BLASTING OUT OF THE PAST WITH OSGi
Agenda

• Prologue: What the Slashers Found
• Chapter 1: The Story So Far
  – Review of the LiveOps Architecture and drivers towards migration and modernization
• Chapter 2: Requiem for a Legacy
  – Exploration of approaches to the problem, and patterned approaches to specific problems
• Chapter 3: No Country for Old Software
  – How LiveOps uses OSGi as a migration and modernization vehicle
• Chapter 4: What Apps May Come
  – Where we go from here, and how OSGi will continue to help us migrate
CHAPTER 1: THE STORY SO FAR
Legacy Platform Capabilities

• Purpose: to provide a SaaS/PaaS product for contact center functionality
• Features
  – Multiple inbound/outbound media channels: PSTN or VoIP voice, email, web chat, SMS, Twitter, Facebook
  – Interaction flow processing
  – Cradle-to-grave reporting
  – Voice and screen recording
  – Agent, Admin, Supervisor applications
• Constraints
  – 24/7 availability; no planned downtime
  – Support for:
    • 8,000+ “live” customer contacts (voice + web chat)
    • Unlimited off-line contacts
    • 5,000+ concurrent contact center agents
Platform Technology Milestones

- 2000 – LiveOps founded in Florida as a BPO
- 2001 – Callcast founded in California as a contact center tech service
- 2003 – LiveOps/Callcast companies technologies merge
- 2005 – SaaS/PaaS business launched, larger customers added
- 2007 – Fortune 500 customers added
- 2008 – Marketplace/LiveWork technology integration, PCI Compliance
- 2009 – Enterprise Agent product launched, continued growth
- 2010 – First OSGi instances: REST API and Salesforce.com integration launched
- 2011 – Integrated multichannel tech stack, Data Exchange product launched on OSGi
- 2012 – LiveOps Application Server project started and launched
Tech Survey – Telephony and Presence

• All components native to LiveOps
  – Mostly Java-based components
    • One exception: Media Server (C++)
  – SIP stack built and maintained in-house
  – All components are redundant and load balanced

• Call Manager
  – Executes call flows
  – Performs whole-call recording

• IVR
  – For self-service, plays announcements, menus, etc. into the call

• Presence
  – Tracks availability of agents for work assignment

• SIP Proxy
  – Manages assignment of inbound calls to call managers
  – Ensures proper routing of outbound calls via proxy rules
Tech Survey – Web Applications

• LAMP stack (general admin apps)
  – Perl/Mason under Apache 1.3/mod_perl
  – Ruby on Rails under Apache 2/Passenger

• Windows Server stack (agent applications)
  – IIS
  – ASP.NET
Tech Survey – Data Management

- Database: MySQL 5.1 (MyISAM and InnoDB)
- Configuration Data
  - MySQL – e.g. ccconf, callflow, nexus databases
  - Slow changing, relatively small in size
- Application Data
  - MySQL – e.g. ccroute, ccoutbound databases
  - Call managers, API servers
  - Comparable read/write frequency, bounded growth
- Historical Data
  - MySQL – cclog and datamart databases
  - Hadoop, ActiveMQ – fine-grained event/log data pipeline
  - GreenPlum – distributed analytical database
  - Ever growing, relatively large in size
- Real-time Data
  - RTM pipeline – scalable delivery to RTM dashboards
  - Rapidly changing, relatively small in size
Tech Survey - APIs & Integration

- Pre-OSGi APIs
  - Transfer Data Service
  - PAPI
  - ACS
- Desktop JavaScript API
  - For integration with browser-based agent client apps
  - Full control of agent capabilities (presence, call handling, etc.)
- Platform Services API (PSAPI)
  - User and configuration management
  - Live call control
  - Data transfer and statistics access
- Data Exchange
  - Data dips for intelligent routing
  - External record creation and update
  - Bulk data export and import (reporting and configuration data)
  - Configuration driven
CHAPTER 2: REQUIEM FOR A LEGACY
What do we mean by “Legacy?”

- “Mature information systems grow old disgracefully…”¹
- Software architectures (planned or “accidental”) are conceived to address a particular need at a moment in time
  - May or may not be built to anticipate new features
  - Probably will not accommodate changes in business focus
- Legacy software has reached a point in its lifecycle at which change becomes difficult
  - Cannot meet new business needs in a timely manner
  - Maintenance tasks become more costly and more difficult to staff

¹Legacy System Anti-Patterns and a Pattern-Oriented Migration Response by Anthony Lauder and Stuart Kent (see References)
Don’t forget the good parts

• A legacy platform exists because it addresses needs
  – If it has existed for a long time, it probably has served its function very well
  – Customers are not likely to allow a shut-down of current systems to move to something new with fewer features

• Implications:
  – Legacy migration must be staged
  – We must find ways to preserve the “good parts” while addressing current limitations
# Legacy Anti-patterns and Fixes

<table>
<thead>
<tr>
<th>Anti-pattern</th>
<th>Fix Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ball and Chain</td>
<td>Portability Adapter</td>
</tr>
<tr>
<td>Tower of Babel</td>
<td>Babel Fish</td>
</tr>
<tr>
<td>Monolithicity</td>
<td>Virtual Componentization</td>
</tr>
<tr>
<td>Gold in Them Thar Hills</td>
<td>Gold Mining</td>
</tr>
<tr>
<td>Tight Coupling</td>
<td>Implicit Invocation</td>
</tr>
<tr>
<td>Code Pollution</td>
<td>Protocol Reflection</td>
</tr>
</tbody>
</table>

**Source:** liveops
Anti-pattern: “Ball and Chain”

• Summary: Software is “chained” to legacy operating systems through use of OS-specific features
  – Unix: System V (vs. Posix) Message Queues
  – Windows: Hooks
  – OS X: Cocoa APIs
  – All: Shell calls to invoke command-line processes

• Portability is limited by original system’s scope
  – Easiest: CentOS 4 → CentOS 5
  – Harder: CentOS 5 → OS X
  – Hardest: CentOS 5 → Windows Server 2008
Fix Pattern: Portability Adapter

- Create or select a single portability layer technology
  - JVM, .NET Common Language Runtime, Posix, LLVM
  - Purpose-built Portability Layer using OSGi
- Create a service representative of OS-specific features for consumption by new applications and services
  - Allow existing apps and services to continue native feature use
  - New apps and services must use features as represented in portability layer
- Pattern application results in sweeping system changes long-term, but leaves existing code intact for immediate future
Anti-pattern: Tower of Babel

• Summary: System components were developed in a variety of programming languages
  – Resource constraints; must use skills available
  – Attempt to attract new talent with new language adoption
  – “Polyglot” language philosophy; language diversity for its own sake
• Over time, organization cannot maintain code in a large number of languages
  – Original developers may have moved on (new project, new company, etc.)
  – Special-purpose implementations in exotic or less-popular languages (e.g., OCaml) become marginalized
  – Common language runtime mitigates the problem to some extent, but does not solve it
• Interoperation between languages and runtimes is difficult, preventing rapid innovation
Fix Pattern: Babel Fish

• Select and integrate a language interoperability technology that creates a bridge between multiple languages and runtimes
  – Cross-language libraries: Thrift, Avro, Pbuf
    • Revisiting the “CORBA” theme
    • Prone to point-to-point IDL definitions
  – SOA: Wrap all services with SOAP and present WSDLs
    • Maintain internal services registry
  – EDA: present all modules with access to a common event bus
    • Asynchronous event patterns
    • Well-defined event stream
    • Complete service decoupling / separation of concerns
Anti-pattern: “Monolithicity”

• Summary: Services are embedded in large, monolithic systems
  – complex custom DB apps with heavy use of stored procedures
  – older ERP systems
  – proprietary voice applications (e.g., IBM WebSphere Voice)
• Not easily componentized, so reuse of discrete features is more difficult
• Developers may “specialize” in these systems
  – System lifetime is prolonged by their experience, but this becomes niche knowledge
  – Risk for the business and for individual careers
Fix Pattern: Virtual Componentization

• Create a model of ideal components for each monolithic system
  – Implement idealized components using a Façade above monolithic system
    • “Fake it until you make it”
  – Expose APIs for new application development

• Gradually replace Façade components with new implementations
  – Process should be invisible to newer applications and services, who rely solely on the Façades
Anti-pattern: “Gold in them thar Hills”

• Summary: Source code is the only representation of domain expertise, and tends to be scattered rather than centralized
  – e.g. internal service for creating a new user may have embedded rules about user defaults, permissions, etc.
  – Accumulation of years of learning about the problem space
  – Downside of “self-documenting code”: poorly written/documented code is worse than a badly written requirements document in this case

• Major cause of legacy system petrification
  – Business rules become fragile as developers fear changes that may have broader effects than intended
  – Original requirements may be lost in the noise introduced by years of modifications and changes in ownership
Fix Pattern: Gold Mining

• Unearth requirements and create formal software contracts representing these for a development audience

• Many paths to discovery
  – One method: lots and lots of meetings (aka. “workshops”) to discover and record existing requirements
  – A better method: define contract structure, let domain experts fill it in, and review as a team

• Contracts can take many forms
  – EDA: event format and sequence definitions
  – SOA: WSDLs capturing detailed type information and message definitions
Anti-pattern: Tight Coupling

• Summary: Elements of a legacy system that directly invoke one another tend to become entangled over time
  – Deeper meaning built into method parameters than original intent
  – Invocation may depend upon external preconditions or side effects
  – If changes are not synchronized, breakage will result, so teams must move more slowly
Fix Pattern: Implicit Invocation

• Remove knowledge of external implementations from system components
  – Move to a model of brokered interaction
  – Components express interest in events without knowledge of the remote implementation

• Can accomplish this with:
  – Disciplined SOA, in which a registry of components provides only component binding
  – Brokered, asynchronous messaging, providing a complete separation of concerns
Anti-pattern: Code Pollution

• Summary: Event-driven systems build up code to deal with special event sequence cases
  – Particularly relevant where Implicit Invocation is applied
• Subsystem guard code builds up over time, but protocol rules will not be enforced elsewhere
• Maintenance complexity increases with every new special case
Fix Pattern: Protocol Reflection

- Explicitly define event protocol, then build a shared state machine to deal with special cases
  - GoF State pattern is a good fit
- State machine is deployed for all services that need to support the event protocol
  - Changes to the protocol imply changes to the shared state machine
CHAPTER 3: NO COUNTRY FOR OLD SOFTWARE
Goal: Increase Agility/Programmability

Need: accelerate feature growth with minimal impact on core development resources

Approach:
• Create an abstraction through which the platform’s services may be accessed
• Expose clean, consistent, functionally complete API
Goal: Simplify/Consolidate the Platform

Need: rationalize the applications and base services of the platform, refactoring and merging to limit divergent technologies

Approach:
• Constrain new application development to the API
• Constrain new base services development to virtual components in a portability layer
• Explore technology alternatives for older base services that are difficult to maintain
## Software Layers

<table>
<thead>
<tr>
<th>Layer</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LiveOps, Customer, and Partner Applications</td>
<td>Contact center functionality provided using APIs and other developer and authoring tools</td>
</tr>
<tr>
<td>REST APIs, Scripting, and Interaction Workflow/Engine</td>
<td>Varying methods of interacting with the Platform Abstraction</td>
</tr>
<tr>
<td>Business Objects (Virtual Componentization)</td>
<td>An ideal social contact center model that removes much of the complexity from Base Services</td>
</tr>
<tr>
<td>Base Services (Portability Adapter)</td>
<td>All basic functions required by social contact centers, including communications and data services</td>
</tr>
</tbody>
</table>
Software Layers

- **LiveOps, Customer, and Partner Applications**
- **REST APIs, Scripting, and Interaction Workflow/Engine**
- **Business Objects (Virtual Componentization)**
- **Base Services (Portability Adapter)**

**Conclusion:** we needed to use an OSGi container for modularity, consistency, and operational repeatability.
Why Apply OSGi to Legacy Migration?

- Directly applicable to addressing legacy anti-patterns
  - e.g., Event Admin implementation to support Indirect Invocation, Declarative Services + Remote Service Admin interfaces for Virtual Componentization
- Separation of concerns built-in
  - Modularity provides barrier that fosters code independence
- Many compendium services are analogous to typical enterprise integration patterns (by design)
  - Event Admin => brokered messaging
  - Config Admin => service configuration access
  - User Admin => AAA and user account management
  - Monitor Admin => monitoring and alerting services for Net Ops
- Developers encouraged by the model to do things the “right” way going forward
  - Easier to reuse
- Most do-it-yourself OSGi options are open source
Previous Work

• We had decided we needed a container for new layers
  – Our first API implementation looked very much like what we needed
Previous Container: Glassfish

• Positives
  – Rich in features, stable
  – High performance HTTP (Grizzly)
  – Rich admin UI
  – Based on Felix
    • HK2 used for web stack, but OSGi at core
Previous Container: Glassfish

- **Negatives**
  - Default deployment profile cluttered
    - Don’t need most of it, but once removed, why use Glassfish?
  - Complex deployment
    - Wanted more control over deployment images
  - Oracle acquisition
    - Marketing messaging reassuring to some on team, but not all
  - Occasional WAB deployment quirks
# OSGi Container Options Explored

<table>
<thead>
<tr>
<th>Container</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
</table>
| Apache Felix Standalone | • Lean, no extras  
• Familiar to us  
• Heavily used | • Need to build management/monitoring tools |
| Glassfish (Based on Felix) | • What we currently deploy  
• Includes Sailfin | • Heavyweight; contains much we do not need |
| JBoss OSGi           | • Includes Mobicents | • OSGi runs above base JBoss services |
| Equinox              | • Heavily used | • Major platform change |
| Knoplerfish          |                                                                       | • Major platform change  
• Not as widely used as others |
| Karaf                | • Makes Felix more manageable  
• Maven support for dependencies | • May be locked into Felix version |
| Concierge            | • Very lightweight | • Major platform change  
• Only OSGi 3 impl. |
| Paremus Nimble       |                                                                       | • Major platform change  
• Not OSS |
| Virgo                | • Multi-bundle model  
• Jetty support | • Same issue as Equinox |
# OSGi Container Options Explored

<table>
<thead>
<tr>
<th>Container</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache Felix Standalone</td>
<td>• Lean, no extras</td>
<td>• Need to build management/monitoring tools</td>
</tr>
<tr>
<td></td>
<td>• Familiar to us</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Heavily used</td>
<td></td>
</tr>
<tr>
<td>Glassfish (Based on Felix)</td>
<td>• What we currently deploy</td>
<td>• Heavyweight; contains much we do not need</td>
</tr>
<tr>
<td></td>
<td>• Includes Sailfin</td>
<td></td>
</tr>
<tr>
<td>JBoss OSGi</td>
<td>• Includes Mobicents</td>
<td>• OSGi runs above base JBoss services</td>
</tr>
<tr>
<td>Equinox</td>
<td>• Heavily used</td>
<td>• Major platform change</td>
</tr>
<tr>
<td>Knoplerfish</td>
<td></td>
<td>• Major platform change</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Not as widely used as others</td>
</tr>
<tr>
<td>Karaf</td>
<td>• Makes Felix more manageable</td>
<td>• May be locked into Felix version</td>
</tr>
<tr>
<td></td>
<td>• Maven support for dependencies</td>
<td></td>
</tr>
<tr>
<td>Concierge</td>
<td>• Very lightweight</td>
<td>• Major platform change</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Only OSGi 3 impl.</td>
</tr>
<tr>
<td>Paremus Nimble</td>
<td></td>
<td>• Major platform change</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Not OSS</td>
</tr>
<tr>
<td>Virgo</td>
<td>• Multi-bundle model</td>
<td>• Same issue as Equinox</td>
</tr>
<tr>
<td></td>
<td>• Jetty support</td>
<td></td>
</tr>
</tbody>
</table>
Decision: Create a LiveOps-Specific App Server

• Benefits
  – Retain control over deployment image
    • Nothing extra, ability to update core bundles with few interaction worries
  – Accelerate development through the use of consistent tools and technologies, safe code sharing, and a layered, extensible platform
  – Wrap all existing LiveOps services (voice, messaging, data, monitoring, etc.) and provide them to LO developers in a consistent way
  – Deploy a single image in LO data centers that Ops can “flavor” in different ways (API server, SIP proxy, call manager…)
  – Expose a consistent social contact center model to customers, partners, and internal LiveOps developers for use in new applications and services

• Base Technologies
  – Daemontools
  – Java 6 JVM
  – Felix 4 / OSGi Core+Compendium 4.3
Winterfell Framework

LiveOps App Server Concept

- Data Access
- Monitor
- Log
- Event
- Config
- Telephony
- Security
- Scheduling
- Multichannel Adapter
- Distributed Memory
- Multichannel
- ResourceID
- Stats Collection
- Remote admin
- Server profile

Felix 4.0.3 / OSGi 4.3

JVM (Java 6)

Daemontools / wfcrl
Migration Stage 1: API

New Application 1

Old Application 1

Old Application 2

Svc. 1

Svc. 2

Svc. 3

Svc. 4

DB
Migration Stage 2: Service Ports

New Application 1

Old Application 1

Old Application 2

Svc. 1

Svc. 2

Svc. 3

Svc. 4
Migration Stage 3: App Ports

New Application 1

New Application 2

Old Application 1

Old Application 2

LiveOps App Server Concept

Svc. 1
Svc. 2
Svc. 3
Svc. 4

DB

0:30:44
Winterfell Tools

LoCo – LiveOps Code Gen
- JSON IDL generates:
  - JavaScript lib and Tests
  - Resource Bundle Skeleton
  - JAX-RS uses raw IDL

Stark Deployment
- Built image Arya images stored on internal github
- New server gets all bits and activates one Arya service per container

Raven Event Bus
- All events contractually defined
- Send/receive using Event Admin + RabbitMQ

Arya Services
- All platform server types defined by JSON deployment file
- All code deployed; service type selected at runtime

Gnostic Monitor
- Monitor Admin implementation sends events from all services to monitoring pipeline
LoCo Framework: Generation

JSON IDL

```json
{
    "basePath" : "/v2",
    "idlVersion" : "1",
    "nameSpace" : "com/liveops/api/system",
    "resources" : [
        {
            "path" : "about",
            "Description" : "Responds with general information about the API",
            "apis" : [
                {
                    "path" : null,
                    "operations" : [
                        {
                            "verb" : "get",
                            "parameters" : [ ],
                            "outputType" : "about",
                            "httpMethod" : "GET"
                        }
                    ]
                }
            ]
        }
    ]
}
```

JavaScript library

Java Bean and Implementation classes

Test HTML + JavaScript + deployable site project
Java Classes
Generated

Resource Bean

```java
/**
 * Liveops Java API stub template
 */
package com.liveops.api.system;

import java.util.*;
import org.codehaus.jackson.annotate.JsonSerialize;
import com.liveops.integration.platform_api.*;
import com.liveops.integration.platform_api.framework.resourceid.*;

@JsonSerialize(include = JsonSerialize.Inclusion.NON_NULL)
public class AboutModel {
    private String version;
    public String getVersion() {
        return this.version;
    }
    public void setVersion(String version) {
        this.version = version;
    }
    private String liveopsTime;
    public String getLiveopsTime() {
        return this.liveopsTime;
    }
    public void setLiveopsTime(String liveopsTime) {
        this.liveopsTime = liveopsTime;
    }
}
```

Resource Implementation Shell

```java
/**
 * Liveops Java API stub template
 */
package com.liveops.api.system;

import java.util.*;
import org.apache.service.log.LogService;
import com.liveops.integration.platform_api.*;
import com.liveops.integration.platform_api.framework.resourceid.*;

public class AboutService extends ApiService {
    @JsonSerialize(include = JsonSerialize.Inclusion.NON_NULL)
    public class AboutModel {
        private String version;
        public String getVersion() {
            return this.version;
        }
        public void setVersion(String version) {
            this.version = version;
        }
        private String liveopsTime;
        public String getLiveopsTime() {
            return this.liveopsTime;
        }
        public void setLiveopsTime(String liveopsTime) {
            this.liveopsTime = liveopsTime;
        }
    }
    protected LogService logService;
    protected void bindLogService(LogService logService) {
        this.logService = logService;
    }
    protected void unbindLogService(LogService logService) {
        this.logService = null;
    }
    protected void start() {
        logService.log(LogService.LEVEL_INFO, "**** Starting AboutService");
    }
    protected void stop() {
        logService.log(LogService.LEVEL_INFO, "**** Stopping AboutService");
    }
```
Runtime API Operation

- REST API endpoint validates requests using JSON IDL
- API Svc. Manager provides binding from URL to Resource Impl.
- Resource Impl. Builds and returns resource bean
- Resource ID Service translates inner to API IDs
WHAT APPS MAY COME
Future Projects

• Winterfell open source
• Raven on ZeroMQ
• Winterfell on EC2
• OSGi 5
REFERENCES
Links

- http://csis.pace.edu/~marchese/CS775/Proj1/legacyinfosys_directions.pdf
- http://kar.kent.ac.uk/21917/1/legacy_system_anti_patterns_lauder.pdf