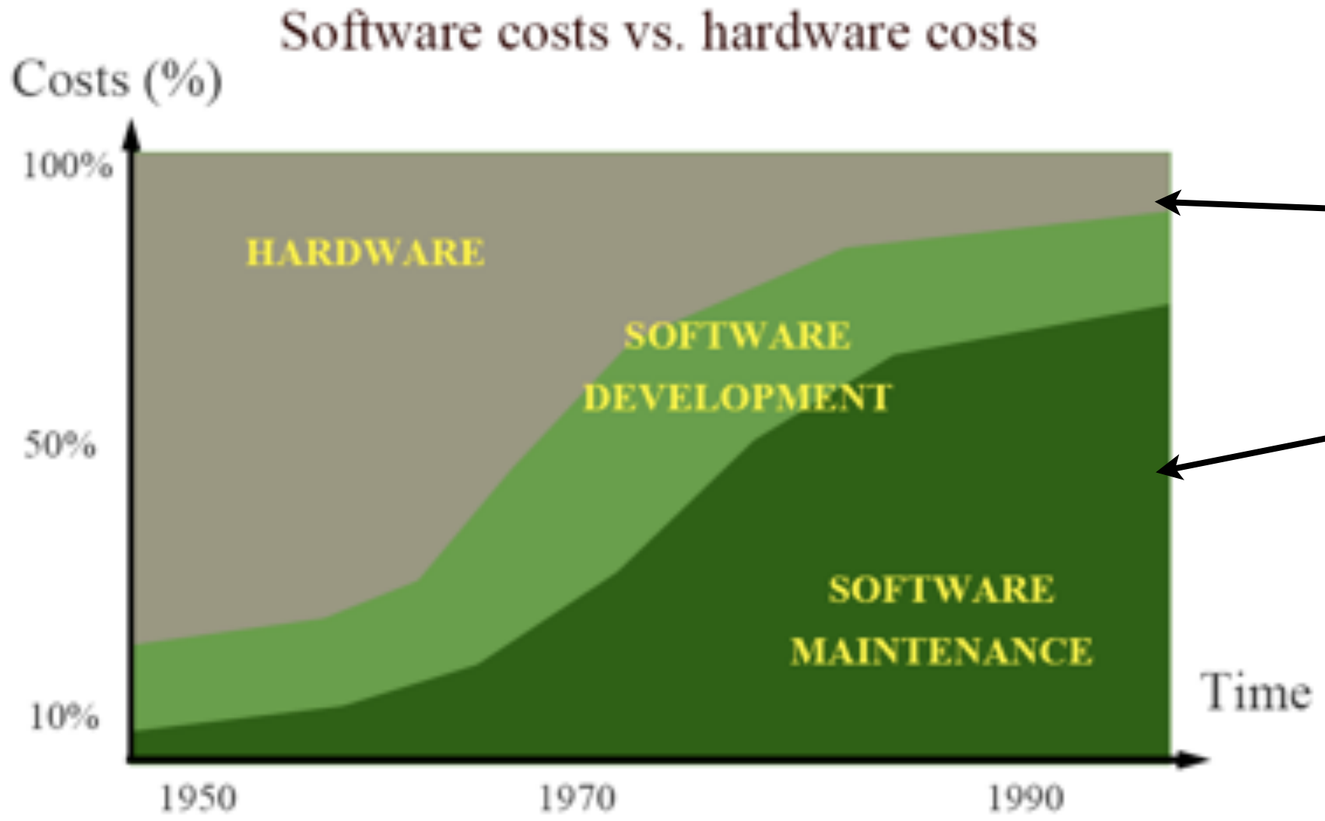
- Senior Consulting Engineer and Architect at Paremus
- 5 years at IBM developing WebSphere Application Server
  - Container Implementation experience with Java EE and OSGi, including Blueprint, JPA, EJB and JTA
- OSGi Specification lead for JPA and Bytecode Weaving
- PMC member of the Apache Aries project
- Previous speaker at EclipseCon, Devoxx, Jazoon, JAX London, OSGi Community Event...
- Author of Manning's Enterprise OSGi in Action
  - http://www.manning.com/cummins

What we're going to cover

- Why we need more Modularity
- Using OSGi to provision applications
- Deploying external processes using Packager
- Building Dynamically wired distributed systems with OSGi remote services

Building Maintainable Systems Using Modularity

The True Cost of Software



Hardware is cheap

Maintenance is not!

Anne Thomas Manes (Gartner)
SOA Symposium: Berlin, October 2010

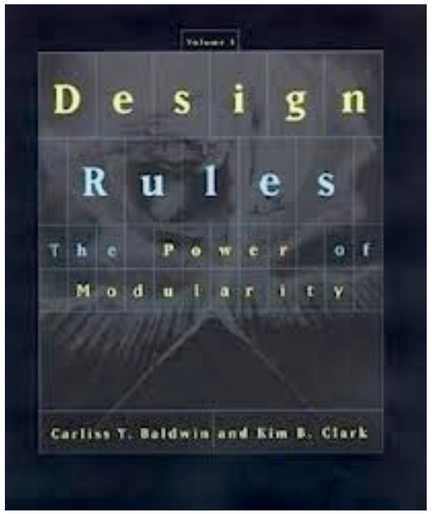
- To reduce cost we need to focus on making maintenance cheaper, not on better Hardware Utilization!

Modular Systems are Maintainable Systems

Modularization serves three purposes, any of which may justify an investment:

- Modularity makes complexity manageable;
- Modularity enables parallel work; and
- Modularity is tolerant of uncertainty.

"tolerant of uncertainty" means that particular elements of a modular design may be changed ***after the fact*** and ***in unforeseen ways*** as long as the design rules are obeyed.



Design Rules, Volume 1: The Power of Modularity (MIT Press, 2000)
http://www.amazon.com/Design-Rules-Vol-Power-Modularity/dp/0262024667

Carliss Y. Baldwin
Kim B. Clark
Harvard Business School

Modularity in the Cloud

- Cloud platforms can benefit from all of these things!
- Enterprise Cloud applications are inherently complex
  - Huge Data/Request volumes
- Clouds only really work for scalable applications
- Cloud platforms are very unstable places
  - DataCentres go down
  - VMs get moved

VMs as the unit of modular Cloud Deployment

VMs as the unit of Cloud Deployment

- Deployment and management is a huge part of life in the Cloud
  - VMs are a standard unit of deployment

- Devops automation is necessary for managing VMs

- VMs are *BIG*, usually tens of GB, sometimes hundreds

- A one byte “update” costs a whole new VM

VM Word association

?

BIG small

fast slow

dynamic static

Modular?

VMs as Monoliths

- Platform VMs are classic monoliths
  - They contain *everything* that you have

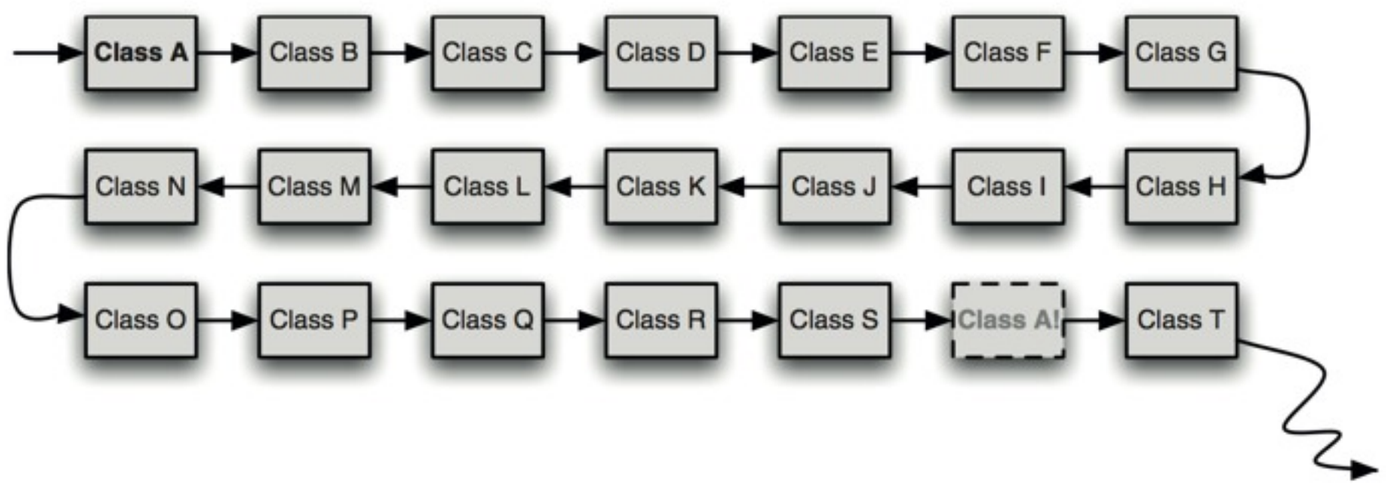
- Devops try to reduce maintenance costs
  - Often a base VM is a template for multiple services

- A single change to any component requires a whole new VM, and could break every user of that VM!
  - Our maintainance problem just got even worse!

A Clever New Solution!

Haven’t we solved this before?

- Cloud applications are huge
  - Size is relative - Clouds are huge too!
- Java Application Servers contain hundreds of JARs
  - Hundreds of thousands of Classes!
- Duplicates and missing classes are horrible to diagnose



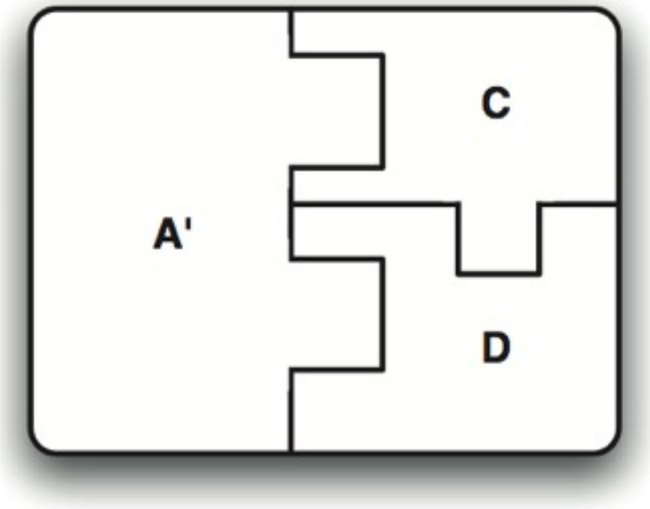
Haven’t we solved this before (2)?

- Java EE vendors also borrowed an existing solution!

- JSR 8 - Open Services Gateway
  - An embedded server running Java
  - Designed for resource constrained systems
  - Pluggable and extensible

- This became the OSGi framework you know and love!
  - So how does OSGi manage complexity?

How OSGi works



- OSGi modules are called “bundles”
  - Bundles are JARs - with a twist
  - The manifest has modularity metadata
- Bundles list the things that they depend on, and the things they expose
  - If it isn’t exported it can’t be imported!
- Your bundle can only “start” when all its dependencies are satisfied
- Exports and Imports have versions for compatibility
  - Semantic Versioning is crucial for managing change

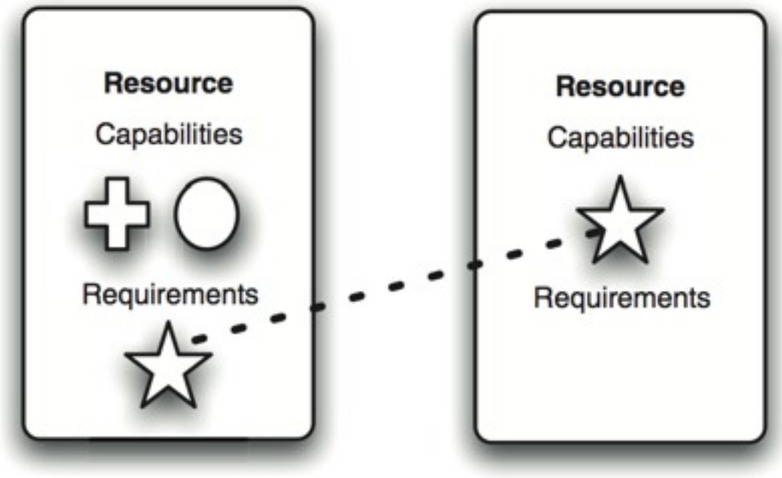
Resolution and Provisioning using OSGi metadata

Deploying to an OSGi framework

- Deployment in OSGi framework is similar to in a cloud
  - An OSGi framework is like a cloud in a JVM
  - A collection of dependencies must be installed
- Unlike most things OSGi bundles are *self-describing*
  - You can look at an OSGi bundle and know whether it will run in your system!
- The OSGi resolver uses this information at runtime to determine whether your bundle can be run or not
  - Until your bundle resolves it is totally inert!

Provisioning and Resolution

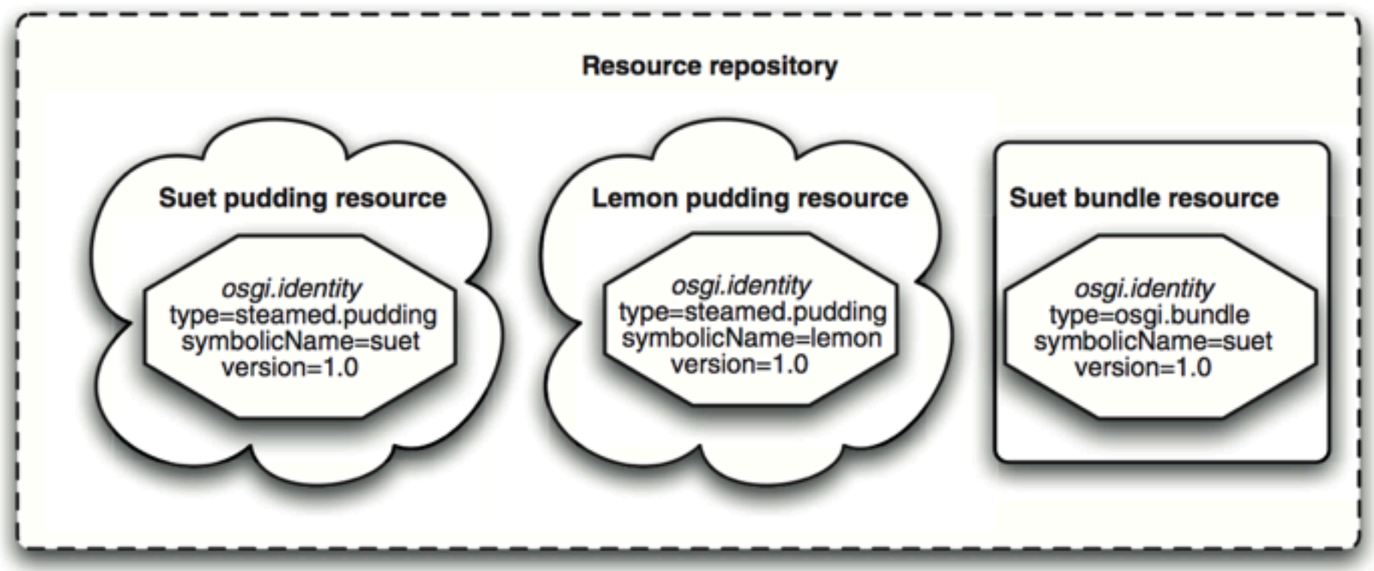
- The OSGi resolver works in the context of the existing resources in the framework
  - But you can add Repositories!
- When the resolver finds an unsatisfied requirement it asks the repository for candidates



- By providing a suitable repository you can ensure only “approved” dependencies are used
- The eventual solution is then “pulled” into the framework so the application can run

Generic Requirements and Capabilities

- Historically there were no repositories, and the resolver was an implementation detail...
  - No provisioning, only package/bundle dependencies
- Generic Resources and the Resolver API since OSGi R5



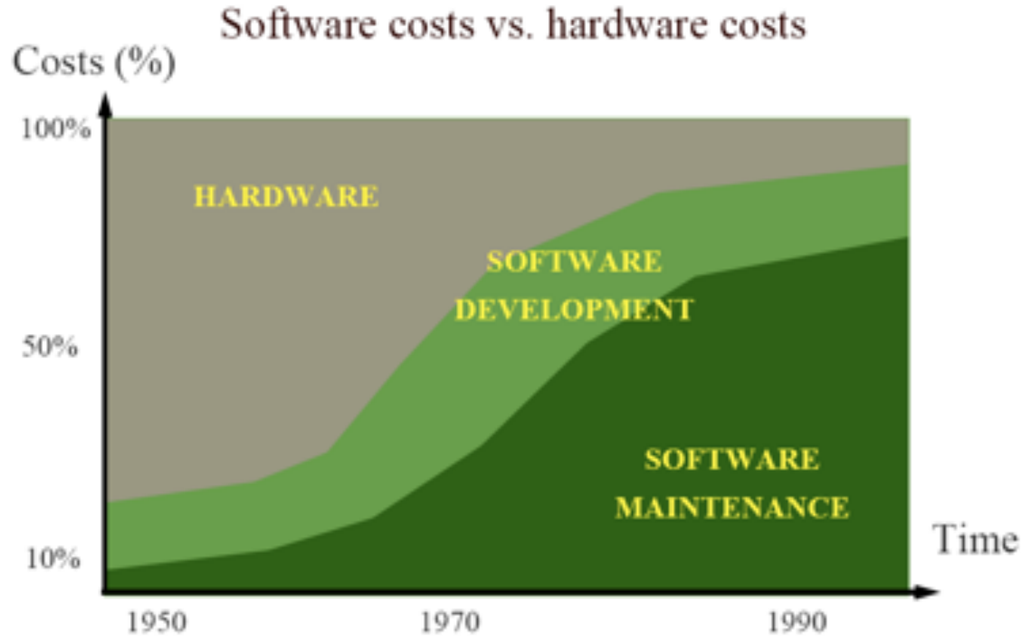
Generic Requirements and Capabilities (2)

- OSGi Requirements and Capabilities can express any type of dependency using namespaces
  - OSGi itself has been extended with new types!
- Indirect dependencies on runtime services can now be expressed, building a whole stack around your bundle!
  - Attributes can be used to communicate specifics about an implementation (just like normal OSGi)
  - Directives can affect cardinality, and “effectiveness”

Demo (1)

But what about my existing code?

- Existing software is our biggest cost!
- We can't afford to rewrite it
- What if we don't want to write OSGi bundles and Java?



- Polyglot OSGi is a reality
  - Scala Modules, C/C++ and JavaScript are all active
  - These still require your code to be run in OSGi

- We could use OSGi as a provisioning and lifecycle layer without running in the same process...

Packaging Applications with Packager

Packager Aims and advantages

- Packaged programs are self-describing OSGi Bundles
  - Allows Dynamic Resolution and Assembly
  - Can be semantically versioned

- Leverage OSGi Alliance Specifications
  - Common Lifecycle and API
  - Dynamic Configuration,
  - Easy acess to Local and Remote Services

- Allow existing code to be packaged without change

- Open Source API for reusable packaged components
  - Also allows competing Packager implementations

Approach

- Wrap Native Artifacts in OSGi Bundles
- **Link the Artifact Lifecycle to OSGi.**
  - Bundle Install/Resolve/Start => Artifact “install”
  - Service registration/unregistration => Artifact “run/stop”
  - Bundle Uninstall => Artifact “uninstall”
- Link to Standard OSGi Services: Configuration Admin, Metatype, Log Service...

```
<system name="BackEnd:MongoDB" boundary="fibre">
  <!-- MongoDB package -->
  <system.part category="msf" name="com.paremus.packager.demos.mongo.guard">
        <property name="port" value="27017" />
  </system.part>
</system>
```

Separation of Concerns

- What are we running, and how do we configure/run it?
  - **Package Type Service**

- When should we start it, and what configuration properties should we use?
  - **Process Guard Service**

- Actually invoke the start/stop scripts, and monitor the state of the external process
  - **Watchdog Process**

Package Type

- Usually **contains** the native parts
  - **Installs** the Native Program
- Configures the program based on the supplied config
- Returns **scripts** to the Packager for: start, stop, ping...
- Is (usually) **platform specific**

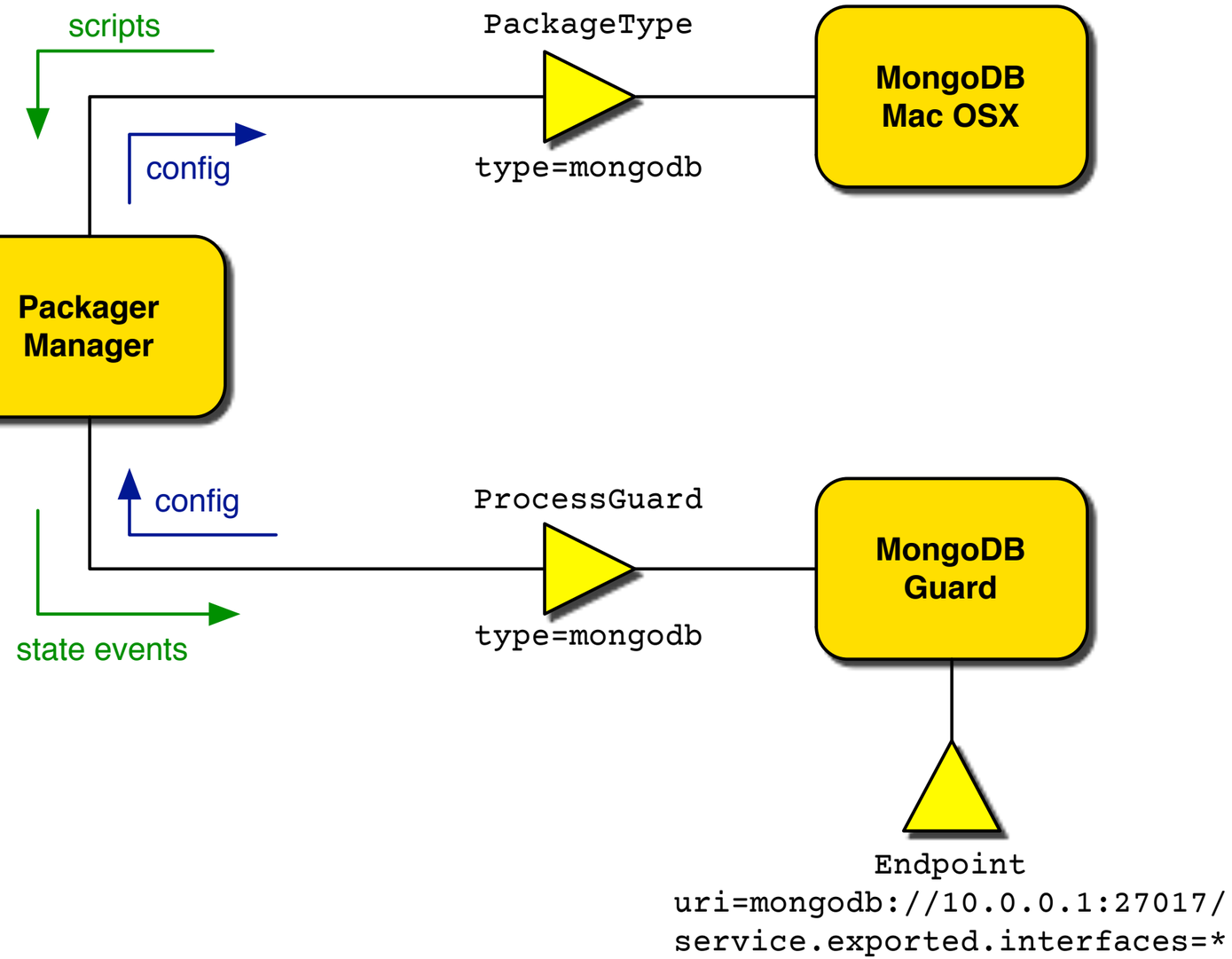
```
Provide-Capability: packager.type;
          packager.type=mongodb;
          version:Version=2.2.0

Require-Capability: osgi.native;
          filter:=”(&
              (osgi.native.osname=Linux)
              (osgi.native.processor=x86-64)
          )”
```

Process Guard Service

- **Gathers** Process Config Properties
- Existence signals **When** to Start/Stop
- Receives State Change **Events** (started, crashed...)
- **Advertises** the Running Package (Optional)
- Usually platform independent

Packager Interactions



Packager Runtime Flow

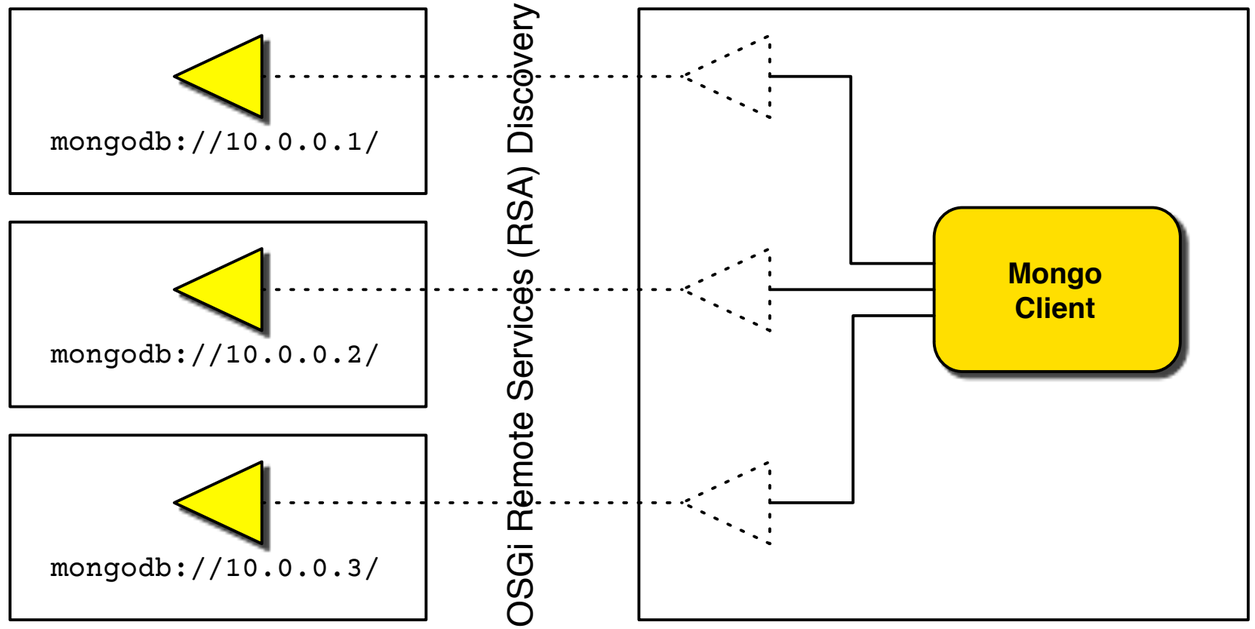
1. **Process Guard** plus **Package Type** Means “Start”
2. **Process Guard** queried for config properties
3. **Package Type** creates config, start and stop scripts
4. **Watchdog** launches external process
5. **Process Guard** notified of success/failure/restart
6. If either service is unpublished then “Stop”

Publishing an Endpoint

- Endpoint Services have a Marker Interface and a URI Property

- This may not seem very useful, but it can be used to establish the location of the process
  - It can also trigger other things...

- OSGi Remote Services mean that you can even notify other frameworks of your location



Extending Packager with the OSGi Extender pattern

Adding Requirements and Capabilities

- Earlier we saw the following:

```
Provide-Capability: packager.type;
          packager.type=mongodb;
          version:Version=2.2.0

Require-Capability: osgi.native;
          filter:="(&
              (osgi.native.osname=Linux)
              (osgi.native.processor=x86-64)
          )"
```

- Uses the default *osgi.native* for platform dependency
- Exposes the new *packager.type* capability

Adding Requirements and Capabilities (2)

- Automatic runtime provisioning of packaged artefacts is easy
  - Process Guard has requirements for
    - Packager Manager
    - A “matching” *package.type*

- Application bundles that need to be co-located with the external process can depend on the Process Guard
  - Another options is to reference the guard directly

- But what if you need more configuration?

The OSGi Extender pattern

- The Extender Bundle Pattern is a powerful tool in OSGi
  - An Extender Bundle searches for other bundles that can be *extended* in some way
  - Usually marked by a Requirement or a Manifest header

- What if we use the extender pattern to configure our Process Guard services?
  - The bundles we extend should be self-describing
    - We know what we need to configure!

Applying Packager to Java EE

- Java EE applications need a Java EE runtime
- Use Packager to install/run the server
- Use the extender pattern to infer the configuration!
  - This requirement can fully configure a JPA app!

```
Require-Capability: com.paremus.javaee;
          filter:="(javaee.version>=6)";
          context.roots="jboss-as-kitchensink-ear-web";
          database.jta.name=kitchensink;
          database.jta.jndi.name="jdbc/kitchensink";
          database.jta.xa="false";
          database.nojta.name=kitchensink;
          database.nojta.jndi.name="jdbc/noJTAkitchensink";
          database.nojta.xa="false";
          database.nojta.jta="false"
```

Waiting for dependencies...

- Earlier I said that Endpoints could be used as triggers
  - This Java EE application needs a database endpoint!
- The Extender gathers the necessary configuration and dependency information from the Java EE bundle
- When a database endpoint is available the extender completes the configuration and registers the Process Guard
  - The implementation and location of the database is automatically determined at runtime!
  - The system can rewire itself if the database moves!

Demo (2)

Platforms must be Modular & Adaptive

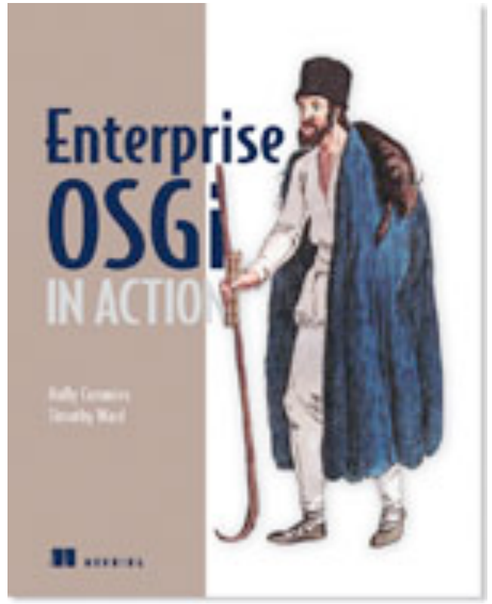
Things Change over time, and no two systems are the same

Tight coupling in complex environments is the root cause of catastrophic cascading failure, or Black Swan events.
(see https://blogs.paremus.com/2012/08/complex-systems-and-failure/).

A Deployment mechanism that fails to acknowledge change, that fails to manage runtime dependencies, drives increasing environmental complexity and OPEX.

Thanks!

- For more about OSGi...
  - Specifications at http://www.osgi.org
  - Enterprise OSGi in Action
  - http://www.manning.com/cummins



45% off using code mljo13cf at manning.com

- For more about Packager...
  - https://docs.paremus.com/display/SF19/Packager

Questions?

http://www.paremus.com
info@paremus.com