**OSGi “in Depth” and “in Action”**

**Суммарный дайджест обоих книг от krocodl.**

**Степень сжатия примерно 8.7**

**Правки и замечания приветствуются.**

**Некоммерческое использование не возбраняется в любой форме**

Оглавление

[Введение 5](#_Toc342742133)

[Терминология 5](#_Toc342742134)

[История 5](#_Toc342742135)

[Общая структура 5](#_Toc342742136)

[Проблемы и решения 6](#_Toc342742137)

[Аналоги 7](#_Toc342742138)

[Состояние рынка 8](#_Toc342742139)

[Архитектура и шаблоны 9](#_Toc342742140)

[Послойная архитектура платформы 9](#_Toc342742141)

[Приложения на основе OSGi 9](#_Toc342742142)

[Потребление сервиса 10](#_Toc342742143)

[Поставщики сервиса 10](#_Toc342742144)

[Точки расширения 11](#_Toc342742145)

[Нотификация 11](#_Toc342742146)

[Расширитель 12](#_Toc342742147)

[Контейнер приложения 13](#_Toc342742148)

[Bundle 14](#_Toc342742149)

[Структура 14](#_Toc342742150)

[Импорт пакетов 15](#_Toc342742151)

[Импорт bundle 16](#_Toc342742152)

[Динамический импорт 18](#_Toc342742153)

[Опциональный импорт 18](#_Toc342742154)

[Фрагментирование bundle 18](#_Toc342742155)

[API исследования 19](#_Toc342742156)

[Программная активация 20](#_Toc342742157)

[Контекст выполнения 21](#_Toc342742158)

[Версионность 22](#_Toc342742159)

[Вычисление зависимостей 23](#_Toc342742160)

[Загрузка классов 24](#_Toc342742161)

[Доступ к ресурсам 27](#_Toc342742162)

[Нативный код 27](#_Toc342742163)

[Сервисы 29](#_Toc342742164)

[Общая архитектура 29](#_Toc342742165)

[Реализация сервиса 31](#_Toc342742166)

[Архитектура клиента 32](#_Toc342742167)

[Реализация клиента 33](#_Toc342742168)

[Фабрика сервисов 35](#_Toc342742169)

[Стандартные сервисы 36](#_Toc342742170)

[Расширенные возможность 36](#_Toc342742171)

[Конфигурирование 37](#_Toc342742172)

[Архитектура 37](#_Toc342742173)

[Получатель конфигурации 37](#_Toc342742174)

[Конфигуратор 38](#_Toc342742175)

[Метаданные 40](#_Toc342742176)

[Проблемы 41](#_Toc342742177)

[Логирование 41](#_Toc342742178)

[Обмен событиями 43](#_Toc342742179)

[Архитектура 43](#_Toc342742180)

[Получение сообщений 43](#_Toc342742181)

[Публикация сообщений 44](#_Toc342742182)

[Недостатки 45](#_Toc342742183)

[Стандартные события OSGi 45](#_Toc342742184)

[Персистентность 47](#_Toc342742185)

[Read-Only свойства 47](#_Toc342742186)

[Локальная файловая область bundle 47](#_Toc342742187)

[Preferences 47](#_Toc342742188)

[JDBC 48](#_Toc342742189)

[JPA 49](#_Toc342742190)

[Транзакции 51](#_Toc342742191)

[JNDI 53](#_Toc342742192)

[Концепция 53](#_Toc342742193)

[Элементарные операции 53](#_Toc342742194)

[Инициализация контекста 55](#_Toc342742195)

[Преобразования объектов 56](#_Toc342742196)

[Пространства имен 58](#_Toc342742197)

[Развертывание приложения 60](#_Toc342742198)

[Загрузка и классы 60](#_Toc342742199)

[Доступ через JNDI 61](#_Toc342742200)

[Уровни исполнения 61](#_Toc342742201)

[Опции запуска bundle 63](#_Toc342742202)

[Развертывание и обновление 64](#_Toc342742203)

[OSGi bundle repository 65](#_Toc342742204)

[Deployment Admin 66](#_Toc342742205)

[JMX 69](#_Toc342742206)

[Основы 69](#_Toc342742207)

[BundleStateMBean (osgi.core:type=bundleState,version=1.5) 70](#_Toc342742208)

[ServiceStateMBean (osgi.core:type=ServiceState,version=1.5) 70](#_Toc342742209)

[PackageStateMBean (osgi.core:type=PackageState,version=1.5) 70](#_Toc342742210)

[FrameworkMBean(osgi.core:type=Framework,version=1.5) 71](#_Toc342742211)

[ConfigurationAdminMBean (sgi.compendium:service=cm,version=1.3) 72](#_Toc342742212)

[Оповещения 72](#_Toc342742213)

[Наименования 73](#_Toc342742214)

[Формирование приложения 74](#_Toc342742215)

[Общая стратегия действий 74](#_Toc342742216)

[Идентификация bundle 74](#_Toc342742217)

[Версионность bundle 74](#_Toc342742218)

[Экспорт пакетов 75](#_Toc342742219)

[Импорт пакетов 76](#_Toc342742220)

[Embedding 76](#_Toc342742221)

[Поддержка ЖЦ библиотек 77](#_Toc342742222)

[Преобразование приложения в целом 77](#_Toc342742223)

[Использование сервисов 78](#_Toc342742224)

[Тестирование 80](#_Toc342742225)

[Unit тесты 80](#_Toc342742226)

[Bundle тесты 81](#_Toc342742227)

[Интеграционные тесты 82](#_Toc342742228)

[Отладка 84](#_Toc342742229)

[Диагностика ошибок 84](#_Toc342742230)

[Проблемы с классами 84](#_Toc342742231)

[Утечки памяти 86](#_Toc342742232)

[Работа с компонентами внутри bundles 88](#_Toc342742233)

[Основы 88](#_Toc342742234)

[Declarative Services 90](#_Toc342742235)

[Blueprint 93](#_Toc342742236)

[Apache Felix IPOJO 99](#_Toc342742237)

[Сводные характеристики 103](#_Toc342742238)

[Встраивание контейнера 104](#_Toc342742239)

[Архитектура 104](#_Toc342742240)

[Основные операции 105](#_Toc342742241)

[Встраивание OSGi контейнера внутрь приложения 106](#_Toc342742242)

[Безопасность 108](#_Toc342742243)

[Общие места 108](#_Toc342742244)

[Java безопасность 108](#_Toc342742245)

[OSGi безопасность 109](#_Toc342742246)

[ConditionalPermissionAdmin 110](#_Toc342742247)

[Кастомные условия 113](#_Toc342742248)

[Включение безопасности 114](#_Toc342742249)

[Веб приложения 115](#_Toc342742250)

[Работа со статическими ресурсами 115](#_Toc342742251)

[Работа с сервлетами 115](#_Toc342742252)

[Работа с WAB 116](#_Toc342742253)

[Распределенные сервисы 117](#_Toc342742254)

[Основные принципы 117](#_Toc342742255)

[Провайдеры 118](#_Toc342742256)

[Облако 120](#_Toc342742257)

[Сводка OSGi сервисов 122](#_Toc342742258)

[Core сервисы 122](#_Toc342742259)

[Compendium сервисы 122](#_Toc342742260)

[Enterprise сервисы 123](#_Toc342742261)

# Введение

## Терминология

* Платформа разработки – набор библиотек и утилит для разработки компонент + runtime окружение, в котором исполняются разработанные компоненты
  + JSE = JDK + JRE
  + OSGi = JSE + OSGi application framework
* Application framework – предоставляет базовую образующую структуру для разработки прикладных приложений
  + Гарантия того, что основа качественно спроектирована и реализована, тщательно протестирована, помогает снизить число дефектов, увеличивает runtime надежность
  + Возможность использовать одну основу для разных приложений, способствует продуктивности разработки, увеличивает design time производительность

## История

* Начало март 1999, консорциум компаний, большей части телекоммуникационных, для разработки универсальной интеграционной платформы для обеспечения возможности взаимодействия приложений и сервисов.
  + Платформа изначально мыслилась как крайне легковесная для развертывания приложений на устройствах с малым количеством памяти
  + Приложения должны были динамически развертываться на устройствах и присоединяться к различным приложениям / сервисам, уже развернутым на устройствах.
  + Из-за нехватки ресурсов они должны тесно взаимодействовать друг с другом в крайне легковесной манере
* Архитектура приложений на основе subj должна была базироваться на идее отделенных друг от друга компонентов и динамически подключаемых сервисов:
  + фактически SOA но на уровне виртуальной машины
  + приложения должны иметь дело с большим количеством сервисных API за которыми располагаются всевозможные реализации предоставляемых сервисов.
  + Реализации сервисов могут динамически меняться во время жизни приложения.
  + Что-то подобное есть в J2EE, но решения тяжелы и не могут быть портированы в окружение с недостатком ресурсов

## Общая структура

* **OSGi framework**
  + Основывается на JVM, обеспечивающей совместимость с различными аппаратными и программными окружениями
  + **Service management system** - подсистемы управления сервисами – для регистрации сервисов, предоставления их OSGi модулям (bundles, пакеты), отделение провайдеров сервисов от потребителей
  + **Dynamic Java modules system** – динамическая загрузка и выгрузка OSGi пакетов
* **OSGi service platform**
  + Включает в себя ряд сервисов общего назначения, можно их представлять как native osgi приложения
  + Инфраструктурные – связывание пакетов, запуск среды
  + Горизонтальные – логирование и конфигурирование
  + Протокольные – HTTP, RMI
* **Enterprise OSGi**
  + Управление и конфигурирование – Configuration Admin service , JMX management service , Metatype Admin service
  + Доступ к данным – JDBC, JPA, JTA
  + Построение распределенных систем – Remote Service, SCA Configuration

## Проблемы и решения

* Проблемы
  + Кроме чисто объектных языковых средств Java не предоставляет поддержки для построения модульных систем
    - JSE не поддерживает модули крупнее классов и пакетов, ориентирована на низкоуровневую ООП инкапсуляцию, а не на разделение системы на логические части.
      * Любые зависимости, выходящие за пределы пакетов, должны быть определены как public, что автоматически делает их доступными всем.
      * Большей частью пакеты используются как пространства имен для различения классов, а не для разделения системы на отдельные части. Семантика «вложения» пакетов друг друга никак не коррелирует с изоляций, вложенные друг в друга пакеты эквиваленты не-вложенным
    - Проблемы с загрузчиками классов и classpath
      * Не поддерживаются понятия версии кода и зависимостей
      * не проверяется набор загружаемых классов на корректность зависимостей, а также на целостность пространства классов
      * Если более чем один jar в classpath предоставляет набор классов, то результат непредсказуем (class path hell)
      * Не обращается внимание на логическое разделение системы на компоненты, все классы в итоге сбрасывается в одну кучу в процессе загрузки.
      * Не обращается внимание на версии загружаемых классов, грузятся первые попавшиеся
      * Из-за on-demand характера операций многие из проблем выявляются не сразу а по мере функционального тестирования
    - JSE не имеет встроенного механизма реестра сервисов
    - JSE не имеет механизма выгрузки однажды загруженных классов
  + Все приложения со временем становятся больше и сложнее, их становится сложнее поддерживать, сложность растет нелинейно, иногда экспоненциально. Приложение становится сложнее расширять без постепенного разрушения
  + Существующие платформы
    - Проблемы с совместимостью между отдельными поставщиками сервисов
    - Платформы становятся огромными коллекциями средств и API, предназначенными для поддержки всего чего угодно. Сложны для освоения и использования. ЖЦ разработки становится очень медленным.
    - Как следствие даже самое простое приложение должно учитывать необходимость инициализации всей огромной платформы со всеми ее расширениями, что занимает много времени. Нельзя отобрать для запуска только нужное. Каждый сервис в платформе имеет единственную реализацию, дублирование невозможно
    - Платформы берутся только в стиле «все или ничего». Нельзя ограничить потребляемые сервисы или взять одни сервисы от одного поставщика, а другие от другого. Если сервиса в платформе нет, то надо ждать пока поставщик платформы его добавит, прикрутить стороннюю реализацию очень трудно
* Традиционно проблема сложности приложений решается за счет разделения всего кода на отдельные изолированные куски. Но по мере роста команды от 1 до 10 человек и роста объема кода от тысяч до десятков тысяч строк сложность работы с кодом и сотрудниками растет, и даже не линейно
  + Изменение в реализации одного компонента может вызвать проблемы по всему приложению, зачастую в казалось бы несвязанных местах. Ни один из разработчиков не в состоянии предсказать последствия изменения простейшего интерфейса
  + В системе множатся версии одних и тех же функциональных кусков, которые опасно поменять из-за непредсказуемых последствий
  + Исправление багов становится все сложнее и сложнее, так как любое вносимое без согласования с авторами первоначального кода изменение становится потенциальным источником непредсказуемых проблем.
  + Enterprise приложения добавляет трудности: процессы и логика обработки сложнее, необходимость устойчивости, управляемости, распределенности
* OSGi позволяет эффективно разделять приложение на отдельные модули, каждый из которых имеет строго ограниченный интерфейс, на который ссылаются все потребляющие его клиенты. Данный контракт определен декларативно, не средствами Java, что позволяет эффективно использовать эти данные всевозможными сторонними средствами.
  + Компоненты-расширения application framework обычно играют роль провайдеров сервисов, которые в свою очередь потребляются фрейморком. Интерфейсы таких сервисов – контракты потенциальных точек расширения поведения платформы
  + Иногда компоненты также играю роль потребителей сервисов, в этом случае платформа предоставляет и формальный контракт и реализацию сервисов
  + В обоих случаях прикладные компоненты отделены от платформы в целом, что снижает уязвимость к ошибкам системы в целом
* OSGi предоставляет только общую структуру для размещения сервисов, которую можно гибко настроить под каждое конкретное приложение.
  + Можно набить только нужными сервисами от наиболее передовых поставщиков. Минимизация необходимой приложению функциональности платформы
  + Переносимость не только между платформами, но и между различными сервисами внутри одной и той же платформы
  + Можно сочетать сервисы разных поставщиков и даже разные версии одного сервиса одного поставщика, что обычно является непреодолимой проблемой

## Аналоги

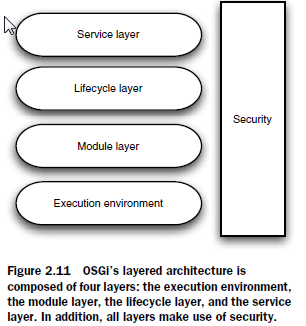
* EJB – возможно наиболее близкая технология. Поддерживает модульность более ориентирована на EE фичи: персистентность, транзакции и безопасность. В последнее время тенденция выстраивания JEE серверов поверх OSGi
* Jini – концепция распределенных по сети, а не в одной VM, систем, понятия провайдера / потребителя сервиса, реестра серверов, больше сфокусирован на распределении, чем на модульности, загрузка классов полностью на JSE концепции
* NetBeans – тоже компоненты, сервисы и реестр, но ориентирован специально на IDE
* JMX – вообще-то для мониторинга, но абстрактная компонентная модель и механизм динамической загрузки классов сделали его основной для JBoss (до 5-ой версии). Сейчас родственная технология для администрирования OSGi
* IOC контейнеры – вместо использование реестра сервисов предлагается DI. Впрочем и в OSGi есть похожие механизмы – Blueprint. В последнее время идет сращивание
* Java Business Integration – средство создания SOA платформы и В2В решений, плагинная архитектура, инсталляция, жизненный цикл, формат пакетировки. По сравнению с OSGi сильно упрощение, только для интеграции в основной фреймворк
* JSR 277 – Java Module System, по идее аналогичная поддержка модульности, проиграл JSR 291, который базировался на OSGi R4 спецификации
* JSR 294 – еще одна модульность, но средствами языка, а не описателей, концепция суперпакетов, заглохла

## Состояние рынка

* **Реализации платформы**
  + Eclipse Equinox: <http://www.eclipse.org/equinox/>
  + Apache Felix: <http://felix.apache.org/site/index.html>
  + Knopflerfish: <http://www.knopflerfish.org/>
* **Репозитории компонент**
  + Felix Repository: <http://felix.apache.org/obr/releases.xml>
  + Knopflerfish Repository: <http://www.knopflerfish.org/repo/repository.xml>
  + SpringSource Repository: http://ebr.springsource.com/repository/app/
  + OSGi Alliance Repository: http://www.osgi.org/Repository/HomePage
  + Oscar Bundle Repository: http://oscar-osgi.sourceforge.net/
  + Eclipse Repository: http://eclipse.org/equinox/bundles/
* **Enterprise решения**
  + Eclipse Gemini: <http://www.eclipse.org/proposals/gemini/>
  + Apache Aries: <http://incubator.apache.org/aries/>
* **Сервера приложений и системы основанные на OSGi**
  + IBM WebSphere Application Server
  + Oracle (formerly Sun) GlassFish Application Server
  + Eclipse Virgo (SpringSource dm Server)
  + JBoss Application Server
  + Apache Camel
  + Apache Sling
  + Apache ServiceMix
  + Apache Karaf
  + Apache Geronimo
  + ObjectWeb’s JOnAS

# Архитектура и шаблоны

## Послойная архитектура платформы



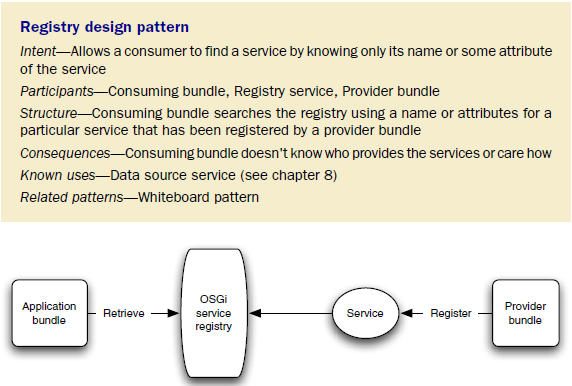
* В основе слой абстракции над JVM.
  + Необходим так как бывают разные JVM (ME, SE, EE), разные версии на разных аппаратных платформах.
  + Bundle могут специфицировать свой требования к JVM за счет заголовка манифеста **RequiredExecutionEnvironment**.
* Слой модулей – импорт / экспорт пакетов, множественные загрузчики классов и делегирование ими ответственности друг другу
* Жизненный цикл модулей – API поддержки операций install / uninstall / start / stop / update. НЕ предоставляет какой-либо стандартный shell управления модулями (консоль, GUI, XML конфигурационный файл, файловая система), вместо этого поставляет только API и возможность всем желающим написать оболочку по своему вкусу.
* Сервисы – регистрация и де-регистрация сервисов с помощью рееста, платформенные сервисы
  + Core сервисы, обязательные для поставки в составе платформы
  + Compendium – вспомогательные сервисы, тесно связанные с инфраструктурой. По умолчанию не устанавливаются, так как снижают уровень безопасности системы.
* Через все слои – архитектура безопасности, предоставляемая Java платформой

## Приложения на основе OSGi

* Модульность
  + Логическая группировка (границы видимости классов) и физическая группировка (совместное развертывание). Традиционно языки ориентируются на первое, а операционные системы на второе. Java приходится иметь дело и с тем и с тем, так как она и язык и платформа
  + Модули скрывают внутри себя классы, позволяя разработчику выражать логические взаимосвязи между ними или концепции
  + Классы – изоляция на микро уровне, думаем в терминах функциональности. Модули – изоляция на макро уровне, думаем в терминах частей приложения
  + Вместо монолита, где все зависит от всего, разделение плотности зависимостей. Внутри модуля –сильно связанные друг с другом классы – high cohesion, модули между собой связаны слабо – low coupling.
  + Модульность расширяет набор модификаторов доступа Java еще одним – module private visibility, аналог сборки в .NET
* Основной подход к организации архитектуры клиента – максимальное разделение на отдельные модули не содержащие зависимости от классов и интерфейсов платформы (кроме активаторов). Это позволяет тестировать части приложения за счет unit тестов, при необходимости легко мигрировать приложение на другую платформу
* Основной недостаток модульности – увеличивает вероятность, что один класс будет загружен несколькими bundle, а затем эти экземпляры встретятся друг с другом. Проблема вероятностна, так как зависит от порядка загрузки bundle и конкуренции процессов загрузки / старта, с трудом ловится
* Основная проблема реализации – невозможность использования клиентом статических методов для получения тех или иных сервисов. К сожалению, частый подход для получения доступа к инфраструктурным сервисам в SE, например JDBC или JNDI инициализация.
  + Негативные последствия:
    - Во-первых, реализации этих сервисов должны создаваться в пространстве классов клиента, а значит он будет зависеть не только от API, но и от всех возможных реализаций
    - Во-вторых, невозможно будет динамически управлять реализацией незаметно для клиента
    - В-третьих, невозможно будет одновременно предоставлять клиенту несколько реализаций сервиса (или несколько версий одной реализации)
  + Решение
    - Делаем входной сервис-синглетон, которому передаются параметры, например наименование класса реализации
    - Все потенциальные реализации регистрируются как сервисы, указав в свойствах значения параметров, по которому их будут различать
    - Входной сервис по параметрам ищет в реестре реализацию, которую возвращает клиенту
    - Реализация при этом оборачивается в прокси, которая будет следить за динамическим поведением сервиса
* Принцип дизайна компонентов: сделать их максимально специализированными, выбрасывая из каждой все, что непосредственно к ней не относится. Игра, где каждый разработчик, старается передать максимум ответственности другим. Акцент на
  + Минимальное зацепление, позволяющее максимально повторно использовать
  + Больше использовать интерфейсы, чем наследование
  + Явно описание зависимостей
  + Изначальная поддержка возможности множественных реализаций

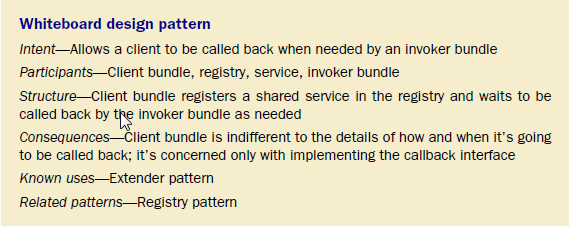
## Потребление сервиса

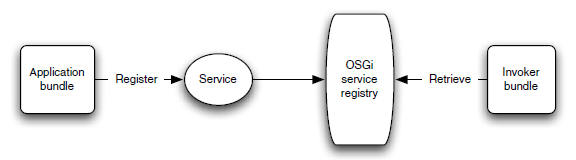
* Push механизм, потребитель активен, его написать достаточно сложно (запрос с фильтром, отслеживание состояния), этим занимаемся сами



## Поставщики сервиса

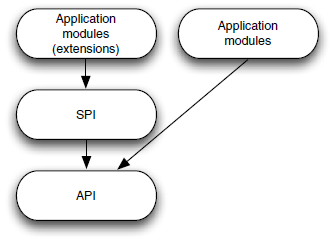
* Pull механизм, поставщик пассивен, его написать просто, это отдается сторонним разработчикам





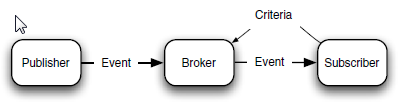
* Сервис как обобщенная форма слушателя, который может получать все сорта запросов, не только события.
* Недостатки
  + слушатель должен зарегистрировать некоторый специальный интерфейс, что делает его зависимым от поставщика. Это снижает повторную используемость
  + Реестр должен быть достаточно масштабируем чтобы поддерживать работу большого числа поставщиков и слушателей

## Точки расширения



* Система, как правило, поддерживает два типа интерфейсов, один для поддержки клиентов (API), другой для поддержки расширений поведения приложения (SPI). Реализации последних ,плагины или подключаемые сервисы, как правило предоставляются вендорами.
* Общее правило: классы SPI используют классы API и никогда наоборот

## Нотификация



* + Компоненты: издатель, брокер и подписчик. Подписчик регистрируется в брокере, указывая критерии отбора тех сообщений, которые ему интересы
  + Брокер

**public interface NotificationBroker {**

**void sendEvent(Object event);**

**int subscribe(String criteria, NotificationSubscriber subscriber);**

**void unsubscribe(int susbcriberId);**

**}**

* + Подписчик

**public interface NotificationSubscriber {**

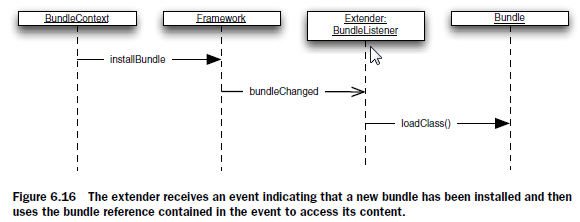
**void onEvent(Object event);**

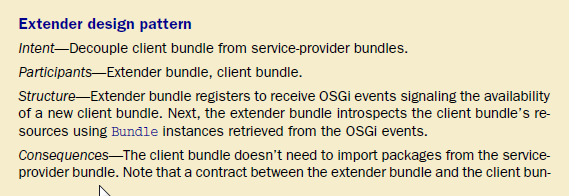
**}**

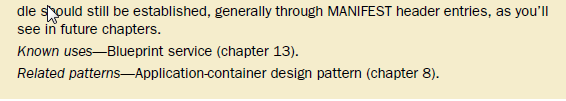
* + Очень важна возможность удаленного подписчика, позволяющего держать брокера как можно ближе к источнику сообщений. Особенно, когда подписчику нужна только малая доля от всех публикующихся сообщений

## Расширитель

* Позволяет bundle, реализующему функционал контейнера, отслеживать загрузку других bundle, получать доступ к их содержимому и быть готовым, на основании этого содержимого, предоставить им какой-либо сервис

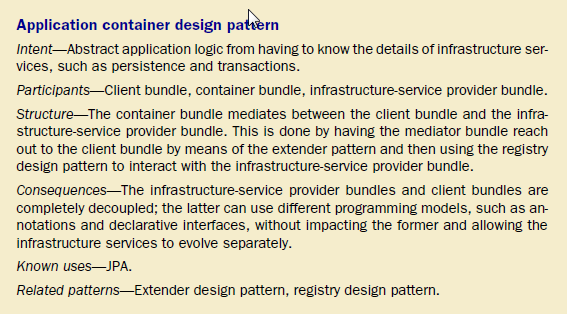






* Примеры:
  + JPA инфраструктура считывает сущности из загружаемых bundle и добавляет их в динамически формируемую конфигурации **EntityManager**.
  + blueprint сервис, позволяющий использовать декларативные определения вместо активаторов
* Контракт между расширяемым и расширителем – наличие в bundle ресурсов определенного типа и структуры, например полей в дескрипторе JAR или XML файла, лежащего по определенному пути
* Расширитель подписывается на события инсталляции новых bundle и исследует их содержимое посредством **Bundle.getHeaders()** или **Bundle.getEntry().** 
  + Также он обрабатывает события их остановки, чтобы вовремя прекратить использование ресурсов, подгружаемых из них.
  + Если прекращение использования требует обращения к используемому классу, то надо обрабатывать не STOPPED, а STOPPING
* Для частичной автоматизации можно использовать наследника **BundleTracker**, который будет самостоятельно подписываться на события, предоставляя пользователю возможность определить операцию обработки содержимого добавленного или удаленного в систему bundle. Использование трекера - хорошая практика, позволяющая возможность сохранять активатор bundle минимальным по объему.

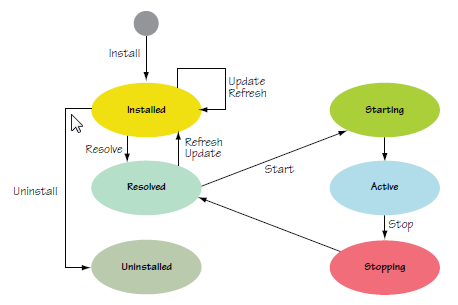
## Контейнер приложения



# Bundle

## Структура

* обычный JAR файл, содержащий тесно связанные между собой классы
  + - какие классы входят в bundle определяется через заголовок **Bundle-ClassPath,** по умолчанию «.»
    - может состоять из нескольких компонент, например .,other-classes/,embedded.jar.
      * Вложенные директории для поиска классов и ресурсов часто используются, например, в war архивах
      * Можно определить библиотеку, приватную для данного bundle. Лучше этим не увлекаться – платформа тратит дополнительное время на распаковку и загрузку
      * классы ищутся в порядке определения компонент
  + bundle можно использовать как обычный JAR если **Bundle-ClassPath** по умолчанию. Java загрузчик не поддерживает загрузку классов из поддиректорий и вложенных JAR
  + JAR можно использовать как bundle, но так как он ничего не экспортирует, то не понятно зачем
* Имеется манифест **META-INF/MANIFEST.MF**
  + Можно было бы использовать метаданные в составе исходного кода, например аннотации, но тогда их будет сложнее менять, кроме того
    - При изменении метаданных будет нужна перекомпиляция
    - Имея jar будет сложно посмотреть значения метаданных вручную, чтобы получить к ним доступ потребуется загружать классы
    - Появятся зависимости времени компиляции от OSGi API
    - Нельзя будет использовать один и тот же код в разных bundle
    - Сложно будет использовать такие JAR в маленьких или старых VM, не поддерживающих аннотации
  + Синтаксис манифеста
    - Пары <имя атрибута>: <значение>, пробел после «:» обязателен
    - значения – любой текст кроме cr и lf
    - строка не может быть длиннее 72 символов, если нужно длиннее продолжаем на следующей строке, начиная ее с пробела
    - между отдельными атрибутами могут быть пустые строки – группы атрибутов, первая группа – главные атрибуты
  + Синтаксис значений OSGi атрибутов
    - Перечень пунктов, разделенных «,»
    - Каждый пункт из предмета и атрибутов или директив - <target>;<param1>=<value1>;<param2>:=<value2>, между ними «;»
      * Директивы специфицированы - известны платформе, для них используется «:=»
      * Атрибуты произвольны, для них используется «=»
    - Можно несколько предметов определить одним набором директив / атрибутов. Предметы при этом тоже через “;”
* Информационные поля манифеста – только для человека и инструментальных средств: Bundle-Name, Bundle-Description, Bundle-DocURL, Bundle-Category, Bundle-Vendor, Bundle-ContactAddress, Bundle-Copyright:
* стандартные поля манифеста
  + **Bundle-ManifestVersion** –стандарт описания, начиная с версии 2 совместимость с R4 требует определения и уникальности символического имени. В принципе, могут встречаться старые bundle, в которых этого нет
  + **Bundle-SymbolicName** – уникальный идентификатор bundle с точностью до версии, обычно используется семантика родственная наименованию пакетов
* если пакет сам активной работы не производит, например, просто поставляет API, то больше ему ничего не нужно, в противном случае необходим активатор, вызываемый платформой, который подключает bundle к услугам платформы (получает сервисы, публикует сервисы, подписывается на события и так далее). Вместо активатора может быть декларативное описание, потребляемое платформенным активатором – Blueprint Service
* Жизненный цикл bundle



* + Переходы в RESOLVED осуществляются неявно, когда bundle запускаются или когда другой bundle путается получить класс из данного. Через API можно явно вызвать процедуру вычисления зависимостей
  + Переход STARTING->ACTIVE всегда неявно, bundle в STARTING от момента запуска до того как Start() завершится. Это включает период ожидания момента, когда уровень исполнения платформы повысится до необходимого. Если start() возбуждает исключение, то состояние откатывается обратно в RESOLVED
  + Состояние STOPPING сохраняется пока метод stop() не завершится
  + В UNINSTALLED состоянии bundle сохраняется столько, сколько необходимо пока что ????
* Идентификация bundle
  + Уникальный идентификатор от производителя – совокупность **Bundle-SymbolicName** и **Bundle- Version**. Гарантируется, что одновременно может быть установлен только один bundle с комбинацией символическое имя / версия. До R4 данных полей не было, а потому понадобился системны номер
  + Платформа в процессе загрузки bundle присваивает ему некоторый системный номер и гарантирует его уникальность в пределах сеанса – **Bundle.getBundleId().** Но он зависит от порядка загрузки bundle, а потому используется в основном администраторами. Существует системный bundle с номером 0 – сам фрейморк, предоставляет расширенное по сравнению с Bundle API.
  + Есть еще наименование ресурса из которого был загружен bundle посредством функции **BundleContext.installBundle()**
    - какой-то URL, его содержание не специфицируется, получается через **Bundle.getLocation()**
    - гарантируется уникальность загруженных bundles по данному параметру

## Импорт пакетов

* Необходимо держать bundle как можно более закрытыми, в идеале он должен содержать все необходимое внутри себя. Если какие-то классы используется только в одном bundle, то можно их (или даже целые jar) сделать приватными для bundle.
  + Кладем jar в директорию, например в lib
  + В манифесте сообщаем, что из директории lib надо загрузить библиотеки посредством заголовка манифеста **Bundle-ClassPath**: .,lib
  + В коде можно получить приватный ресурс от загрузчика классов как **Bundle.getResource(String resourceName).**
* Зависимости bundle
  + Экспортируемые пакеты, то есть ресурсное API:
    - **Export-Package** - перечень пакетов внутри архива через «,»
    - При указании пакета можно использовать атрибуты, которые его описывают
    - Один и тот же пакет и разные версии одного и того же пакета может импортировать множество bundle, ограничений нет
    - Любой экспортируемый пакет это увеличение вероятности конфликта с другим bundle, который возьмется экспортировать тот же самый пакет. ОСОБЕННО, если его пакет будет содержать другой набор классов, нежели ваш
  + Импортируемые пакеты, то есть требуемое API:
    - **Import-Package –** перечень импортируемых пакетов
    - при указании пакета можно использовать фильтр на атрибут
      * можно использовать только «=»
      * например, чтобы отфильтровать пакеты с реализацией API от какого-то производителя или для какой-то платформы
      * наиболее частой применении – фильтр на версии
    - Поддерживается как статический импорт так и импорт через отражение
    - Импорт пакетов не транзитивен, если bundle А импортирует пакеты у Б, а Б у С, то А по умолчанию НЕ импортирует пакеты bundle C
    - При импорте пакета его вложенные пакеты НЕ импортируются, это надо делать явно
  + Импортировать можно только одну версию пакета, а вот экспортировать можно несколько версий.
    - Используется, когда надо ввести новую версию пакета, но обозначить его совместимость со старой версией
    - Конечно, лучше играться при этом диапазоном импорта, но это не всегда возможно
    - Можно за этот счет реализовать концепцию друзей. Один и тот же пакет
      * Экспортируется просто, за исключением какого-то набора классов
      * Экспортируется полностью со свойством, например, fiends=”true” и mandatory:=”friends”.
      * То есть для bundle, которые захотят импортировать данный пакет полностью (например, с какими-то утилитами, облегчающими реализацию публичного API), будет необходимо объявить себя при импорте друзьями через фильтр импорта friends=true
* Импортируются / экспортируются ВСЕ классы в ТОЛЬКО перечисленных пакетах.
  + Соответственно экспортируемые классы должны быть выделены в отдельный пакет
  + Классы суб-пакетов НЕ экспортируются и соответственно НЕ импортируются. Такие пакеты придется перечислять отдельно
  + Можно исключить классы из импорта / экспорта за счет опций **include / exclude**
    - Пример - **Export-Package:** manning.osgi.test; **include:=”**Foo, Bar\*”; **exclude**=”FooImpl”
    - По умолчанию **include=\* и exclude=«»**
    - порядок указания не важен, класс экспортируется только если он содержится в одном листе и не содержится в другом
    - вообще-то конечно лучше разделять импортируемые и приватные классы за счет пакетов, данные опции дают возможность работать с унаследованым кодом
  + можно автоматически исследовать код внутри bundle и генерировать импорты, это делает, например, <http://www.aqute.biz/Code/Bnd>

## Импорт bundle

* Можно импортировать не набор пакетов, а цельный bundle, тесть все его экспортируемые пакеты, за счет использования заголовка манифеста **Require-Bundle**
  + Основной источник возникновения: могут быть пакеты, который содержатся в нескольких bundle либо исторически, либо по мере роста, в связи с необходимостью разными bundle поддерживать разные типы клиентов.
    - Разделять такие пакеты нельзя – придется переделывать клиентов. Клиенты же в процессе разрешения зависимостей будут хвататься за пакет в одном из двух bundle и значит получать вместо совместного содержимого двух частей пакета содержимое только одной его части
    - Если одни и те же классы есть в обоих bundle, которые содержат разделенный пакет, то очередность порядка указания импортируемых bundle имеет значение – классы с одним наименованием, загруженные из разных bundle будут считаться разными
    - Если какой-то bundle хочет импортировать только часть пакета, то он должен ее указать через фильтр:
      * либо bundle-symbolic-name
      * либо через какой-то атрибут, определенный при экспорте в обоих bundle, например split=”part1”
    - защититься от клиентов, которые попробуют все-таки использовать импорт пакета (он даст случайный результат), можно через указание фильтра на атрибут обязательным mandatory=”split”
    - если уже есть старые клиенты, которые хотят получить пакет целиком – надо сделать фасадный bundle, который импортирует эти два bundle с частями разделенного пакета и экспортирует этот пакет БЕЗ обязательного атрибута.
      * Таким образом клиент, не указав фильтр на часть пакета, получит пакет у фасадного bundle.
      * Недостаток: полнота импорта клиентом всех частей пакета остается на совести разработчика фасада, фреймворк не может этого проконтролировать
    - best practice
      * всегда экспортировать части пакетов с обязательными атрибутами, чтобы предотвратить непредусмотренный импорт части
      * для импорта использовать **Require-Bundle** или **Import-Package** с указанием требуемой части за счет обязательного атрибута
      * унаследованных клиентов поддерживать за счет фасадного bundle
  + Поддерживается фильтр на версии импортируемых bundle, может быть опциональный импорт
  + Можно даже сказать, чтобы импортировать еще и те пакеты, которые импортируемый в свою очередь сам импортирует, например **Require-Bundle:** B2; **visibility:=reexport,** лучше так не делать:
    - приводит к возрастания связывания модулей в системе
    - выгрузка или обновление одного bundle в цепочке приведет к серии удалений / обновлений
  + Источник различных проблем, так как зависимость «что импортируем» подменяется на зависимость «кто экспортирует»
    - Split пакетов: BI импортирует В2 (экспортирует пакет Р) который импортирует В3 (экспортирует пакет Р). Классы пакета Р будут загружены двумя загрузчиками классов и это приведет к непредсказуемым проблемам во время эксплуатации
    - В1 определяет зависимость не от пакетов, а от имени В2. Значит не защищен соглашениями о версиях от изменений реализации в В2 и уж тем более в В3
    - Если использовали импорт bundle, а потом решили импортируемый bundle отрефакторить, разделив на несколько, то придется переписывать импорт во всех клиентах старого bundle. С пакетами можно обойтись без переписывания
    - Не возможно будет подменить bundle реализации, предоставив другую версию классов пакета
    - IDE не смогут анализировать зависимости и предлагать подходящие импорты
    - если разные bundle содержат одни и те же классы, то результат поиска классов зависит от порядка импорт и непредсказуем. Особенно, когда какие-то bundle опциональные. Ресурсы могут заслонять друг друга. Примерно тот же classpath hell, что и при традиционной загрузки коллекции jar-ов
    - Так как каждый импортируемый bundle имеет собственный отдельный загрузчик классов, то приватные классы каждого bundle не будут видны классам соседнего bundle. Пакеты все-таки остаются отдельными, хотя и сливаются перед использованием
  + Авторы книги считают, что без данной возможности вполне можно обойтись, так как обычно мы все-таки зависим не от целого bundle, а от какого-то экспортируемого им подмножества
* так как bundle обычно экспортирует много пакетов, а изменяется только некоторые, то определение зависимости от версии bundle может приводить к ошибке, когда клиенты на самом деле используют только какую-то часть на самом деле не изменившейся функциональности, а из-за смены версии bundle в целом теряют возможность ее использовать вообще.
* Допустимо, когда все экспортируемые bundle пакеты тесно связаны между собой и меняются только вместе. Во-первых, это достаточно редкая ситуация, во-вторых, подобные внутренние зависимости экспортируемых пакетов между собой лучше определять через «uses»

## Динамический импорт

* Используется когда мы не знаем какие точно нам классы понадобятся, но знаем в каких пакетах они должны быть. Собираемся грузить классы и создавать экземпляры через отражение. Частая ситуация, например, с драйверами СУБД, когда мы собираемся работать с не-фиксированным перечнем баз данных. Проблема – вышеперечисленные возможности импорта статичны
* Используем заголовок манифеста **DynamicImport-Package:** org.apache.derby.jdbc, который позволяет контейнеру поискать запрашиваемый класс в перечисленных пакетах. В наименовании пакета можно использовать «\*», поддерживаются фильтры на импортируемые пакеты
* Ищет сначала в статически определенных местах, а только потом в динамических пакетах. Однажды найденная связка с динамически импортируемым пакетом фиксируется
* Недостатки
  + Система модулей OSGi идет побоку со всеми своими проверками, нарушение архитектурных принципов
  + Затраты на поиск во время выполнения и непредсказуемый результат операции

## Опциональный импорт

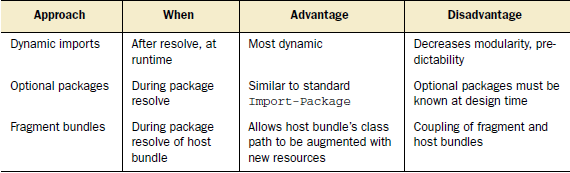
* Когда знаем что импортируем, но то ли будем использовать классы, то ли нет
* Используем опцию заголовка манифеста **Import-Package:** org.apache.derby.jdbc;**resolution**:=**optional**
* Во время загрузки bundle будет произведено связывание с пакетом, но если пакет не будет обнаружен (никто его не экспортирует), то ошибки не будет. Соответственно надо защищать вызов **Class.forName()**
* Применяется, когда нам надо грузить набор классов из некоторого числа предопределенных (совместимых, например)
* Динамический VS опциональный
  + И динамические и опциональные импорты никогда не приводят к ошибкам связывания. Работа с классами из таких пакетов должны обрамляться обработкой ClassNotFoundException.
  + Связывание опциональных импортов производится только один раз при загрузке bundle и больше не повторяются, связывание динамических – по мере необходимости классов, попытки повторяются снова и снова до успешной загрузки класса.
  + В остальном очень похоже, можно менять одно на другое просто изменением метаданных, без переписывания кода. Опциональный добавлен для совместимости с опциональным импортом bundle

## Фрагментирование bundle

* Другая возможность разделить один bundle на несколько частей.
* Используемый bundle просто добавляет свое содержимое в основной bundle.
  + Результат представляет собой одно целое и грузится ОДНИМ загрузчиком классов, таким образом проблема с доступностью приватных классов исчерпывается. В отличии от импорта bundle мы имеем реальное представление одного пакета, а не искусственную сумму
  + Вначале ищется класс в основном bundle, затем в добавленных фрагментах в порядке увеличения ID bundle
  + Главный bundle может использоваться и самостоятельно, а вот части его самостоятельно быть использованы не могут.
* Добавляются в основной bundle не только классы, но и содержимое манифеста, например упомянутые в **Import-Package** пакеты
* Употребляется, когда
  + пишем расширение к уже известной системе, для которой перечень импортов уже зафиксирован не нами. То есть система про нас ничего не знает, но мы хотим, чтобы она могла динамически загрузить наш класс.
  + При локализации, когда она занимает много места или пишется не нами + возможность добавления новых без изменения в основном bundle и без изменения его версии
* Используем заголовок манифеста **Fragment-Host:** manning.osgi.federated-database
* Подгружаемый bundle
  + Может принадлежать только одному bundle
  + не может определить свой активатор, так как не имеет своего поведения
  + должен быть инсталлирован перед основным bundle. Инсталлированные после основного фрагменты просто игнорируются, то есть динамическая подгрузка фрагментов не поддерживается
  + не может использовать заголовков: Import-Package, Require-Bundle, Bundle-NativeCode, DynamicImport-Package, and Bundle-Activator
  + может добавлять экспортируемые пакеты
* Для подгружаемых через Bundle-ClassPath внедренных jar, определенных в основном bundle
  + Перечень может включать jar как из основного jar, так и их фрагментов. Если фрагмент не найден, то jar из него просто не грузится
  + Если и основной bundle и фрагмент содержат один и тот же jar, то используется jar из фрагмента – возможность переопределения дефолтной функциональности, патчирования, адаптации к операционной системе и так далее

## API исследования

* Сравнение опций



* API исследования
  + **Bundle.getHeaders()** возвращает атрибуты из **META-INF/MANIFEST.MF**
  + Можно определять свои пространства имен для публикации собственной информации о bundle и пакетах
  + **BundleRevision** может описывать как последнюю ревизию, так и предыдущие, которые уже деинсталированы, но еще не удалены, так как не отпущены клиентами
  + Исследование экспорта – **BundleRevision** и **BundleCapability**

**BundleRevision revision = bundle.adapt(BundleRevision.class);**

**List<BundleCapability> capabilities =**

**revision.getDeclaredCapabilities(BundleRevision.PACKAGE\_NAMESPACE)**

**g 4.1 Retrieving afor (BundleCapability capability : capabilities) {**

**Map<String, Object> attributes = capability.getAttributes();**

**System.out.println("Package name = " + attributes.get("osgi.wiring.package"));**

**System.out.println("Package version = " + attributes.get("version"));**

**}**

* + Исследование зависимостей

**List<BundleRequirement> requirements =**

**revision.getDeclaredRequirements(BundleRevision.PACKAGE\_NAMESPACE);**

**for (BundleRequirement requirement : requirements) {**

**Map<String, Object> attributes = requirement.getAttributes();**

**System.out.println("Package name = " + attributes.get("osgi.wiring.package"));**

**System.out.println("Package version = " + attributes.get("version"));**

**}**

* + Исследование связей

**List<BundleWire> providedWires = revision.getWiring().**

**getProvidedWires(BundleRevision.PACKAGE\_NAMESPACE);**

**for (BundleWire bundleWire : providedWires) {**

**Bundle requirerBundle = bundleWire.getRequirerWiring().getBundle();**

**System.out.println("Requirer's bundle symbolic name = "**

**+ requirerBundle.getSymbolicName());**

**}**

* Развертывание
  + **Bundle.getLastModified()**
  + **Bundle.update() и Bundle.update(InputStream)**
  + **Bundle.getLocation()**

## Программная активация

* Вместо main в пакете должен быть класс реализующий интерфейс **BundleActivator,** класс должен быть упомянут в манифесте как **Bundle-Activator.**
* Нет необходимости экспортировать пакет в котором находится активатор, если, конечно, нет идеи разделить его с каким-то другим bundle. Хотя вообще-то это достаточно плохая идея и лучше держать его приватным.
* Методы интерфейса:
  + **public void start(BundleContext context)**
    - захватить необходимые ресурсы, запустить потоки обработки, установить необходимые слушатели, получить ссылки на необходимые сервисы
    - при старте активаторы запускаются последовательно одним потоком платформы
      * необходимо отработать как можно быстрее, все длительное – выбрасывать в отдельный поток. Любая задержка препятствует активации остальных bundle
      * То есть практически любая реальная работа клиента должна осуществляться асинхронно, в отдельном потоке
  + **public void stop(BundleContext context)**
    - высвободить ресурсы, остановить собственные потоки, выполняющие реальную работу, ну и так далее
    - выполнение может занять значительное время
* Функции start и stop вызываются платформой на одном и том же экземпляре объекта. В процессе вызова данных методов bundle находится соответственно в состояниях STARTING / STOPPING. В принципе не гарантируется, что методы будут вызваны одной и той же нитью, поэтому разумно члены класса делать **volatile**
* В импортируемых пакетах должен быть упомянут **org.osgi.framework**
* Весь код, работающий со спецификой OSGi платформы должен локализоваться в активаторе и сопутствующих классах, например, слушателе.
  + Остальные классы приложения, как реализации сервисов, так и их клиенты, НЕ должны знать о платформе. Активатор берет на себя очистку передаваемых им объектов от специфики развертывания
  + За счет этого остальные классы становятся легко доступными для тестирования, а также при необходимости для миграции на другие платформы
* можно активировать не программно, а декларативно за счет blueprint сервиса, но это фактически то же самое, просто централизованный активатор, располагающийся в blueprint сервисе, использует поставляемое bundle декларативное описание
* Вместо класса активатора можно определить заголовок манифеста Main-Class, который запустит главный класс по типу того, как это делает JVM. Аргументы можно передать методу main там же, после наименования класса

## Контекст выполнения

* Основное средство, предоставляемое активатору – интерфейс **BundleContext,** предоставляет приложению методы для доступа к функциональности OSGi
  + Методы интерфейса
    - **String getProperty(String key);**
    - **Bundle getBundle();**
    - **Bundle installBundle(String location, InputStream input)**
    - **throws BundleException;**
    - **Bundle installBundle(String location) throws BundleException;**
    - **Bundle getBundle(long id);**
    - **Bundle[] getBundles();**
    - **void addBundleListener(BundleListener listener);**
    - **void removeBundleListener(BundleListener listener);**
    - **void addFrameworkListener(FrameworkListener listener);**
    - **void removeFrameworkListener(FrameworkListener listener);**
  + Каждому активному bundle предоставляется собственный уникальный экземпляр **BundleContext**, существующий столько же, сколько соответствующий экземпляр **Bundle**. Если bundle не активный, то большинство методов контекста возбуждают исключения. Роль контекста – безопасность и управление ресурсами. Приватный объект, не рекомендуется выпускать экземпляр контекста за пределы bundle.
* Описание bundle и методы управления им доступны посредством интерфейса Bundle
  + Методы
    - **BundleContext getBundleContext();**
    - **long getBundleId();**
    - **Dictionary getHeaders();**
    - **Dictionary getHeaders(String locale);**
    - **String getLocation();**
    - **int getState();**
    - **String getSymbolicName();**
    - **Version getVersion();**
    - **void start(int options) throws BundleException;**
    - **void start() throws BundleException;**
    - **void stop(int options) throws BundleException;**
    - **void stop() throws BundleException;**
    - **void update(InputStream input) throws BundleException;**
    - **void update() throws BundleException;**
    - **void uninstall() throws BundleException;**
  + если bundle уже ACTIVE, то вызов Bundle.start() ничего делать не будет
  + вызов Bundle.start(), Bundle.stop(),Bundle.update(),Bundle.uninstall() на самом себе выдаст IllegalStateException
    - причина - после вызова код обнаружит, что контекст испорчен, а классы грузить неоткуда, так как bundle больше не подключен к платформе. Если в Stop() есть ожидание окончания всех порожденных потоков, то попытка из них остановить bundle вызовет взаимную блокировку.
    - можно вызвать из отдельного потока, если вызов данных функций будет последней вызванной операцией
  + Bundle.update() без параметров обновляет либо из того места, откуда он был загружен, либо из места, определенного заголовком **Bundle-UpdateLocation**. Возникает путаница, а потому этот заголовок использовать не рекомендуется

## Версионность

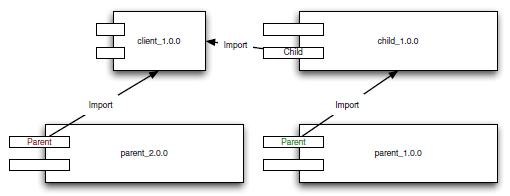
* Версии есть как для bundle, так и для пакетов. В отличии от maven, где тоже есть версии, в OSGi это не что-то вспомогательное, а корневое понятие. Так как bundle импортируют пакеты из других bundle на основе диапазона версий, то правильное определение версии напрямую влияет на видимость классов. Целостность пространства классов напрямую зависит от механизма версий.
* Формирование политики определения и изменения версии достаточно сложная задача, лучший способ ее решения – не экспортировать пакет, тогда и версия не понадобится
* Типовые проблемы: хочется переименовать интерфейс или изменить параметры его методов
  + Просто изменить существующий интерфейс мы не можем, можно сделать новый интерфейс2, но
    - Клиент получив новейший пакет не знает какой интерфейс использовать интерфейс2, интерфейс3 или какой еще
    - Клиент не может сослаться на «последнюю» версию интерфейса, только на какую-то последнюю
  + Решается введение версий bundles и пакетов внутри них, изменение интерфейса в обязательном порядке ведет к смене версий
* Дефолтная политика
  + Формат **- major\_number. minor\_number.micro \_number.qualifier\_(alpha|numeric|’\_’|’-’)+** Дефолтная – 0.0.0, остальные приводятся 1 == 1.0.0. Чтобы определить квалификатор обязательны первые три компоненты версии
  + Изменение **major** обозначает, что измененные классы более не совместимы обратно с использующими их клиентами из-за измененных сигнатур / удаления методов, то есть попытка запуска даст ошибку отсутствия нужных методов. Добавление новых методов не приводит с изменению данной компоненты версии.
  + Изменение **minor** обозначает, что измененные интерфейсы совместимы с использующими, но более не совместимы с имплементирующими их клиентами из-за добавления методов. То есть попытка компиляции даст ошибку, что не все требуемые методы реализованы. Использующие же интерфейсы клиенты сохраняют совместимость.
  + Изменение **micro** и **qualifier** обозначает, что изменилось поведение классов, например, исправлены баги или увеличена производительность
  + **qualifier** - сравниваются как строки, отсутствие – «», поэтому 1.0.0.beta новее чем 1.0.0
* Для bundle
  + Позволяет разработчику вносить изменения в интерфейс без нарушения работы уже существующих клиентов
  + определяется заголовком **Bundle-Version** в манифесте
  + сочетание **Bundle-SymbolicName** и версии однозначно идентифицирует bundle в среде. Попытка загрузить два bundle с одинаковым наименованием и версией приведет к исключению
  + прилично именовать файл как <наименование>\_<версия>.jar
* Для экспортируемого пакета указывается в составе наименования **Export-Package**: manning.osgi.helloworld;**version**=1.1
* Для импортируемого пакета в составе наименования может указываться как одна версия, так и диапазон версий.
  + Если клиент пакета не определил версию, то он согласился на максимальную из доступных контейнеру, то есть от 0.0.0 до бесконечности **Import-Package:** manning.osgi.helloworld; **version**="[1.1,1.1.9]",
  + Если мы используем классы из пакета, то мы должны определять совместимость на уровне **major** компоненты – [1,2)
  + Если мы реализуем интерфейсы из пакета, то мы должны определять совместимость на уровне **minor** компоненты – [1,1.1)
  + При определении диапазон «[» - включая, «(» - не включая, с правыми аналогами аналогично
* Смена версии хотя бы одного из пакетов должна приводить к смене версии всего bundle
  + Типовая политика: менять в версии bundle компоненту, соответствующую измененной компоненте версии пакета.
  + в противном случае получим проблему расщепления пакета, когда клиент может, сославшись на два других bundle (Б1, Б2), зависящих от разных версий данного (А), в итоге получить классы пакета А, загруженные разными загрузчиками, а потому несовместимые между собой
  + вообще нельзя экспортировать одну версию пакета в более чем одном bundle
  + к сожалению, зачастую версия bundle / пакета должны соответствовать версии продукта или библиотеки, что сильно затрудняет процесс

## Вычисление зависимостей

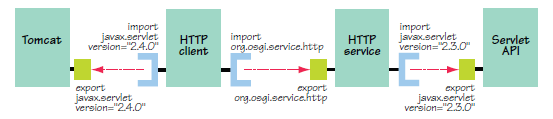
* OSGi автоматически выполняет привязку bundle к источникам импортируемых пакетов (resolving package dependencies) при запуске bundle или при попытке использования класса со стороны другого bundle. Resolving одного bundle в свою очередь может вызвать resolving других и далее рекурсивно. Если у bundle нет зависимостей, то он изначально resolved.
* При этом загрузчик классов импортирующего bundle получает ссылку на загрузчик классов bundle, который экспортирует класс и делегирует реальную загрузку классов ему.
  + Исключение – динамический импорт, обсуждается позже
  + Uninstall импортируемого bundle, остановка / запуск импортирующего bundle и даже перезагрузка контейнера (при перезагрузке восстанавливает состояние bundle) данную связь не разрывает и не приводит к изменению перечня доступных классов
  + только uninstall / install или update (reinstall из сохраненного в кэше файла bundle) импортирующего bundle.
* Если несколько bundle экспортирует один и тот же пакет, то цель загрузчика – сделать в итоге консистентный набор пакетов. То есть для каждого bundle прямо или опосредовано должна быть видна только одна версия пакета предоставляемая одним пакетом-провайдером.
  + Высший приоритет у уже resolved кандидата, затем по версии и по порядку инсталляции
  + Следующий приоритет у еще не Resolved кандидатов, затем по версии и порядку инсталляции
* Информацию о привязке bundle можно получить посредством сервиса **PackageAdmin**
  + Методы
    - **Bundle getBundle(Class clazz);**
    - **Bundle[] getBundles(String symbolicName, String versionRange);**
    - **int getBundleType(Bundle bundle);**
    - **ExportedPackage getExportedPackage(String name);**
    - **ExportedPackage[] getExportedPackages(Bundle bundle);**
    - **ExportedPackage[] getExportedPackages(String name);**
    - **Bundle[] getFragments(Bundle bundle);**
    - **RequiredBundle[] getRequiredBundles(String symbolicName);**
    - **Bundle[] getHosts(Bundle bundle);**
    - **void refreshPackages(Bundle[] bundles) –** если null, то все обновленные / удаленные bundle
    - **boolean resolveBundles(Bundle[] bundles) –** если null, то все пакеты, иначе список разделяемый « »
  + Основные возможности
    - Определение, какой bundle владеет каким классом
    - Определение для bundle установленных зависимостей и экспортируемых пакетов
    - Определение для пакета экспортирующих и импортирующих bundle
    - Ручной запрос повторного выполнения операции вычисления зависимостей
* При загрузке новой версии bundle невозможно просто автоматически начать грузить классы из него, так как часть классов из старой версии уже загружено и используется – можно устроить неконсистентность. Соответственно деинсталляция / обновление bundle выполняется в два этапа – подготовка + пересчет зависимостей за счет **refreshPackages()**.
  + при обновлении в какое-то время в системе реально существует две версии перегруженного bundle. Любой update без последующего refresh это всегда установка двух версий bundle.
  + при удалении bundle удаляется их всех списков, но продолжает существовать в памяти, так как его классы еще используются клиентскими bundle для загрузки классов
  + автоматический refresh при любой операции обновления / деинсталляции не производится, так как операция может затрагивать целую группу bundle
  + Без выполнения **refreshPackages()** клиенты bundle не увидят новых версий классов, данная функция выполняется в асинхронном режиме:
    - В перечень пересчитываемых добавляются все bundle использующие пакеты, экспортируемые bundle в аргументе (указанные или все обновленные / деинсталлированные с последнего Refresh). Операция рекурсивно повторяется с пакетами, экспортируемыми bundle, добавленными в перечень
    - Каждый bundle из перечня ACTIVE->RESOLVED, RESOLVED->INSTALLED, зависимости разрушаются
    - Каждый bundle в UNINSTALLED удаляется из системы
    - Все bundle бывшие в ACTIVE возвращаются в него
    - По системе рассылаются **FrameworkEvent.PACKAGES\_REGRESHED**
  + Как быстро новые классы обновленного bundle будут увидены зависит от того, были ли они private или exported, в последнем случае еще и от того использовались ли они кем-либо или нет
    - Если классы private или их никто не использует – то сразу начинают использоваться новые
    - Если их кто-то использует – то только после **refreshPackages()**
  + Если bundle импортирует пакеты, которые он же экспортирует, то ситуация становится прикольной
    - Если импортированные пакеты соответствуют тем, которые экспортировались старой версией, то bundle остается привязанным к старой версии
      * Если только фиксили private классы, то может ничего и не случиться
      * Если новые версии private классов используются вместе со старыми версиями экспортированных, то могут случатся проблемы
    - Если импортированные пакеты НЕ соответствуют тем, которые экспортировались старой версией
      * Новые экспортируемые классы становятся данному bundle видны сразу
      * Остальные bundle продолжают видеть старые классы, вплоть до refresh
* Чтобы избежать подобных проблем надо использовать три bundle: клиент, сервер и API, при этом клиент и сервер оказываются непосредственно не связанными друг с другом, что позволяет их подменять.

## Загрузка классов

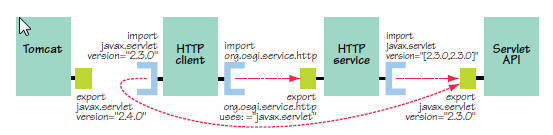
* Обычное приложение – 2 загрузчика классов, системный и приложения (дочерний). Дочерний делегирует системному загрузку всего, чего не имеет сам. В случае JEE приложения тоже иерархия загрузчиков, просто более длинная, но проблемы загрузки одного класса из нескольких источников нет.
* OSGi – каждый bundle имеет свой загрузчик классов для внутреннего содержимого (на основе **Bundle-ClassPath**), кроме того он делегирует загрузку остальных классов загрузчикам bundles, чьи пакеты импортирует. Те в свою очередь поступают так же. В результате имеем граф загрузчиков, в котором могут быть даже циклы.
* Проблема - вероятность загрузить один класс несколькими загрузчиками резко возрастает. Например:



* Налицо конфликт между зависимостями двух bundle и их внешними зависимостями. Возникает когда
  + сигнатуры методов экспортируемых пакетов содержат классы импортируемых пакетов
  + классы экспортируемых пакетов расширяют классы импортируемых пакетов
* В представленном примере пространство классов неконсистентно – client\_1.0.0 видит две версии пакета Parent, одну непосредственно импортируемую, и одну через импорт пакета Child
* В случае пары конфликтов в системе такое легко разрулить, но все усложняется
  + когда bundle сотни и подобных циклов много…
  + когда bundle динамически добавляются / удаляются
  + когда resolving транзитивно захватывает новые bundle
  + Возникает желание определить не просто экспорт пакета, но и учесть требования к пакетам, которые тот в свою очередь импортирует. То есть определить, классы из каких пакетов он собирается в свою очередь использовать и по ним разрулить конфликт версий
    - **Export-Package:** manning.osgi.child;**version**=1.0.0;**uses**:= manning.osgi.parent, перед uses используется «;»
    - В использованном примере платформа сможет зафиксировать конфликт требований. То есть надо будет либо клиенту использовать версию 1, либо child версию 2
    - Атрибут **uses** – транзитивный, то есть последовательно может налагаться несколько ограничений за счет использования в свою очередь ограничений пакета, упомянутого в **uses –** если bundle экспортирует foo, который использует импортируемый bar, а выбранный экспортер bar использует baz, тогда bundle импортирующий foo будет иметь одинаковые со всеми ними версии bar и baz
    - Было

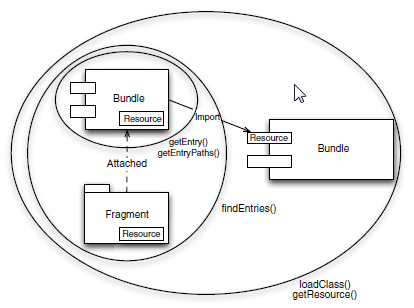


* Стало



* + - Слишком широко использовать **uses** плохо – нет ничего плохого, когда разные компоненты используют разные версии средств общего назначения, например XML парсеров
* Другая проблема – race condition между bundle. Например, какие-то статические поля могут инициализироваться активатором в методу start. Но другие bundle посредством импорта могут получить к ним доступ уже после загрузки и до того как активатор был запущен. В результате увидят мусор. Статических членов надо в общем случае избегать.
* Порядок загрузки классов
  + Если класс \*.java, то он грузится корневым загрузчиком системы. То же самое относится ко всем классам, упомянутым в свойстве **org.osgi.framework.bootdelegation**.
    - В значениях можно использовать «\*»
    - Данной возможностью лучше пользоваться по минимуму, чтобы минимизировать зависимость от нижележащей JVM
  + Поиск в пакетах, упомянутых в **Import-Package**, если он относится к импортируемому пакету, но не найден, то ошибка.
    - Ошибка будет даже в том случае, когда класс на самом деле существует внутри самого bundle
    - Можно пакет упомянуть и в импортируемых и в экспортируемых. Тогда сначала будет произведена попытка найти в других bundle и только в случае неудачи класс будет загружен из себя. Своеобразное переопределение класса другими bundle
  + Поиск в импортируемых bundles. Если класс не найден в одном импортируемом bundle, то поиск продолжается в следующем.
  + Поиск в себе
  + Поиск в фрагментах
  + Поиск в динамически импортируемых пакетах
* Bundle, который импортирует пакет, который сам же экспортирует –специальный случай, связанный с тем, что OSGi эот легковесный фреймворк, где отдельные компоненты получают прямые ссылки на объекты классов друг друга. Значит, все участвующие в кооперации bundle должны использовать одни и те же версии классов, загружаемые из одного и того же места
  + Например, API и несколько реализаций. Если мы экспортируем API из каждой реализации, то разные клиенты могут схватиться за разные API и потом не смогут общаться друг с другом. Если выделяем API в отдельный bundle, то усложняем развертывание.
  + Если же каждый bundle экспортирует и импортирует API, то фрейморк может экспортировать API из одного bundle, а все остальные и клиенты и реализации это API импортируют, обеспечив совместимость. Так как сначала грузятся классы из импортируемых bundle, а потому уже собственные, то в итоге все bundle, в том числе bundle реализации, будут использовать одно и то же API –substitutable export
  + Изначально до R4 так и было, экспорт всегда автоматически означал и импорт. В R4 решили данный кейс сделать явным
  + Можно экспортировать и импортировать разные версии пакетов, как правило экспортируемая версия > импортируемая, это означает, что сам bundle может работать также и с более старой версией
  + Если экспортируемые пакеты как-либо зависят от приватных, не экспортируемых, то мы только экспортируем их. Если же можно ожидать, что bundle могут оказаться в ситуации выбора наших пакетов от разных провайдеров, то надо и экспортировать и импортировать. Например, если мы содержим и экспортирует какую-нибудь библиотеку и ожидаем, что какой-то еще bundle тоже может это сделать.
* Можно вручную импортировать пакет из какого-то специального bundle
  + указав его идентификатор / версию в свойствах импорта **bundle-symbolic-name / bundle-version**
  + указав в экспорте этого пакета какое-то свойство и задав в импорте пакета фильтр на данное свойство, например **provider-name**
* можно наоборот экспортировать пакет, наложив условие, что другой bundle может его импортировать, только специально указав фильтр на какое-то свойство.
  + Для этого в описании экспортируемого пакета надо упомянуть свойство, по которому будет фильтроваться импорт, в атрибуте **mandatory**. Bundle НЕ наложивший фильтр на свойство никогда этот пакет не получит
  + Полезно, если мы хотим экспортировать какую-то особенную (например патчированную) реализацию API для специального применения и не хотим, чтобы оно попала к обычным пользователям.

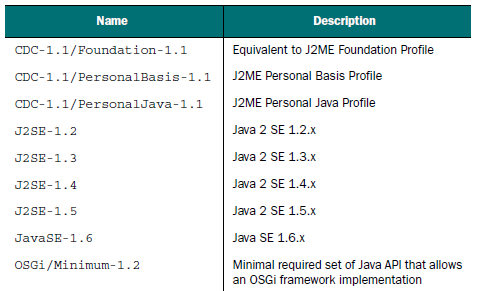
## Доступ к ресурсам



* **Bundle.loadClass(String className)** – загружает классы из bundle, из импортированных им пакетов, из его фрагментов. Требует, чтобы bundle был предварительно успешно связан, иначе **ClassNotFoundException** возбуждается и событие **FrameworkEvent** с ошибкой отправляется. Если связывания еще не производилось, то оно инициируется.
* **Bundle.getResource(String name) и Bundle.getResources(String name)** - загружает классы из bundle и из импортированных им пакетов
* **Bundle.findEntries(String path, String filePattern, Boolean recurse)** - загружает классы из bundle и из его фрагментов, соответственно НЕ требует выполнения его связывания
* **Bundle.getEntry(String path) и Bundle.getEntryPaths(String path)** – только из самого bundle
* Аналогично ServiceTracker есть BundleTracker, позволяющий отслеживать появление / исчезновение bundle в системе. Очень полезная штука для расширителей, который для вновь появившихся bundle перебирают ресурсы и предоставляют их другим компонентам системы

## Нативный код

* Если имеем требования со стороны bundle к окружению, то можно их определить через **RequiredExecutionEnvironment**, в него через «,» перечень поддерживаемых окружений. Наименование текущего окружения доступно через свойство контекста **org.osgi.framework.executionenvironment**



* Фреймворк НЕ проверяет содержимое bundle на совместимость с окружением, а просто налагает требования на этапе загрузки (resolve).
  + То есть можно загрузить несовместимый с окружением bundle, но если никто не попытается от него зависеть, то ничего страшного не случится.
  + Сделано, так как тип окружения может меняться от запуска к запуску, например, версия JVM
* Принципы работы с нативным кодом в Java
  + Нативный код реализуется как специальный тип методов класса, которые помечаются native. Его клиенты – нормальные классы, они ничего не знают о специфике имплементации
  + Они компилируются нормально, но затем посредством javah генерируются стабы на С и Н файлы для формирования имплементации
  + Библиотека с реализацией грузится через System.loadLibrary()
* Недостатки: нативный код ничего не знает о сборке мусора, подвержен проблеме с указателями, зависит от платформы
* OSGi предлагает инструмент внедрения множества нативных библиотек в bundle, позволяя их различать по описателям и подгружать необходимые
* Описание через **Bundle-NativeCode**
  + Грамматика и параметры



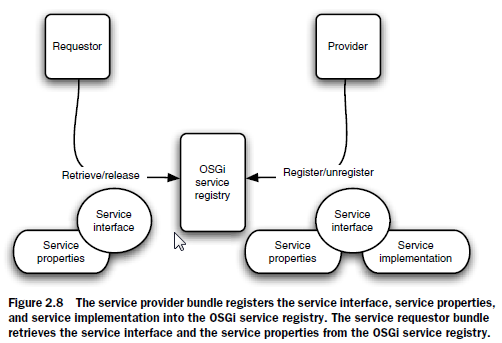
* + osname—Name of the operating system
  + osversion—Operating system version range
  + processor—Processor architecture
  + language—ISO code for a language
  + selection-filter—LDAP selection filter
  + пример: Bundle-NativeCode: lib/math.dll; lib/md5.dll; osname=WindowsXP; processor=x86
* Если Bundle-NativeCode определен, то bundle должен в обязательном порядке налагать требования на поддерживаемые платформы
* Если говорим, что в отсутствии подходящих библиотек реализуем функциональность самостоятельно через java, то в конце после запятой ставим "\*"

# Сервисы

* Отличие от просто вызова метода: сервис в обязательном порядке предполагает контракт главным образом в виде интерфейса , потребителю не важно как и кто обеспечивает сервис, пока он соответствует контракту. Использование сервиса зачастую предполагает этапы поиска сервиса и установки соглашений между поставщиком и потребителем.
* Компонент-ориентированная архитектура концентрируется на провайдере сервиса, облегчает архитектору процесс реализации и развертывания функциональности.
* сервис-ориентированная архитектура концентрируется на потребителе, облегчая ему процесс использования функций без учета механизма их предоставления, но с акцентом на качество: безопасность, транзакции, шифрование и так далее
* интерфейс не полностью описывает сервис, не учитывает показатели качества, возможность конфигурационных настроек и так далее, нефункциональные требования и так далее. Все это отражается в метаданных, которыми сопровождается интерфейс
* показатели для использования сервисов: низкое зацепление, использование интерфейсов, метаданные, множественные реализации, AOP. Не стоит использовать сервисы для взаимодействия двух тесно связанных кусков кода, которые всегда разрабатываются и обновляются вместе.
* OSGi изначально не скрывает динамизм составляющих (компоненты и сервисы), но предлагает набор техник и утилит, позволяющих адаптировать данную гибкость в соответствии с требованиями приложения.

## Общая архитектура

* поменяв поведение реализации можем перезалить bundle – поведение клиента не меняется пока мы не перепривяжем его.
  + Проблема – клиент в момент инициализации связывается с конкретными классами и заменить их динамически не представляется возможным.
  + Решение – связывать клиента только с некоторым интерфейсом, а OSGi будет отвечать за динамическую подгрузку его реализации. Концепция сервиса
* Основная цель – возможность смены реализации предоставляемой услуги без выполнения переустановки клиента
* Состоит из трех частей
  + Интерфейс.
    - Никак не связывается с OSGi классами и вообще с какой-либо инфраструктурой, что позволяет легко тестировать сервис, а также переносить между различными платформами
    - Интерфейс необходимо выносить
      * В отдельный импортируемый пакет, чтобы клиент принципиально не мог импортировать и использовать классы реализации
      * В отдельные bundle, чтобы иметь возможность обновить реализацию отдельно от API. Клиент привязывается к конкретному bundle с API и если реализация будет в том же bundle, то реализация останется загруженной обновления клиента, который отпустит bundle
  + Реализация.
    - Также нет требований по наследованию / реализации каких-либо инфраструктурных классов, что делает реализацию легко тестируемой и переносимой
    - Необходимые услуги, например связь с источником данных, доставляются реализации сервиса активатором, который берет на себя всю работу, связанную со спецификой платформы
  + Свойства сервисов. Набор пар «имя-значение», описывающих сервис.
    - Свойства могут использоваться клиентом для отбора только «подходящих» ему реализаций сервиса среди множества доступных
    - В принципе в качестве отдельных сервисов можно публиковать и прикладные объекты приложения, используя средства поиска в качестве некоторой внутренней базы данных
    - В качестве значений свойств можно использовать только примитивы или стандартные типы / коллекции. Зависимость bundle друг от друга по пакетам и доступность классов для свойств не работает
    - Если планируется описывать сервис свойствами, то наименования свойств необходимо описывать как константы в интерфейсе
* Работа с сервисами осуществляется через OSGi реестр, который позволяет
  + Сервисам регистрироваться и раз-регистрироваться
  + Клиентам получать ссылку на сервис и высвобождать ее
  + OSGi реестр работает как медиатор, изолируя поставщиков сервисов от потребителей.
    - Участвуют три bundle: интерфейс сервиса, поставщик сервиса и потребитель сервиса
      * Поставщики имеют дело с интерфейсом, имплементацией и свойствами сервиса
      * Потребители имеют дело только с интерфейсом и свойствами
    - Реестр динамически связывает получаемый потребителем интерфейс с предоставляемой поставщиком реализацией



* Уникальная идентификация сервиса
  + Есть свойство **SERVICE\_ID,** его значение генерится автоматически и присваивается платформой когда сервис регистрируется. После перерегистрации разумеется меняется. Платформа гарантирует уникальность.
  + Можно использовать свойство **SERVICE\_PID,**которое задается регистратором самостоятельно и переживает рестарт системы. Платформа должна гарантировать уникальность, но пока никто этого не делает
    - Если задан **SERVICE\_PID,** то интерфейс сервиса при запросе у реестра можно не указывать
    - Надо использовать **SERVICE\_PID,** а не собственное какое-нибудь свойство для унификации, а также потому, что платформа должна гарантировать его уникальность
* Основной инструмент для работы с сервисами – BundleContext
  + **void addServiceListener(ServiceListener listener, String filter);**
  + **void addServiceListener(ServiceListener listener);**
  + **void removeServiceListener(ServiceListener listener);**
  + **ServiceRegistration registerService(String[] clazzes, Object service, Dictionary properties);**
  + **ServiceRegistration registerService(String clazz, Object service, Dictionary properties);**
  + **ServiceReference[] getServiceReferences(String clazz, String filter);**
  + **ServiceReference[] getAllServiceReferences(String clazz, String filter);**
  + **ServiceReference getServiceReference(String clazz);**
  + **Object getService(ServiceReference reference);**
  + **boolean ungetService(ServiceReference reference);**
* Пакетирование интерфейса сервиса вместе с реализацией VS в отдельном bundle
  + Поддерживаются обе опции, обеспечивается корректное связывание при условии правильного указания метаданных
  + За - каждый bundle с реализацией становится самодостаточным, нет опасности забыть установить bundle с API. Выделение отдельного API для интерфейсов ведет к росту числа bundle
  + Против
    - Ели мы обновим реализацию, то волной пойдут обновляться все клиенты, завязанные теперь не только на API, но и на реализацию – слишком тяжелый процесс
    - Если мы удалим реализацию, то реально она выгрузится только тогда, когда клиенты перестанут использовать классы API – они используют общий загрузчик классов

## Реализация сервиса

* + При старте bundle регистрируем реализацию в реестре, описываем ее набором свойств

**private ServiceRegistration serviceRegistration;**

**public void start(BundleContext bundleContext) throws Exception {**

**Dictionary<String, Object> properties = new Hashtable<String, Object>();**

**properties.put(Auction.TYPE, SEALED\_FIRST\_PRICE);**

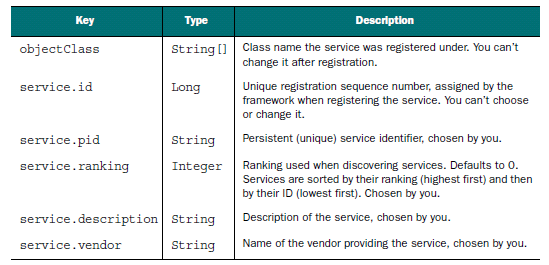
**properties.put(Auction.DURATION, DURATION);**

**serviceRegistration = bundleContext.registerService(**

**Printer.class.getName(),new PrinterImpl(),properties);**

**}**

* Регистрируемый класс интерфейса сервиса будет доступен в описании сервиса как свойство **objectClass**
* В принципе можно использовать и просто класс, не обязательно интерфейс, но это не рекомендовано.
* Использование Class.getName() облегчает рефакторинг системы
* Ссылка на регистрацию сервиса
  + Используется в дальнейшем для обновления метаданных сервиса и для удаления сервиса из реестра
  + из рук не выпускать. Концепция владельца сервиса, который ответственен за его жизненный цикл. Кроме того, регистрация связана с контекстом bundle, в котором она была получена и с ЖЦ публикатора
* регистрация сервиса автоматом порождает свойства Service.id и objectClass, которые не могут быть впоследствии изменены. Остальные стандартные свойства:



* + При остановке bundle дерегистрируем

**public void stop(BundleContext bundleContext) throws Exception {**

**serviceRegistration.unregister();**

**}**

В принципе можно и не де-регистрировать, это выполнится реестром автоматически при остановке bundle. Но самостоятельно выполнить это – хорошая практика.

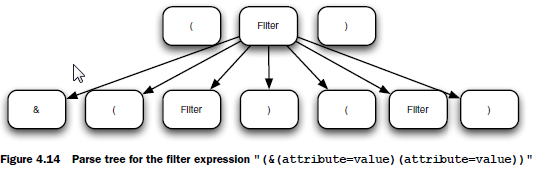
* + Можно регистрировать несколько реализаций одного интерфейса. Как правило, они различаются свойствами.
    - Если запрашивая, сервис клиент не указывает идентифицирующие свойства, то ему возвращается несколько с учетом приоритетности, задаваемой как свойство **SERVICE\_RANKING** при регистрации.
    - Если приоритетность не указана при регистрации , то устанавливается заданная системным свойством **osgi.framework.Constants.SERVICE\_RANKING**
    - Запрос с учетом свойств

**ServiceReference[] serviceReferences = bundleContext.getServiceReferences(**

**Auction.class.getName(),**

**"(&(auction-type=English)(auction-duration=50))");**

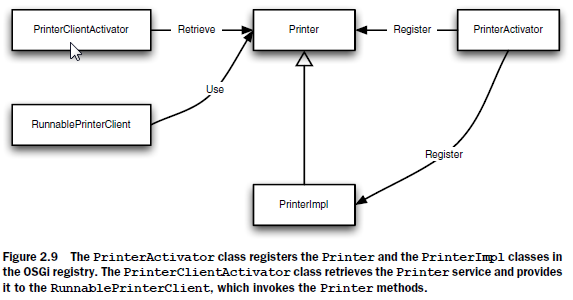
* + - * Фильтр базируется на LDAP синтаксисе с префиксным форматом
      * Сравнение наименований свойств без учета регистра
      * Сравнение значений с учетом их типа на основе операции equal(). То есть 10 и 10.0 будет разное или одинаковое для String и float типов свойств
* После регистрации свойства сервиса индексируются реестром сервисов. Изменение исходного объекта **Dictionary** бессмысленно.
  + Некоторые свойства принципиально неизменяемы, например **objectClass**
  + Но можно воспользоваться функцией **ServiceRegistration.setProperty(Dictionary properties)**
  + В свою очередь клиенты могут отслеживать событие **ServiceEvent.MODIFIED,** чтобы отреагировать на изменение значение свойства
  + Но вообще-то лучше не связываться с изменением свойств, а вместо этого использовать механизм конфигурирования
* Язык отбора сервисов по свойствам
  + Префиксная (или польская) запись операций



* + Наименования свойств не зависят от регистра, могут содержать пробелы, начальные и конечные отрубаются. Значения свойств чувствительны к регистру, начальные и конечные пробелы не убираются
  + Результат операции сравнения зависит от типов значения
    - = обрабатывается за счет **Object.equal().** Могут использоваться те классы, которые имеют конструктор, принимающий String.
      * Допустимы коллекции типа [a,b,c], тогда результат true, если хотя бы один член коллекции удовлетворяет условия. Данная особенность связана с тем, что «=» используется для поиска по **objectClass**, значение которого это коллекция наименований интерфейсов, реализуемых сервисом.
      * Если значение типа String или коллекция String, то допустимо использование «\*»
    - =\* задает требование присутствия свойства с таким наименованием
    - <= и=> обрабатываются за счет **Comparable.compareTo(),** соответственно значения свойств должны реализовать **Comparable** и иметь конструктор из String
    - ~= зависит от реализации, как минимум должен игнорировать регистра и пробелы при сравнении значений
  + Отдельные операции сравнения соединяются при помощи &|!, например (&(attr1=value1)(|(attr2=value2)(attr3=value3)))

## Архитектура клиента

* + Клиент должен зависеть только от интерфейса сервиса и никак не должен зависеть от классов платформы. Это позволяет держать клиента тестируемым и переносимым между различными платформами
    - Правильно получить ссылку на сервис, чтобы передать ее клиенту – обязанность активатора клиента. Таким образом, если нам надо клиента переместить на другую платформу, то придется поменять только активатор
    - В принципе можно использовать не активатор, а декларативную ссылку. Тогда обязанность доставки возлагает на соответствующий инфраструктурный сервис платформы (blueprint)
  + Обязанности активатора
    - Получить ссылку на сервис при старте клиента.
      * Активатор получает от реестра не непосредственно ссылку, а объект **ServiceReference**, который содержит в себе как ссылку, так и дополнительную информацию, например коллекцию свойств.
      * Активатор запускается служебной нитью платформы и должен как можно быстрее закончить работу
    - Высвободить ссылку на сервис при остановке клиента или когда сервис становится более не нужным клиенту. В принципе можно этого и не делать, так как это будет сделано автоматически платформой при выгрузке клиента, но хороший тон и позволяет правильно работать **Service Facory**
    - Если сервис может стартовать позже клиента, динамически подменяться или асинхронно включаться / отключаться (а это практически всегда), то зарегистрировать слушатель, который будет реагировать на события остановки реализации сервиса и ее замены. Вообще порядок загрузки и старта bundles не должен влиять на их работоспособность.



## Реализация клиента

* Первое о чем всегда должен помнить клиент – динамический характер сервиса, он может перестать существовать в любой момент времени или даже еще не существовать вовсе в момент старта клиентского bundle. ЖЦ сервиса абсолютно асинхронен с ЖЦ клиента и задача клиента это все обрабатывать. Стратегии обработки:
  + Хранить ссылку и получать из нее реальный указатель перед каждым использованием. Реализация может исчезнуть как раз между получением и использованием. Также может устареть сама ссылка на сервис, если, к примеру, поставщик сменит ее версию. Имеем типичный Race condition. Выход – Try / catch
  + Подписываться и обрабатывать поток событий **REGISTERED / MODIFIED / UNMODIFIED** от сервиса за счет **ServiceListener**, отслеживая самостоятельно его ЖЦ и получая при необходимости новый
  + Отдать это на откуп сервисному классу ServiceTracker
  + декларативная подписка за счет Blueprint сервиса, фактически тот же трекер, но создаваемый извне
* Получение ссылки на сервис

**private ServiceReference serviceReference;**

**public void start(BundleContext bundleContext) throws Exception {**

**serviceReference = bundleContext.getServiceReference(Printer.class.getName());**

**Printer printer = (Printer)bundleContext.getService(serviceReference);**

**// стартовать прикладной поток клиента, передав ему ссылку на сервис**

* + Использование ссылки на сервис вместо прямой ссылки на объект-реализатор позволят отделить использование сервиса от его реализации. Посредством непрямой ссылки мы можем контролировать доступ, отслеживать изменения, в том числе удаление или смену реализатора, поддерживать ленивую активацию по требованию
  + Все **ServiceReference**, относящиеся к однажды зарегистрированному сервису, эквивалентны между собой. Но если этот же сервис дерегистрировать и зарегистрировать опять, то ссылки будут уже не эквивалентны
  + Клиент должен хранить объект **ServiceReference**, хранить непосредственно ссылку на полученный интерфейс сервиса в долго живущей переменной не имеет смысл – реализация в любой момент может быть выгружена или заменена
  + В отличии от регистрации, ссылку на сервис можно передавать другим компонентам – она НЕ привязана к контексту зарегистрировавшего bundle
  + Каждый вызов **bundleContext.getService()** увеличивает счетчик использования экземпляра сервиса bundle клиентом, соответственно, когда сервис больше не нужен ОБЯЗАТЕЛЬНО надо
    - его отдать при помощи **bundleContext.ungetService()**
    - удалить ссылку, чтобы мусорщик смог бы выполнить свою работу и освободить при необходимости предоставляющий класс bundle. Особенно это плохо, когда ссылка идет не на интерфейс, а прямо на реализацию – она удерживает реализацию, даже при том, что bundle поставщик уже деинсталлирован
* любое использование сервиса должно оборачиваться try / catch – обработка его асинхронной природы и потенциально возможность удаленной реализации
  + Регистрация слушателя с определением фильтра на свойства получаемых событий при помощи LDAP синтаксиса. Подразумевается, то активатор сам работает слушателем, реализуя интерфейс **ServiceListener**

**bundleContext.addServiceListener(this,"(objectClass=" +Printer.class.getName() + ")");**

* + Обработка слушателем событий связанных с сервисом

**public void serviceChanged(ServiceEvent serviceEvent) {**

**switch (serviceEvent.getType()) {**

**case ServiceEvent.*UNREGISTERING:* {**

**// остановить клиента**

**bundleContext.ungetService(serviceEvent.getServiceReference());**

**break;**

**}**

**case ServiceEvent.*REGISTERED:* {**

**Printer printer = (Printer) bundleContext.getService( serviceEvent.getServiceReference());**

**// передача новой ссылки на сервис клиенту**

**break;**

**}**

**}**

* + - Получать ссылку на реализацию сервиса в событии REGISTERED безопасно, так как платформа доставляет события регистрации сервиса синхронно в одном и том же потоке с обработкой сервиса
    - Обрабатывая UNREGISTERED надо убеждаться, что оно относится именно к той реализации сервиса, которую мы сейчас используем
    - Событие регистрации сервиса могло произойти в прошлом, поэтому надо сначала поискать уже имеющийся сервис, а затем уже пописываться на события его регистрации. И все равно – событие может проскочить между двумя этими действиями. А потому ему можно послать ручками событие о псевдорегистрации – пусть проверит при старте
  + Высвобождение ссылки на сервис

**bundleContext.ungetService(serviceEvent.getServiceReference());**

* + Основной недостаток ручной реализации слушателя – масса кода, который необходимо будет написан для КАЖДОГО используемого сервиса. А их может быть десятки в приложении.
  + Вместо ручной регистрации слушателя можно использовать класс **ServiceTracker**, позволяющий удобнее отслеживать события. Особенно удобен для отделения прикладного кода от интерфейсов платформы, шаблон bridge.
    - * Создание в процессе старта активатора, при этом происходит скрытое создание слушателя и инициализация списка потенциально доступных в соответствии с фильтром сервисов, чтобы в случае, если один ушел – вовремя предложить другой

**String filter = "(&(objectClass=" + Auction.class.getName()+ ")(" + Auction.TYPE +**

**"=Sealed-First-Price))";**

**ServiceTracker tracker = new ServiceTracker(bundleContext, filter, null);**

**tracker.open();**

* + - * Использование (переспрашивание свежей реализации сервиса)

**// при старте bundle**

**Auction auction = (Auction) tracker.waitForService(0);**

**// в процессе работы**

**Auction auction = (Auction) tracker.waitForService();**

* + - * Очистка в процессе остановки активатора, отсоединение слушателя, очистка списка кандидатов

**Tracker.close()**

* + Можно кастомизировать работу трекера за счет передачи ему при открытии объекта, реализующего **ServiceTrackerCustomizer**:
    - **public Object addingService(ServiceReference reference);**
    - **public void modifiedService(ServiceReference reference, Object service);**
    - **public void removedService(ServiceReference reference,Object service);**
    - при получении сервиса-кандидата можно выделить какие-то ресурсы, можно его обернуть прокси и выдать ее в качестве результат, можно отказаться от данного конкретного сервиса, вернув null
* Одновременно может быть установлено в системе множество версий интерфейса сервиса. Bundle клиент связывается с какой-то конкретной и ему возвращаются только реализации сервисов, относящиеся к данной версии интерфейса
  + Какие-то реализации сервиса bundle может просто не видеть, они реализуют другие версии интерфейса, нежели использует он, попытка привести ссылку приведет к исключению
  + Способ увидеть все сервисы не зависимо от того, какие интерфейсы они реализуют

**ServiceReference[] references = bundleContext.getAllServiceReferences(null, null);**

* + Для ServiceListener есть вариант с All семантикой, позволяющий получать сообщения для всех сервисов. Для ServiceTracker такого нет, просто используется ServiceTracker .open(true)
  + Два bundle с одними и теми же импортами должны видеть один и тот же набор доступных сервисов

## Фабрика сервисов

* + Обычно ссылка с сервисом соотносится как один к одному, singleton. Однако может возникнуть необходимость в один ко многим
    - когда например сервис это telnet и каждый клиент хочет соединяться со своим хостом, то есть иметь собственный экземпляр реализации сервиса – prototype
    - когда надо выдать реализую сервиса, заточенную под конкретный запрашивающий bundle
    - когда надо создать экземпляр сервиса не сразу, а по требованию клиента
  + Реализация
    - При регистрации сервиса вместо реализации указываем фабрику реализаций имплементирующую **ServiceFactory**
      * Создает экземпляр сервиса в ответ на запрос клиента

**public Object getService(Bundle bundle,ServiceRegistration registration) {**

**return new TelnetImpl();**

**}**

* + - * Удаляет экземпляр сервиса

**public void ungetService(Bundle bundle,ServiceRegistration registration,**

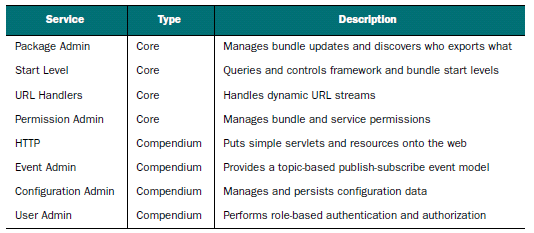
**Object serviceImplementation) {**

**////**

**}**

* + - Данные методы вызываются **ServiceReference**, которая видит, что данный bundle еще не запрашивал сервис. То есть один bundle – одна реализация сервиса, которые кэшируются в рамках bundle, что не вполне удобно, но что есть.
    - Удаляются закэшированные экземпляры, когда bundle клиент прекращает свою работу, bundle фабрика прекращает свою работу или когда число использований (get / unget service) становится равным 0.
    - В дальнейшем реальные реализации сервиса (TelnetImpl) находятся под управлением платформы, заботиться о них не надо
    - Фабрика изначально многопоточная, это надо учитывать
  + При регистрации в качестве интерфейса сервиса мы, по-прежнему, указываем интерфейс именно сервиса, а не класс фабрики. А вот в качестве реализации сервиса уже ставим фабрику
  + Клиент не в курсе как именно создается экземпляр сервиса – напрямую или фабрикой

## Стандартные сервисы



* К сожалению Java спецификация позволяет установить только одну реализацию URLStreamHandlerFactory во время старта системы. Фреймворк устанавливает собственную и дает возможность клиентам расширять его поведение

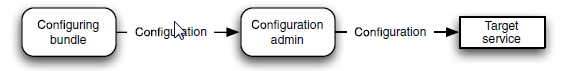
## Расширенные возможность

* + - * R4.2 спецификация предлагает расширенные возможности для вмешательства в работу реестра сервисов в виде трех интерфейсов, которые могут быть имплеметированы сервисами, желающими контролировать реестр на системном уровне.
* Подобные сервисы могут мониторить и даже маскировать системные события на уровнее реестра сервисов. Соответственно они легко могут накрыть систему медным тазом
  + org.osgi.framework.hooks.service.FindHook – вызываются, когда сервисы запрашиваются у реестра
  + org.osgi.framework.hooks.service.ListenerHook – когда в реестре регистрируются слушатели
  + org.osgi.framework.hooks.service.EventHook – позволяют маскировать события побликации, модификации и удаления сервисов
* одно из применений данных возможностей – поддержка работы удаленных сервисов и внедрение в систему собственной стратегии поиска реализации запрошенного сервиса, имеющего удаленную природу.

# Конфигурирование

## Архитектура

* Базовые требования к механизму конфигурации



* + Механизм конфигурирования не должен быть связан со спецификой конфигурируемого приложения и не должен ничего знать о его классах
  + Изменения перечня конфигурируемых параметров не должно влиять на интерфейс сервиса и требовать изменения клиентов
  + Централизованное хранилище данных, сервис не должен быть ответственен за их хранение, различные стратегии хранения
  + Ограничение доступа к конфигурированию сервисов
  + Конфигурационные данные должны становиться свойствами сервиса, чтобы по ним можно было найти сервис
* Представлен **ConfigurationAdmin** и **Metatype** сервисами
  + Конфигурационный сервис служит медиатором между конфигурирующим и конфигурируемым bundle
  + Можно определять конфигурацию bundle, которого еще нет в системе. Когда он запустится – он ее получит
  + Также ответственен за сохранение данных между перезапусками сервера
  + ConfigurationAdmin это интерфейс сервиса, а значит можно использовать разные реализации
* Конфигурационный объект – словарь конфигурационных свойств, идентифицируемый PID конфигурации.
  + Вообще, содержимое конфигурационного объекта становится артефактом разработки, так как новые / старые версии bundle должны корректно иметь дело с уже хранящимися в системе объектами. Это не так сложно в части upgrade, но достаточно неприятно в случае downgrade
  + Каждый конфигурационный объект уникально идентифицируется за счет PID. По умолчанию это <ид bundle>.<что угодно>. Вообще можно использовать любой формат и содержание, главное чтобы PID-ы были уникальными
  + Null переданный сервису вместо конфигурационного объекта означает, что сервис должен высвободить занимаемые ресурсы и прекратить работу

## Получатель конфигурации

* Получатель конфигурационного объекта должен реализовывать **org.osgi.service.cm.ManagedService**, так как реализация сервиса должна быть свободных от платформы, то объектом конфигурирования делаем активатор, который в свою очередь ответственен за правильное переконфигурирование регистрируемого им сервиса. Кроме того, конфигурационный объект заведомо требуется активатору при первичном создании реализации сервиса.
  + Интерфейс получателя включает только одну функцию - **public void updated(Dictionary properties) throws ConfigurationException**
  + Каждый объект конфигурирования должен иметь уникальный идентификатор, который будет использоваться администрирующим клиентом. Для этого используется свойство **SERVICE\_PID**
  + Bundle может зарегистрировать любое число сервисов-получателей конфигурации, главное чтобы у каждого был свой уникальный PID. Каждая зарегистрированная реализация **ManagedService** используется для конфигурирования одной сущности внутри bundle или вне него, например, внешнего устройства
  + Реализация, САМ активатор регистрируется как сервис типа **ManagedService**, потенциальный получатель конфигурационных сообщений

**public class NotificationBrokerActivator implements BundleActivator, ManagedService {**

**// ...**

**public void start(BundleContext bundleContext) throws Exception {**

**// ...**

**Dictionary<String, Object> properties = new Hashtable<String, Object>();**

**properties.put(Constants.SERVICE\_PID,"manning.osgi.notification.broker");**

**registration = bundleContext.registerService(ManagedService.class.getName(),**

**this,properties);**

**}**

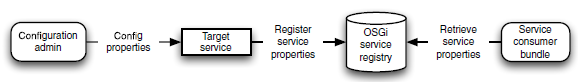
**@SuppressWarnings("unchecked")**

**public void updated(Dictionary configuration) throws ConfigurationException {**

**// ...**

**}**

* + То есть реализация сервиса регистрируется с интерфейсом сервиса, а активатор сервиса регистрируется с интерфейсом получателя конфигурационных объектов от Configuration Admin. Разнесение интерфейсов сервиса и его конфигурирования большой плюс
  + Получив конфигурационный объект, активатор должен либо сконфигурировать ранее созданный экземпляр сервиса, либо удалить старый и создать на основе полученных данных новый. Параметры конфигурации лучше засунуть в свойства сервиса, чтобы по ним клиент мог найти нужный экземпляр.



* + Метод **ManagedService.updated** вызывается платформой асинхронно и требует правильной синхронизации
* Кроме **ManagedService** также используется **ManagedServiceFactory** – первый получает один конфигурационный объект для одного сервиса, второй – множество конфигурационных объектов для множества сервисов
  + Интерфейс фабрики
    - **public String getName();**
    - **public void updated(String pid, Dictionary properties) throws ConfigurationException;**
    - **public void deleted(String pid);**
  + Необходимо использовать **ManagedService** когда конфигурация требуется для одного объекта /сервиса внутри / вне bundle.
    - В этом случае реализация **ManagedService** доставляет конфигурационный объект потребителю основываясь на его идентификаторе в форме PID.
    - по получению конфигурационного объекта мы переконфигурируем объект, по получению null – прекращаем предоставлять сервис или рвем связь со внешним объектом.
  + Необходимо использовать **ManagedServiceFactory,** когда мы одновременно поддерживаем конфигурацию множества сервисов / объектов, каждый из которых конфигурируется индивидуально.
    - по получении набора конфигураций закрываем удаленные сервисы, переконфигурируем существующие (с существующими PID), открываем новые (с новыми PID).
    - Фабрика оперирует FACTORY.PID вместо PID, каждый индивидуальный сервис по-прежнему идентифицируется как PID, который выглядит как FACTORY.PID.<уникальный идентификатор>
    - Внутри себя фабрика как правило хранит перечень активных сервисов в виде Map<PID,сервис >

## Конфигуратор

* Конфигурирующий bundle должен пользоваться услугами сервиса **ConfigurationAdmin**, получая у него конфигурационный объект с текущими параметра конфигурации, редактируя его и отдавая обратно.
  + Интерфейс **ConfigurationAdmin**
    - **public Configuration createFactoryConfiguration(String factoryPid) throws IOException;**
    - **public Configuration createFactoryConfiguration(String factoryPid, String location) throws IOException;**
    - **public Configuration getConfiguration(String pid, String location) throws IOException;**
    - **public Configuration getConfiguration(String pid) throws IOException;**
    - **public Configuration[] listConfigurations(String filter) throws IOException, InvalidSyntaxException;**
  + Интерфейс конфигурационных данных в виде **Configuration**
    - **public String getPid();**
    - **public Dictionary getProperties();**
    - **public void update(Dictionary properties) throws IOException;**
    - **public void delete() throws IOException;**
    - **public String getFactoryPid();**
    - **public void update() throws IOException;**
    - **public void setBundleLocation(String location);**
    - **public String getBundleLocation();**
  + Реализация конфигуратора

**ServiceReference serviceReference =**

**bundleContext.getServiceReference(ConfigurationAdmin.class.getName());**

**Configuration configuration =**

**configAdmin.getConfiguration("manning.osgi.notification.broker");**

**Dictionary<String, Object> configProperties = new Hashtable<String, Object>();**

**configProperties.put("port", 8080);**

configuration.update(configProperties);

* + Чтобы создать конфигурационный объект вызывается ConfigurationAdmin.getConfiguration(), который будет возвращать один и тот же объект для всех последующих вызовов. Поле bundleLocation идентифицирует bundle-получателя. Если его установить в null, то конфигурационный объект достанется первому подходящему по PID получателю, тот установит поле location в соответствующее себе значение и в дальнейшем только он и будет получателем данной конфигурации. Если получатель будет удален из системы, то поле автоматом установится в null и процесс выбора получателя повторится. Это некоторый механизм безопасности.
  + Пока метод **Configuration.update()** не вызван, никакой деятельности вокруг конфигурируемого сервиса не происходит, его вызов последовательно вызывает сохранение данных в хранилище, а потом вызов **ManagedService.updated**
  + Можно менять реализации стратегии хранения конфигурационной информации. По умолчанию Felix использует файловую структуру, но можно предложить собственную реализацию интерфейса **org.apache.felix.cm.PersistenceManager**, зарегистрировав ее как сервис с достаточно высоким рангом.
  + Значения свойств: простые Java типа, их массивы и коллекции. Прикладные объекты НЕ поддерживаются: не понятно как их поля использовать в фильтрах, а также сложно отслеживать зависимости для подобных типов. Решение – использовать XML, но язык запросов по отношению к нему сильно ограничен
  + Удаление конфигурации – через **Configuration.delete(),** конфигурируемый активатор получает вместо конфигурационного объекта null и должен соответствующим образом это обработать, например через установление дефолтных значений или просто опускание сервиса
* Специфика конфигурирования для фабрик сервисов:
  + Вместо **configAdmin.getConfiguration** для конфигурирования использовать **configAdmin.createFactoryConfiguration**, которой передается FACTORY.PID. Дальше все стандартно, включая вызов **configuration.update**
  + Клиент в **SERVICE\_PID** объекта конфигурации вместо «**broker**» должен использовать «**broker**-**factory**». При этом конфигурируемый объект по-прежнему регистрируется со старым **SERVICE\_PID**
  + Конфигурируемый объект должен теперь реализовывать **ManagedServiceFactory** и регистрироваться как сервис с интерфейсом **ManagedServiceFactory**
    - Методы интерфейса
      * **public synchronized void updated(String pid, Dictionary configuration)**
      * **public synchronized void deleted(String pid)**
    - возник дополнительный параметр PID
      * генерируется сервисом конфигурации, основан на **SERVICE\_PID** фабрики плюс дополнение, уникальной для каждого экземпляра сервиса
      * внутри себя активатор должен иметь карту pid-созданный за счет **getService** сервис
      * соответственно как только фабрика видит незнакомый PID она должна создать новый экземпляр сервиса и зарегистрировать в OSGi реестре сервисов (ну и прикопать естественно)
      * если конфигурация удаляется, то сервис надо дерегистрировать
* успешность конфигурирования
  + если конфигурационный сервис при применении конфигурации к сервису получает исключение, то он его отдает OSGi Log сервису (без сервиса логирования конфигуратор работать откажется) и продолжает работать.
    - Конфигурирующий bundle никакой информации о произошедшем не получает.
    - Если конфигурируемый bundle будет повторно запущен, то он опять получит те же самые действия и выбросит исключения
  + Варианты решения
    - Предоставлять метаописание конфигурационных данных, чтобы конфигурирующий bundle реже ошибался
    - Проверить свойства сконфигурированного сервиса (значения должны проноситься из конфигурации), вывести из этого успешность. Проблема:
      * фактическое обновление свойств выполняется асинхронно, надо ждать неопределенное время
      * проблемы с применением конфигурации могут быть не связаны непосредственно с данными

## Метаданные

* + Идеологически поддерживаются множественные конфигурирующие агенты: GUI, CLI, XML файла. Чтобы они работали их надо обеспечивать метаданными, обеспечивающими автоматическую генерацию интерфейса по описанию конфигурируемых свойств
  + Описание конфигурации должно реализовывать **ObjectClassDefinition**
    - ИД - совпадает с PID сервиса и конфигурационного объекта
    - Наименование метаданных
    - локализованное описание
    - иконка
  + перечень атрибутов, каждый из которых реализует **AttributeDefinition (**ИД – ИД описания конфигурации + свое имя, дефолтные значения, имя, опциональное наименование, опциональные значения, описание, кардинальность, 0 – одиночное значение и так далее, обязательность / опциональность, тип, валидатор значения)
  + Также как и конфигурационный объект, метаданные конфигурации привязываются к PID конфигурируемого сервиса, то есть один PID описывает ТРИ сущности
  + Метаданные конфигурации сервиса предоставляются активатором сервиса, который для этого реализует **MetaTypeProvider**
    - Импортировать пакет **org.osgi.service.metatype**
    - При запуске регистрирует себя как источник метаданных

**public void start(BundleContext bundleContext) throws Exception {**

**this.context = bundleContext;**

**Dictionary<String, Object> properties = new Hashtable<String, Object>();**

**properties.put(Constants.SERVICE\_PID, "manning.osgi.notification.broker");**

**registration = bundleContext.registerService(new String []**

**{ManagedService.class.getName(),MetaTypeProvider.class.getName()},**

**this,properties);**

**//….**

**}**

* + - По запросу предоставляет доступные локализации описания

**public String[] getLocales() {**

**return null;**

**}**

* + - По запросу предоставляет метаданные

**public ObjectClassDefinition getObjectClassDefinition(String pid,String locale) {**

**if (configurationSchema.getID().equals(pid)) {**

**return …;**

**}**

* + Вместо реализации **MetaTypeProvider** можно завести XML файл с описанием в директории OSGI-INF/metatype, за счет соответствующего стандартного расширителя, присутствующего в платформе, на основе этих данных будет автоматически создан и зарегистрирован соответствующий сервис
    - Пример файла

**<?xml version="1.0" encoding="UTF-8"?>**

**<MetaData xmlns="http://www.osgi.org/xmlns/metatype/v1.0.0">**

**<OCD name="EchoServer" id="4.7.1.1" description="Echo Server Config">**

**<AD name="port" id="4.7.1.1.1" type="Integer"**

**description="The port the Echo Server listens on"/>**

**</OCD>**

**<Designate pid="org.foo.managed.service">**

**<Object ocdref="4.7.1.1"/>**

**</Designate>**

**</MetaData>**

* + - Соответственно элементы OCD – ObjectClassDefinition, AD – AttributeDefinition, Designate – связь PID с определением метаданных
  + Потребление метаданных
    - Выполняется конфигурирующим агентом, но почему-то НЕ выполняется автоматически сервисом конфигурирования перед передачей объекта сервису
    - Пример

**ServiceReference [] serviceReferences = bundleContext.getServiceReferences(**

**MetaTypeProvider.class.getName(),**

**"(" + Constants.SERVICE\_PID + "="+ "manning.osgi.notification.broker" + ")");**

**MetaTypeProvider metaTypeProvider =(MetaTypeProvider)**

**bundleContext.getService(serviceReferences[0]);**

**ObjectClassDefinition ocd = metaTypeProvider.getObjectClassDefinition(**

**"manning.osgi.notification.broker",null);**

## Проблемы

* Безопасность - явных возможностей ограничить доступ нет, но:
  + во-первых, удаленное конфигурирование не поддерживается, то есть конфигурирующий bundle должен быть явно размещен в контейнере
  + каждый конфигурационный объект при первом получении ассоциируется с конфигурируемым bundle и в дальнейшем дается либо только ему, либо при помощи метода **ConfigurationAdmin.getConfiguration(String targetServicePID, String location).** Соответственно только один bundle может работать с каждым конфигурируемым сервисом
* транзакционность – нельзя обновить конфигурацию либо нескольких сервисов, либо ни одного

# Логирование

* Платформа имеет встроенный Log сервис, многие другие инфраструктурные сервисы без него вообще не запускаются, так как он для них стандартный канал сообщений об ошибках
* Как слушать Log сервис в поисках записей, исходящих от сервиса конфигурации.
  + - * Слушатель должен реализовывать **LogListener**
      * На запуске зарегистрироваться

**public void start(BundleContext context) throws Exception {**

**ServiceReference serviceReference =context.getServiceReference(**

**LogReaderService.class.getName());**

**LogReaderService logReader =(LogReaderService)**

**context.getService(serviceReference);**

**logReader.addLogListener(this);**

**}**

* + - * Обработка события

**public void logged(LogEntry entry) {**

**if (entry.getException() instanceof ConfigurationException) {**

**ConfigurationException configExcep = (ConfigurationException)**

**entry.getException();**

**if (configExcep.getProperty().equals("port")) {**

**// ...**

**}**

**}**

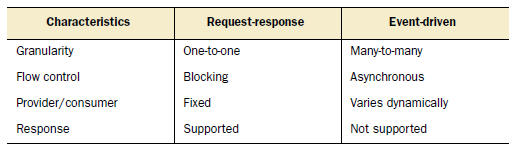
**}**

* Предоставляет только базовые возможности конфигурирования, прикладному коду лучше использовать что-нибудь получше, например Log4J
* Но плюс – стартует вместе с платформой, в самом начале, поэтому гарантированно доступен для bundle независимо от порядка загрузки
* Для публикации событий надо пользоваться сервисом LogService – compendium или none-core сервис
  + **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);**

# Обмен событиями

## Архитектура

* Осуществляется за счет Event Admin Service, который является медиатором между публикатором и подписчиками
* Слабейший способ связывания между поставщиком и потребителем какой-нибудь информации, общая точка – перечень свойств сообщения
* Пониженная связность публикатора и получателей способствует
  + увеличению вероятности возникновения deadlock. Пример: публикатор забирает монитор, а обработчик пытается сделать то же самое.
  + Увеличению масштабируемости
* Требованию к механизму
  + Публикатор не должен ничего знать о подписчиков
  + Новые публикаторы и подписчики должны добавляться и удаляться динамически
  + Подписчики, обрабатывающие события, не должны блокировать публикаторов
  + Поддержка N публикаторов к M подписчиков
* Архитектура событий vs архитектуры синхронных запросов



## Получение сообщений

* Получатель сообщений регистрируется активатором как сервис с определенными свойствами (шаблон whiteboard). В качестве получателя реализация интерфейса **EventHandler**, которая ответственен за передачу данных из событий коду приложения. Таким образом, приложение ничего об инфраструктуре может и не знать.
  + Регистрация активатора

**public void start(BundleContext context) throws Exception {**

**SimpleSubscriber subscriber = new SimpleSubscriber();**

**Dictionary dict = new Properties();**

**dict.put(EventConstants.EVENT\_TOPIC,"manning/osgi/LoginEvent");**

**context.registerService(EventHandler.class.getName(),subscriber, dict);**

**}**

* + - Подписка на сообщения определенного топика осуществляется за счет свойства сервиса **EventConstants.EVENT\_TOPIC.** Обычно имя топика строится из наименования класса сообщения, заменяя «.» на «/». Лучше подписаться на несколько типиков, перечисляя через «,».
      * В наименовании можно использовать «\*» в иерархическом стиле, то есть «"manning/osgi/TravelAgent/\*» НЕ соответствует
        + «manning/osgi/TravelAgent/HotelEvent/USA"
        + "manning/osgi/TravelAgent"
      * Подписка на сообщения из ВСЕХ топиков это «\*»
    - Фильтрация сообщений по свойствам осуществляется за счет свойства сервиса **EventConstants.EVENT\_FILTER.** Синтаксис значения LDAP, например «(price<100)»
  + Получатель должен реализовывать интерфейс с функцией получения и обработки сообщения. Каждое сообщение содержит идентификатор топика и коллекцию именованных свойств

**public void handleEvent(Event event) {**

**System.out.println("Received event on topic = " +event.getTopic());**

**for (String propertyName : event.getPropertyNames()) {**

**System.out.println("\t" + propertyName + " = " + event.getProperty(propertyName));**

**}**

**}**

* В качестве сообщений не используются прикладные классы чтобы обеспечить универсальность, а также возможность фильтрации по значениям свойств.
  + Таким образом, имеем слабый контракт между публикатором и получателем на обязательность наличия в сообщении каких-либо именованных средств
  + Хорошая идея – определить наименования свойств как константы в некотором интерфейсе. Это НЕ для связывания через интерфейс отправителя и получателя, а для документирования
* Стандартные свойства событий
  + Информация о bundle источнике сообщения (**BUNDLE\_SIGNER String | Collection <String>, BUNDLE\_VERSION String, BUNDLE\_SYMBOLICNAME String)**
  + **EVENT Object**
  + Описание возникшего исключения (**EXCEPTION Throwable, EXCEPTION\_MESSAGE String, EXCEPTION\_CLASS String)**
  + **MESSAGE String**
  + Описание сервиса, добавляемого или изменяемого (**SERVICE ServiceReference, SERVICE\_ID Long, SERVICE\_OBJECTCLASS String [ ], SERVICE\_PID String | Collection <String>, TIMESTAMP Long)**

## Публикация сообщений

* Публикатор сообщения должен посредством активатора получить доступ к сервису EventAdmin
  + Асинхронная отправка:

**Dictionary payload = new Properties();**

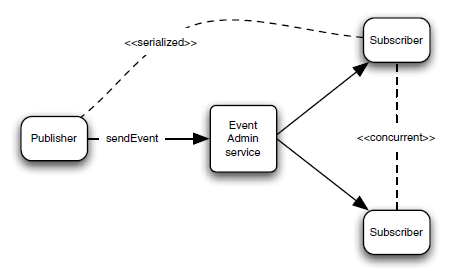
**payload.put("userid", userid);**

**payload.put("timestamp",System.currentTimeMillis());**

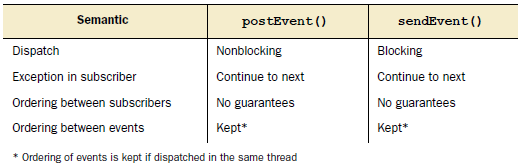
**Event event = new Event("manning/osgi/LoginEvent",payload);**

**admin.postEvent(event);**

* + Синхронная отправка осуществляется при помощи функции **admin.sendEvent(event),** публикатор блокируется пока все получатели не обработают получаемое сообщение. Лучше подобного избегать
    - Так как подписчик не знает как именно ему отправляют сообщение, то он должен как можно быстрее обработать полученное сообщение. А вдруг его послали синхронно?
    - Даже синхронная отправка не гарантирует какой-либо синхронизации между получателями. Например рассылка сообщения может осуществляться в несколько потоков с ожиданием завершения всех потоков



* Синхронная vs асинхронная обработка сообщений



* При рассылке Event Admin делает снимок набора подписчиков, которые должны получить сообщение. Соответственно, если подписчик зарегистрировался после публикации события, то он его не получит. Непосредственно перед отсылкой проводится проверка, что получатель еще существует
* Если получатель сообщения при обработке возбуждает исключение, то оно логируется и продолжается рассылка. НЕТ возможности сообщить публикатору о результате обработки полученного от него сообщения

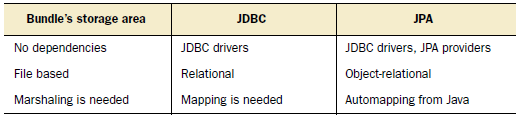
## Недостатки

* потеря сообщений при крахе или перезагрузке системы. Они хранятся только в памяти
* сложная фильтрация (Complex Event Processing)
  + по свойствам объектов = значений свойств. Соответственно приходится писать много кода, заполняющего свойства сообщения
  + групповая обработка сообщений, например, сообщение с наибольшим значением какого-то свойства
  + отбрасывание дублей
* не поддерживаются удаленные клиенты
* случайность порядка обработки сообщения получателями, невозможность определить порядок обработки и передавать информацию между обработчиками

## Стандартные события OSGi

* можно регистрировать слушателя через семейство функций **bundleContext.addXXXListener()** и де-регистрировать через **bundleContext.removeXXXListener().** В процессе остановки bundle фреймворк де-регистрирует их самостоятельно, можно ручками это не делать
* Framework changes – изменения boot classpath, обновления пакетов, платформенные исключения, остановка платформы.
  + Получатель реализует **FrameworkListener**
  + получает тип события, исключение, bundle источник сообщения. Сама OSGi платформа здесь представлена как системный bundle
  + можно подписаться на топик **org/osgi/framework/FrameworkEvent/<framework-event-type>**, где типы это
    - **STARTED**
    - **ERROR –** ошибки, требующие реагирования, посредством данных событий можно, например, слушать ошибки конфигурирования
    - **PACKAGES\_REFRESHED,**
    - **STARTLEVEL\_CHANGED**
    - **WARNING –** предупреждения некритичного характера, потенциальные ошибки
    - **INFO –** информация общего характера по поводу происходящего
* Bundle changes – инсталляция / удаление, старт / остановка
  + Получатель реализует **BundleListener**
  + Получает тип события, bundle источник сообщения как Bundle
  + Можно подписаться на топик **org/osgi/framework/BundleEvent/<bundle-event-type>,** где типы это **INSTALLED, STARTED, STOPPED, UPDATED, UNINSTALLED, RESOLVED, UNRESOLVED.**
* Service changes – регистрация / деригистрация / модификация сервиса
  + Получатель реализует **ServiceListener**
  + Получает тип события, источник сообщения как **ServiceReference**
  + Сообщения в общем случае рассылаются асинхронно, получатель может реализовать **SynchronousBundleListener**, чтобы гарантированно обработать уведомление до того, как событие его вызвавшее закончится.
    - Синхронные слушатели отрабатывают перед обычными слушателями
    - Например, перед удалением bundle успеть очистить ресурсы, связанные с ним
    - Сообщения **BundleEvent.STARTING** и **BundleEvent.STOPPING** рассылаются ТОЛЬКО синхронным слушателям
  + Можно подписаться на топик **org/osgi/framework/ServiceEvent/<service-event-type>** где типы это **REGISTERED, UNREGISTERED, UNREGISTERING**
* Платформа сначала делает снимок слушателей получателей, а потом начинает рассылку сообщений. Соответственно событие получат только те слушатели, которые успели к моменту его возникновения и еще не был удалены к моменту получения
* Уведомления об одном событии подписчики получают конкурентно. Поэтому стоит обращать внимание на потенциальные проблемы с многопоточностью. Уведомления от последовательных событиях подписчики получают строго последовательно. После того, как уведомление разослано, оно более не хранится.

# Персистентность



## Read-Only свойства

* System.getProperty() метод слишком глобальный, поэтому рекомендуется пользоваться BundleContext.getProperyt(), который позволяет получить значения свойств в контексте определенного экземпляра фрейморка. Как именно задаются значения подобных свойств, определяется реализацией фрейморка
* Стандартные свойства
  + **org.osgi.framework.version** - OSGi framework version
  + **org.osgi.framework.vendor** - Framework implementation vendor
  + **org.osgi.framework.language** - Language being used; see ISO 639 for possible values
  + **org.osgi.framework.os.name** - Host computer operating system
  + **org.osgi.framework.os.version** - Host computer operating system version number
  + **org.osgi.framework.processor -** Host computer processor name

## Локальная файловая область bundle

* Пример:

**File eventLog = bundleContext.getDataFile("event-" + (counter++) + ".dat");**

**eventLog.createNewFile();**

**BufferedWriter writer = new BufferedWriter(new FileWriter(eventLog));**

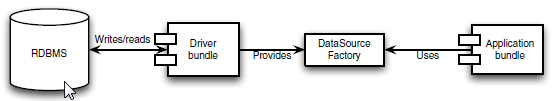
* **BundleContext.getDataFile()** может вернуть null, если реализация OSGi на данной платформе или устройстве не поддерживает подобный тип хранилища.
* Недостатки:
  + сложно определить какой именно файл надо читать / писать в той или иной ситуации
  + требуется обеспечить возможность миграцию формата файла вперед и назад (что самое неприятное – не понятно с какого формата мигрировать), когда версия инсталлированного bundle увеличивается или уменьшается (upgrade / downgrade).
* Достоинства:
  + bundle не может определить, что он переходит в состояние UNINSTALLED и почистить хранилище за собой. Файловое хранилище для деинсталлируемого bundle фреймоворк должен очистить самостоятельно
  + фреймоворк автоматически предоставляет bundle соответствующие права на работу с объектами в файловой системе

## Preferences

* Так же как и предыдущий вариант тоже предназначены для хранения внутренних данных bundle, к которым есть только внутренний доступ (изменить снаружи невозможно), но имеет следующие плюсы
  + Стандартные методы для работы с типизированными значениями
  + Иерархическая система хранения
  + Разделение хранимых значений для отдельных пользователей
  + Не требует наличия файловой системы, механизм хранения абстрагируется контейнером
  + Возвращает дефолтные значения вместо исключений если хранилище данных недоступно
  + Работает даже на Java 1.3, где системного механизма Preferences нет в принципе
* С точки зрения bundle хранилище выглядит как несколько деревьев, одно системное и по одному для пользователя. Можно навигироваться как перебирая для каждого узла дочерние, так и задавая маршрут с использованием «/». Каждый именованный лист хранит коллекцию именованных свойств, значения которых должны быть представимы в виде String. Имеется поддержка для чтения / записи примитивных типов.
* Это приватное хранилище bundle, отсутствует механизм доступа снаружи, только если сам bundle передаст кому-то свой экземпляр Preferences
* Чтобы работать с реестром bundle должен получить экземпляр сервиса PreferencesService
  + Preferences getSystemPreferences();
  + Preferences getUserPreferences(String name);
  + String[] getUsers();
* Каждый элемент реестра описывается Preferences и содержит следующие типовые методы для хранения различных типов данных
  + public void put(String key, String value);
  + public void putInt(String key, int value);
  + public void putLong(String key, long value);
  + public void putBoolean(String key, boolean value);
  + public void putFloat(String key, float value);
  + public void putDouble(String key, double value);
  + public void putByteArray(String key, byte[] value);
* Соответствующие get методы объекта Preferences параметризуются кроме имени также дефолтным значением

## JDBC

* Стандартный подход: **Class.forName + DriverManager.getConnection**. Не подходит для OSGi
  + Клиентский bundle вынужден импортировать классы JDBC драйвера, хотя в принципе может работать с любой базой
  + Так как каждый клиентский bundle грузит классы JDBC самостоятельно, а метод **DriverManager.getConnection** статический и также вызывается в контексте bundle, то при передаче объектов между ними будут ошибки приведения
  + Невозможность совместного использования источников данных и соединения ведет к низкой масштабируемости решения
* **DriverManager** выполняет роль реестра соединений, но в OSGi один реестр уже есть. Потому вместо статического метода **DriverManager** надо использовать сервис **org.osgi.service.jdbc.DataSourceFactory**, возвращающий фабрику источников данных. В результате
  + JDBC специфические классы грузятся в этот сервис, а клиентский bundle оказывается зависим только от JDBC интерфейсов.
  + Можем динамически добавлять и удалять JDBC драйвера. Клиент указывает нужный ему драйвер через свойство **OSGI\_JDBC\_DRIVER\_CLASS**
  + Можно использовать различные версии JDBC драйверов. Клиент указывает нужную ему версию драйвера через свойство **OSGI\_JDBC\_DRIVER\_VERSION**
* Провайдер JDBC драйвера должен упаковать драйвер в bundle и зарегистрировать реализацию **DataSourceFactory** в реестр, снабдив ее соответствующими свойствами



* Потребитель JDBC драйвера

**ServiceReference [] serviceReferences = bundleContext.getServiceReferences(**

**DataSourceFactory.class.toString(),"(" + DataSourceFactory.OSGI\_JDBC\_DRIVER\_CLASS +**

**"=org.apache.derby.jdbc.EmbeddedDriver)");**

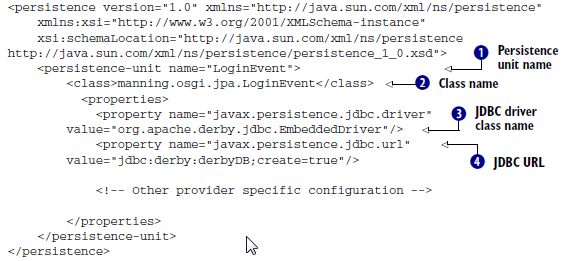
**Properties props = new Properties();**

**props.put(DataSourceFactory.JDBC\_URL,"jdbc:derby:derbyDB;create=true");**

**DataSource ds = dsf.createDataSource(props);**

## JPA

* Конфигурирование
  + JPA конфигурация организована в виде набора юнитов Persistence Unit (PU). В нашем случае логично иметь одни юнит на bundle и использовать в качестве идентификатора юнита символическое имя bundle



* + В JEE окружении мы должны просто положить XML файл конфигурации как **META-INF/persistence.xml**, в случае OSGi этот файл нужно описать в заголовке bundle с наименованием **Meta-Persistence.** Таким bundle мы называем персистентные
  + Кроме того надо развернуть в контейнере ряд инфраструктурных bundle: javax.persistence, org.apache.derby, org.eclipse.persistence, org.eclipse.persistence.core, org.eclipse.persistence.jpa
* Использование достаточно трививально
  + Получение **EntityManager**

**ServiceReference [] serviceReferences = bundleContext.getServiceReferences(**

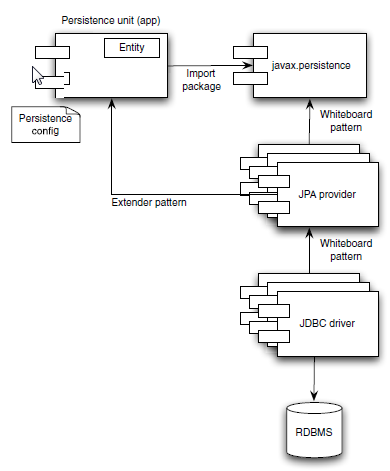
**EntityManagerFactory.class.toString(),"osgi.unit.name=LoginEvent");**

**EntityManagerFactory emf = (EntityManagerFactory)**

**bundleContext.getService(serviceReferences[0]);**

**EntityManager em = emf.createEntityManager();**

* + В JEE есть механизм CMP, избавляющий нас от большинства кода. В OSGi такого стандартного JPA контейнера нет
* Архитектура клиента



* + Javax.persistence bundle является фрейморком, задачей которого являет найти JPA провайдера, на которой возложить всю реальную работу
  + Проблема: классы, которые должны мапиться на таблицы, располагаются в персистентных bundle и не известны JPA провайдера
  + Соответственно JPA провайдер работает как расширитель по отношению к персистентным bundle
    - Регистрирует **BundleListener** и слушает события добавления bundle
    - Для добавляемых персистентных bundle исследует файлы конфигурации и получают перечень персистентных классов
    - За счет ClassLoader персистентного класса получает доступ к классам-сущностям, за счет **DataSourceFactory** получает доступ к источнику данных, инициализирует ORM
    - встраивается в JPA фреймворк посредством регистрации сервиса типа **javax.persistence.spi.PersistenceProvider**. В свойстве сервиса **javax.persistence.provider** указывается класс провайдера.
    - Регистрирует **EntityManagerFactory** сервис для предоставления персистентным bundle
  + Инфраструктура персистентности
    - Javax.persistence bundle ответственен за поиск **PersistenceProvider** сервиса, соответствующего конфигурации персистентного провайдера, заказывающего сервис
    - В результате:
      * Можем динамически управлять JPA провайдерами
      * JPA провайдеры могут использовать различные JDBC драйвера, которыми можно управлять динамически
      * Персистентные bundle не должны импортировать ничего кроме JPA интерфейсов (javax.persistence)

# Транзакции

* В отсутствии транзакций единственная возможность обеспечить подобный эффект – компенсирующие действия, выполняемые в случае неудачи основного бизнес процесса. Недостатки:
  + Компенсирующее действие тоже может навернуться и далее до бесконечности
  + Если бизнес процесс включает больше двух стадий, то логика компенсации становится слишком сложной. Отсутствие масштабируемости
* Локальные транзакции тривиальны за счет **Connection.setAutoCommit(false),** однако если бизнес процесс использует два источника данных, то данный подход не работает и необходимо использовать механизм глобальных транзакций
* Глобальная транзакция в SE

**ServiceReference serviceReference =**

**context.getServiceReference("javax.transaction.TransactionManager");**

**TransactionManager tm = (TransactionManager)context.getService(serviceReference);**

**XADataSource ds1 = ...**

**XADataSource ds2 = ...**

**XAConnection xaConn1 = ds1.getXAConnection();**

**XAConnection xaConn2 = ds2.getXAConnection();**

**tm.begin();**

**Transaction transaction = tm.getTransaction();**

**transaction.enlistResource(xaConn1.getXAResource());**

**transaction.enlistResource(xaConn2.getXAResource());**

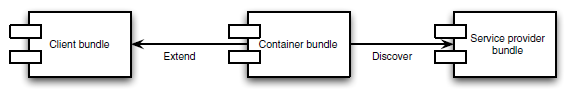
**Connection conn1 = xaConn1.getConnection();**

**Connection conn2 = xaConn2.getConnection();**

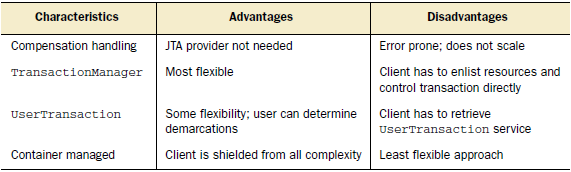
**…**

**tm.commit();**

* + В момент открытия транзакция ассоциируется с текущим потоком выполнения. Ее можно получить за счет **TransactionManager.getTransaction(),** вызываемой из того же потока, в котором транзакция была начата.
  + Не может быть транзакции, распределенной между несколькими потоками, так как при этом невозможно зафиксировать порядок выполнения операций
  + **Commit** вызываем для транзакции, а уж менеджер транзакций в свою очередь запрашивает коммиты локальных транзакций на отдельных ресурсах по двухфазному протоколу
* Глобальная транзакция в контейнере
  + Должен быть единственный JTA провайдер, предоставляющий **TransactionManager** и **UserTransaction** сервисы. В качестве провайдера можно, например, использовать Java Open Transaction Manager 2.0.10.
  + Клиенты запрашивают данные сервисы через OSGi реестр по интерфейсу. Кроме того, они должны определить свою зависимость от пакета avax.transaction
  + Чтобы более гибко управлять транзакцией можно использовать сервис **UserTransaction**,позволяющий на время приостанавливать транзакцию, чтобы чередовать транзакционные и нетранзакционные куски кода в рамках одного бизнес процесса
* Работа с глобальными транзакциями в OSGi требует большого количества инфраструктурного кода, тесно переплетенного с бизнес кодом. Можно оптимизировать за счет специального инфраструктурного bundle
  + Определяем аннотации @Resource и @Transaction, которыми помечаем методы класса, соответственно предоставляющий ресурсы, которые должны быть включены в транзакцию, и бизнес логику, которая должна быть выполнена в рамках транзакции
  + Bundle, выполняющий транзакционную операцию, должен в своем заголовке определить наименование класса с транзакционной операцией
  + Инфраструктурный bundle реализует шаблон «расширитель»
    - Регистрирует слушателя на событий загрузки bundle
    - Отлавливает bundle, которые сообщают о классе с транзакционной операцией.
      * Можно конечно исследовать все классы всех bundle на наличие соответствующих аннотаций, но слишком тяжело. Кроме того это слишком сильно бы связало инфраструктурный bundle с клиентским
    - Создают экземпляр класса, начинают транзакцию
    - получают при помощи методов, аннотированных @Resource, транзакционные ресурсы, включают их в транзакцию,
    - вызывают метод, аннотированный @Transaction, если все нормально комитят транзакцию, если нет – откатывают
  + Итого: есть клиентский bundle с прикладным кодом, инфраструктурный с единой логикой обработки транзакций, и bundle с JTA провайдером и реализацией менеджера транзакций. Точки связывания – конфигурационный заголовок в клиентском bundle и JTA интерфейс



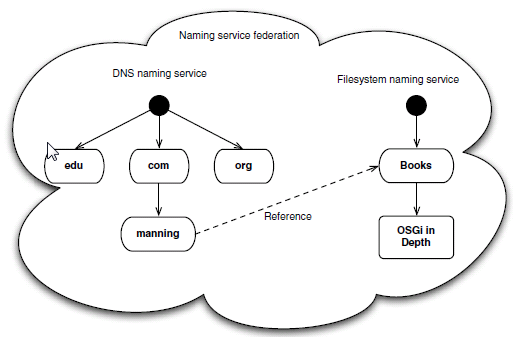
* + Все вместе это пример шаблона «контейнер приложения»
* Сравнение возможных подходов



# JNDI

## Концепция

* При запуске OSGi приложений в JEE контейнере насущная проблема – как сделать ресурсы на базе этих двух инфраструктур взаимно видными
* JNDI – инфраструктура, которая позволяет клиенту совместно использовать набор сервисов имен и сервисов директорий. Различные провайдеры имен / директорий подключаются как плагины, поставляемое ими содержимое образуют единое пространство.
  + Сервис имен – ассоциирует наименования с объектами, пример DNS. Имена иерархические, каждое состоит из отдельных компонент(узлов), разделенных «/». Имя более чем из одной компоненты – compound
    - Можно добавлять специальные плагины-источники, поддерживающие отдельные пространства имен, то есть иметь не единое дерево, а набор деревьев, каждое из которых поддерживается своим источником
    - Можно к узлу подключать ссылку на отдельное поддерево (имен или директорий) – федеративный сервис.
      * Как правило, ссылки служат для смены механизма доступа к узлам при перехода из одной стратегии наименования к другой, например ведут в удаленный реестр или файловую систему.
      * Например: в пути [www.manning.com/Books/OSGi-InDepth.pdf](http://www.manning.com/Books/OSGi-InDepth.pdf) (composite наименование) есть два фрагмента
        + [www.manning.com](http://www.manning.com) представляется DNS сервисом имен
        + Books/OSGi-InDepth.pdf представляется соответственно директорией и файлом



* + Сервис директорий – позволяет ассоциировать с объектом коллекцию именованных атрибутов и осуществлять поиск объекта по условию над атрибутами, например, используя LDAP синтаксис

## Элементарные операции

* Пример доступа к сервису имен

**InitialContext context = new InitialContext();**

**Context subContext = (Context) context.lookup("jdbc");**

**DataSource ds = (DataSource) subContext.lookup("AccountDS");**

**..**

**DataSource ds = (DataSource) context.lookup("comp/env/jdbc/AccountDS");**

* + Можно переходить от контекста к субконтексту, а можно сразу воспользоваться compound именем
  + В JEE окружении как правило все объекты лежать в пространстве имен “java:”
* Пример поиска а сервисе директорий

**InitialDirContext context = new InitialDirContext();**

**BasicAttributes attrs = new BasicAttributes();**

**attrs.put(new BasicAttribute("ver", "1.1"));**

**DataSource ds = (DataSource) context.search("java:comp/env/jdbc/AccountDS",attrs);**

* + Вместо **InitialContext** надо использовать **InitialDirContext**
  + Если использовать **SearchControl**, то можно задавать условие в качестве строки, а также управлять опциями поиска, такими как максимальное время поиска, максимальное число получаемых результатов и так далее

**SearchControls control = new SearchControls();**

**DataSource ds = (DataSource) context.search(**

**"java:comp/env/jdbc/AccountDS", "(ver=1.1)", control);**

* + Условия формируются по-разному в зависимости от того, куда смотрит **InitialDirContext**, в данном случае предполагается, что он смотрит в LDAP. Могут быть различные варианты
* Добавление объекта в дерево

**DataSource ds = ...**

**InitialContext context = new InitialContext();**

**context.bind("java:comp/env/jdbc/AccountDS", ds);**

* Между JNDI и OSGi реестрами очень много общего, есть кардинальное различие:
  + один иерархический, второй плоский (не смотря на кажущуюся иерархичность имен). В JNDI в отличии от OSGi можно получить список детей для каждого узла

**NamingEnumeration<Binding> bindings =**

**context.listBindings("java:comp/env/jdbc");**

**while (bindings.hasMore()) {**

**Binding bd = (Binding) bindings.next();**

**System.out.println("Name = " + bd.getName() + ", Object = " + bd.getObject());**

**}**

* + OSGi реестр монолитный, JNDI поддерживает федеративность
* Федеративность JNDI сервиса
  + Реализуется путем подключения ссылки (Reference) при переходе через очередной разделитель компонент имени «/». Далее Reference конвертируется в реальный объект при помощи одной из фабрик **ObjectFactory**
  + Далеко не всегда можно из узла дерева вернуть реальный объект, например, если он живет на отдельной машине или в отдельном процессе. Ну или он вообще слишком большой, чтобы хранить его в дереве. В этом случае
    - Reference хранит информацию о реальном местоположении объекта в виде коллекции **RefAddr**. Множество членом коллекции используется для поддержки отказоустойчивости сервиса. Например, сервер приложений использует данный механизм для отказоустойчивости RMI объектов
    - Reference хранит наименование класса, реализующего **ObjectFactory**, который может создать локальный прокси к удаленному объекту. Пример – **DirObjectFactory**
  + Продолжая пример с наименованием книги
    - Создание федеративной ссылки на «Books» объект

**manningContext.bind("Books",**

**new Reference(Context.class.toString(),**

**"com.manning.FSObjectFactory", null));**

* + - Класс **FSObjectFactory** ответственен за создание нового Context объекта, который ответственен за подключение к удаленному сервису имен, который располагается на отдельной машине. Его задача - конвертировать адрес машины, взятый из предыдущего фрагмента наименования, с дальнейшим путем в удаленной файловой системе в локальный объект, представляющий доступ к удаленному файлу
  + **ObjectFactory** может также применяться не только для федеративного перехода, но и для создания адаптирующей обертки вокруг объекта

## Инициализация контекста

* **InitialContext** это не только ссылка но рутовый элемент иерархии имен, но и ссылку на поставщика сервиса. Вопрос: как этого поставщика правильно определить в имеющемся окружении
* SE
  + Пример

**Hashtable env = new Hashtable();**

**env.put(Context.INITIAL\_CONTEXT\_FACTORY,**

**"com.sun.jndi.fscontext.RefFSContextFactory");**

**InitialContext context = new InitialContext(env);**

* + Класс, ответственный за создание локального контекста, подключенного к какому-то сервис-провайдеру (файловая система или LDAP) в простейшем случае указывается в переменной **INITIAL\_CONTEXT\_FACTORY**. Если параметр не указан, то используется переменная окружения **java.naming.factory.initial.** Пользователь может переопределить данную логику посредством регистрации своего кода в **NamingManager**
  + Таким образом инициализация происходит следующим образом
    - Пользователь создает экземпляр **InitialContext**
    - Конструктор **InitialContext** обращается к статическому методу **NamingManager.getInitialCOntext**
      * Если свойство менеджера **InitialContextFactoryBuilder** не установлено, то используются значений вышеуказанных свойств
      * Если установлено, то вызывается **InitialContextFactoryBuilder.createInitialContextFactory(),** как этот метод работает не определено – точка расширения платформы и возможность для пользователя переопределить дефолтное поведение платформы
    - Вызов **InitialContextFactory.getInitialContext()** завершает создание рутового контекста
  + В OSGi окружении с таким подходом будут проблемы
    - Клиентский bundle будет должен импортировать все нужные классы, так как они должны будут быть загружены его загрузчиком
    - Вызов статического метода означает, что отрубается возможность динамической смены реализации сервиса. Нарушается принцип модульности.
    - Невозможно одновременно использовать множественные провайдеры JNDI сервиса
* OSGi
  + Как и раньше вместо статического метода используем синглетон **JNDIContextManager**, который заменит собой **NamingManager.getInitialContext** и будет искать в реестре зарегистрированную реализацию необходимого сервиса:

**ServiceReference ref = bundleContext.getServiceReference(**

**JNDIContextManager.class.toString());**

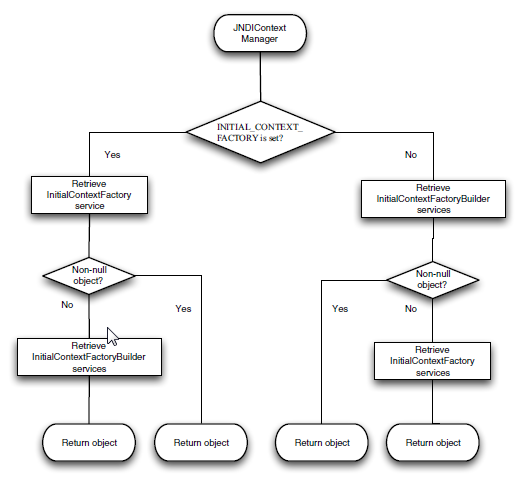
**JNDIContextManager ctxtMgr = (JNDIContextManager) bundleContext.getService(ref);**

**Hashtable env = new Hashtable();**

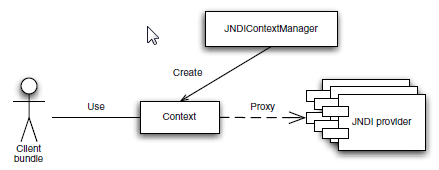
**env.put(Context.INITIAL\_CONTEXT\_FACTORY,"com.sun.jndi.fscontext.RefFSContextFactory");**

**Context context = ctxtMgr.newInitialContext(env);**

* + - **newInitialContext** возвращает **Context** - интерфейс, а не **InitialContext** – класс реализации, чтобы избежать избыточного связывания
    - соответственно, если хотим сервис директорий, то надо использовать не **newInitialContext**, а **newInitialDirContext**
  + каким образом **JNDIContextManager** должен искать реализацию провайдера, поставляемую **InitialContextFactory**:
    - очевидно он должен быть как сервис зарегистрирован в реестре OSGi и его имя должно быть упомянуто в свойстве **INITIAL\_CONTEXT\_FACTORY.** Или же аналогичным образом должен быть использоваться сервис типа **InitialContextFactoryBuilder**, который нужную породит **InitialContextFactory**
    - последовательность действий
      * проверяем значение системного свойства **INITIAL\_CONTEXT\_FACTORY**
      * если оно установлено, то ищем в реестре реализацию **InitialContextFactory**, реализованную классом, упомянутом в **INITIAL\_CONTEXT\_FACTORY**
        + если нашли, то используем для получения начального контекста
        + если нет, то ищем в реестре реализации **InitialContextFactoryBuilder**, перебираем в порядке приоритетности, берем первую, которая вернет не null результат, используем **InitialContextFactoryBuilder**. **createInitialContextFactory()** для получения начального контекста
      * если не установлено, то действуем в обратном порядке: ищем в реестре реализации **InitialContextFactoryBuilder**
        + если нашли, то производим им фабрику и начальный контекст
        + если не нашли, то ищем в реестре реализации **InitialContextFactory**, перебираем в порядке приоритетности, ищем первую которая вернет не null результат, используем его в качестве начального контекста

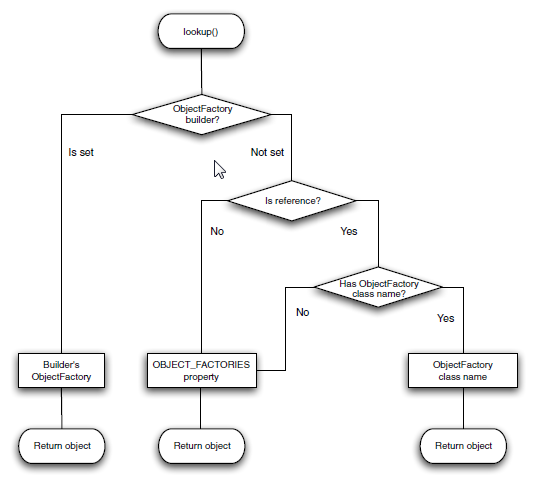


* + Производимый таким образом начальный контекст являет прокси, который позволяет за сценой подменять реализации сервиса, незаметно для клиента

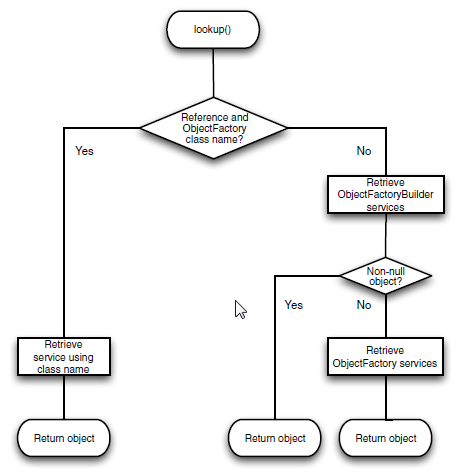


## Преобразования объектов

* **Reference** используются не только для организации федеративного дерева, но и для преобразования зарегистрированных в дереве объектов перед выдачей их клиенту. Например, оборачивание proxy или переконфигурирование на ходу.
  + На самом деле JNDI для любого запрошенного объекта пытается найти подходящую **ObjectFactory**, которая его обработает. И только если все найденные **ObjectFactory** для объекта вернули null, отказавшись его преобразовывать, объект возвращается клиенту как есть. В противном случае возвращается результат работы фабрики
  + Мощный механизм реализации инфраструктурных сервисов, прозрачно оборачивающих публикуемые для пользователей объекты
* Процесс преобразования в SE
  + Клиент вызывает **Context.lookup(name),** объект ищется в дереве, перед выдачей клиенту обрабатывается за счет статического **NamingManager.getObjectInstance()**
  + Проверяется **NamingManager.getObjectFactoryBuilder()**
    - Если билдер установлен, то производится **ObjectFactory**, которой обрабатываем объект. Билдер НЕ может вернуть null вместо фабрики
    - Если билдер НЕ установлен, то проверяется является ли объект Reference
      * Если нет, то проверяем значение свойства **Context.OBJECT\_FACTORIES**, которое содержит список классов зарегистрированных фабрик.
        + Перебираем все фабрики, пытаемся каждую применить.
        + Если какая-то при этом возбудит исключение – возвращаем его вызывающему коду
      * Если да, то проверяем определила ли ссылка наименование класса **ObjectFactory**
        + Если определила используем ее
        + Если не определила, то опять-таки используем значение системного свойства **Context.OBJECT\_FACTORIES**



* + Опять-таки имеем статические методы, непригодные для OSGi окружения
* Процесс преобразования в OSGi
  + **Context.lookup(name)** за конвертацией найденного объекта обращается к **JNDIProviderAdmin.getObjectInstance()**
    - Если объект **Reference** и у него определено наименование класса фабрики **ObjectFactory**, то **JNDIProviderAdmin** лезет в реестр за соответствующим сервисом, параметризовав запрос наименованием класса
    - Если наименование фабрики не известно, то в реестре ищутся реализации **ObjectFactoryBuilder**
      * Если найдены, то перебираем их, от каждого получаем фабрику и пытаемся сконвертировать объект
      * Если не найдены, то ищем реализации **ObjectFactory** и пытаемся применить уже их



* Пример, попытка сконфигурировать **DataSource** перед передачей запросившему клиенту
  + Пишем активатор, который реализует **ObjectFactory**, регистрируем его в контексте
  + Реализуем метод конвертации

**public Object getObjectInstance(Object obj, Name name, Context nameCtx,**

**Hashtable<?, ?> environment) throws Exception {**

**Object ret = null;**

**if (nameCtx.getNameInNamespace().equals("comp/env/jdbc")) {**

**if (obj instanceof DataSource) {**

**DataSource jeeDS = (DataSource) obj;**

**// обрабатываем объект**

**ret = jeeDS;**

**}**

**}**

**return ret;**

**}**

* + Если не можем сконвертировать, то важно вернуть null, чтобы дать отработать другим фабрикам. Если вернуть сам объект, то он немедленно улетит пользователю как результат успешной конвертации

## Пространства имен

* Варианты подключения OSGi сервисов в JNDI контекст
  + Можно просто регистрировать сервисы и там и там, но это двойная работа и необходимость синхронизации двух деревьев
  + Можно сделать федеративное дерево, но тогда пути будут выглядеть уродливо, например **comp/env/services/javax.sql.DataSource/AccountDS**
  + Лучший вариант – отдельное пространство имен для OSGi сервисов, типа **osgi:service/javax.sql.DataSource,** тогда можно иметь достаточно расширенный синтаксис запроса сервисов:

**context.lookup("osgi:service/javax.sql.DataSource/(ver=1.1)");**

**NamingEnumeration<Binding> bindings =**

**context.listBindings("osgi:servicelist/javax.sql.DataSource");**

**while (bindings.hasMore()) {**

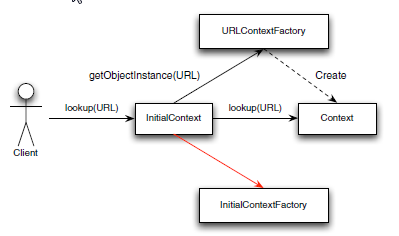
**Binding bd = (Binding) bindings.next();**

**System.out.println("Name = " + bd.getName() + "Class = " + bd.getClassName() +**

**", Object = " + bd.getObject());**

**}**

* Поддержка пространства имен в INDI позволяет иметь не единое дерево, а набор деревьев, по дереву на пространство имен. Начальный контекст создается как прокси, указывающий на конкретное дерево, за счет анализа префикса URL и вызова соответствующей фабрики контекста.



* В SE
  + Исследуем значения свойства **Context.URL\_PKG\_PREFIXES,** которое содержит перечень пакетов для потенциальных фабрик контекстов пространств имен
    - Для каждого пакета ищется класс **package.scheme.scheme + URLContextFactory**, где **scheme** – префикс пространства имен
    - Для каждого найденного класса, используем его метод **URLContextFactory.getObjectInstance(url),** если он возвращает не null, то это и есть контекст
  + Значение свойства **INITIAL\_CONTEXT\_FACTORY** на этом этапе игнорируется и используется только в том случае, когда ни одна фабрика не распознала пространство имен предложенного URL
  + Если **InitialContextFactoryBuilder** установлен, то данный процесс пропускается целиком и полностью отдается на откуп билдеру – можно кастомизировать как угодно
* В OSGi
  + Поступаем все по тому же принципу: фабрики контекстов должны регистрироваться в реестре как **ObjectFactory** сервисы, описываемые свойством **osgi.jndi.url.scheme** установленным в префикс схемы, которую обслуживает фабрика
  + **JNDIContextManager**, получив URL, сначала пытается создать начальный контекст путем поиска соответствующей фабрики и только потом начинает обычную операцию. Свойству **Context.URL\_PKG\_PREFIXES** в SE соответствует реестр с описаниями потенциально применимых фабрик.
  + Если прописан кастомный **InitialContextFactoryBuilder**, то все передоверяется ему

# Развертывание приложения

## Загрузка и классы

* Так как OSGi это более новая и гибкая платформа, то рассматриваем ее запуск поверх уже имеющихся платформ – embedding OSGi
* SE запуск

**ServiceLoader<FrameworkFactory> services = ServiceLoader.load(FrameworkFactory.class);**

**FrameworkFactory frameworkFactory = services.iterator().next();**

**Framework osgiFrw = frameworkFactory.newFramework(null);**

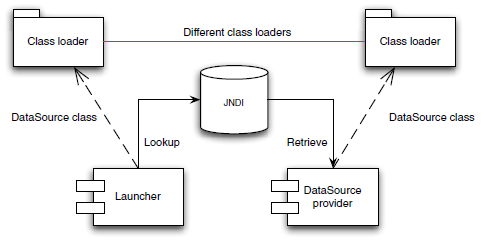
**osgiFrw.start();**

**//...**

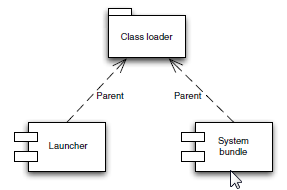
**osgiFrw.stop();**

**osgiFrw.waitForStop(10000);**

* + Загрузка платформы реализуется **FrameworkFactory**, которая ищет реализацию в ресурсе **/META-INF/services/org.osgi.framework.launch.FrameworkFactory** JAR файла имплементации
  + Framework – наследник Bundle и представляет системный Bundle
  + Остановка платформы – асинхронный процесс, поэтому придется подождать
* Основная проблема как всегда с загрузкой классов
  + Один и тот же класс, например интерфейс, может быть загружен как родительским загрузчиком классов JRE, так и загрузчиком классов какого-то bundle. Если они встретятся, то будет проблема



* + Соответственно надо добиться, чтобы подобные классы грузились только один раз и из одного источника. Надо чтобы, основной загрузчик, используемый для запуска платформы, грузил бы их, а сами bundle пользовались уже загруженными классами.



* + Это достигается за счет системного свойства **org.osgi.framework.system.packages.extra,** содержащего перечень пакетов, классы которых системный bundle платформы будет получать у рутового загрузчика. Обычно значение данного свойства java.\*
    - Пример использования

**Map properties = new HashMap();**

**properties.put("org.osgi.framework.system.packages.extra","javax.sql");**

**Framework osgiFrw = frameworkFactory.newFramework(properties);**

* + - То, что класс упомянут в org.osgi.framework.system.packages.extra не избавляет bundle от необходимости определять их импорт через заголовки манифеста
    - Отличие от свойства **org.osgi.framework.bootdelegation** – классы перечисленных в данном свойстве пакетов становятся автоматически доступными всем bundle платформы без необходимости определять их импорт. Это довольно опасное свойство, которым не стоит злоупотреблять

## Доступ через JNDI

* Чтобы сервисами OSGi могли пользоваться не только bundle платформы, надо чтобы они оказались опубликованными в JNDI. Для этого сначала опубликуем OSGi контекст в JNDI
  + Делаем за счет свойства **osgi.service.jndi.bundleContext**

**Framework osgiFrw = frameworkFactory.newFramework(properties);**

**osgiFrw.start();**

**Hashtable env = new Hashtable();**

**env.put("osgi.service.jndi.bundleContext",osgiFrw.getBundleContext());**

**InitialContext ic = new InitialContext(env);**

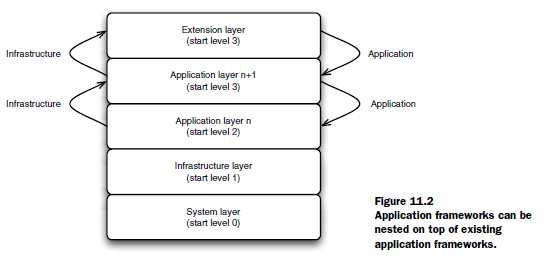
* + Затем определим фабрику для работы с URL через системное свойство **Context.URL\_PKG\_PREFIXES**
  + Теперь можно пользоваться контекстом

**DataSource ds = (DataSource) ic.lookup("osgi:service/javax.sql.DataSource");**

* + Теперь можно

## Уровни исполнения

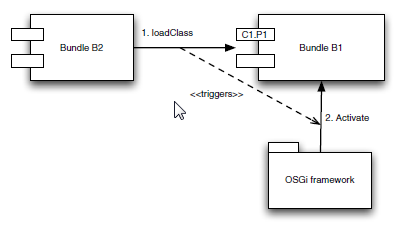
* Проблема – пользователь установил bundle, который зависает в процессе инициализации, вечно **STARTING**.
  + Причины: deadlock в процессе инициализации, попытка вызова не отвечающего удаленного сервиса без timeout
  + Процесс консоли управления блокируется, нет возможности его выгрузить, при силовой перезагрузке платформа опять пытается его запустить и все повторяется
  + Выходы
    - Убить из кэша JAR
    - Разделить процесс старта на этапы, чтобы получить возможность при перезагрузке выгрузить сбойный bundle до того, как он будет запущен
* Этапы загрузки платформы.
  + Основная цель: привести все bundle к тому же сохраненному состоянию, в котором они были перед запуском процесса перезагрузки.
  + Все bundle, которые были **INSTALLED**, инсталлируются заново
  + Все bundle,которые были **RESOLVED**, заново проходят эту операцию. Чтобы данный этап был выполнен, они обрабатываются в порядке их инсталляции
  + Все bundle,которые были **RESOLVED**, запускаются, для каждого вызывается **BundleActivator.start()**
  + Всем bundle рассылается сообщение **FrameworkEvent.STARTED,** обозначающее, что они могут начать работу
* Процесс запуска bundles разбит на несколько этапов, каждому из которых присвоен свой номер – уровень запуска. Идея пришла из Unix, в которой все сервисы разбиты на уровни и в соответствии с ним поднимаются / опускаются
* Уровни запуска
  + **Bundle start level** – уровень запуска bundle, определяет уровень запуска платформы при котором, bundle будет запущен, по умолчанию 1, может быть изменен динамически
  + **Initial bundle start level** – уровень запуска, присваиваемый bundle при инсталляции, может быть изменен, действует только на вновь инсталлируемые bundle
  + **Active start level** – уровень запуска платформы, меняется сначала в процессе запуска автоматически, а потом по запросу администратора. По мере его изменения система запускает или останавливает bundle в соответствии с их уровнем запуска
  + **Beginning active start level** – дефолтный уровень запуска платформы, то есть она стартует автоматом до данного уровня, конфигурируется
* Когда мы запускаем bundle, он реально не стартует, но помечается как “to be started”. Это состояние сохраняется, переживает перезагрузку, и когда уровень выполнения достигает порогового bundle наконец запускается
* Соответственно, мы разбиваем все bundle по степени инфраструктурности и надежности, от наиболее надежных, предоставляющих базовые сервисы, к расширениям приложения, которые могут и испортить систему. Enterprise приложения должны иметь КАК МИНИМУМ два дополнительных уровня к двум базовым, представляемым платформой



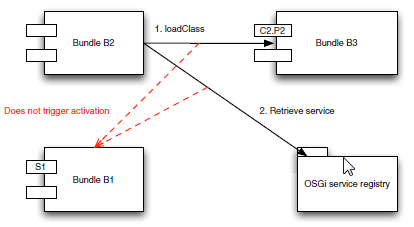
* + 0 – сама платформа, только системный bundle
  + 1 – системные сервисы платформы. Например, Configuration Admin и консоль управления. Возможности необходимые всем без учета специфики развернутого приложения
  + 2 – дополнительные инфраструктурные сервисы, необходимые для приложения. Например, JPA и транзакции. Поступают от доверенных поставщиков
  + 3 – само приложение, разработанное и протестированное, в котором можно быть уверенными.
  + 4 – расширения приложения. Наиболее ненадежная часть системы
* Уровень запуска платформы устанавливаем, к примеру, в 2, затем ручками стартуем приложение. Можно сделать управляющий агент, который проверяет работу уже застартовавших сервисов и продвигает текущий уровень запуска дальше
* По мере продвижения вверх уровни становятся все более специализированными и все менее надежными. По мере продвижения вниз все более абстрактными
* Процесс остановки системы проходит в обратном порядке. Хотя bundle останавливается, его состояние остается ACTIVE, поэтому при старте системы он будет запущен.
* Уровни применяются только для запуска / остановки системы. Их НЕ надо использовать для учета каких-либо зависимостей между bunle.
  + Предполагается, что bundles должны быть так спроектированы, чтобы не зависеть от порядка установки / активации / деактивации.
  + Для этого им предоставляется механизм рассылки сообщений, в ответ на которые они должны получать / прекращать доступ к нужным им ресурсам / сервисам
  + Тем не менее иногда необходимо использовать что-то подобное, например, когда во время старта системы надо выводить сообщения на некоторое стандартное окно, которое очевидно должно быть поднято первым.
  + То есть это опять-таки некоторая форма группировки функциональных возможностей: базис, прикладные процессы и так далее
* Управление уровнями ведется за счет **StartLevel** сервиса, получаемого из репозитария.
  + int getStartLevel();
  + void setStartLevel(int startlevel);
  + int getBundleStartLevel(Bundle bundle);
  + void setBundleStartLevel(Bundle bundle, int startlevel);
  + int getInitialBundleStartLevel();
  + void setInitialBundleStartLevel(int startlevel);
  + boolean isBundlePersistentlyStarted(Bundle bundle);
  + boolean isBundleActivationPolicyUsed(Bundle bundle);
* При изменении активного уровня работы контейнера рассылается сообщение типа FrameworkEvent.STARTLEVEL\_CHANGED

## Опции запуска bundle

* временный старт bundle
  + за счет использования **bundle.start(Bundle.START\_TRANSIENT).**
  + bundle будет запущен, но его состояние останется **RESOLVED**, не будет изменено на **STARTING** / **ACTIVE**, а значит при перезагрузке системы он повторно запущен НЕ будет
  + удобно для тестирования, но не применимо при эксплуатации
* Отложенный старт bundle
  + Решаемые проблемы
    - 10 секунд на старт каждого bundle, а в системе их до сотни, в итоге получаем очень медленный старт системы
    - Bundle экспортирует пакеты, которые не будут функционировать правильно без BundleContext
  + Можно ручками за счет вынесения процесса инициализации в отдельный поток, запускаемый активатором при старте
  + Можно системно за счет заголовка манифеста **Bundle-ActivationPolicy**: lazy
    - в этом случае bundle не будет запущен и **BundleActivator.start()** не будет реально вызван до тех пор, пока bundle не понадобится какому-нибудь другому bundle. Соответственно при этом bundle переходит дальше в **STARTING** и в **STARTED**
    - Процесс запуска bundle не блокируется, но bundle остается в состоянии **STARTING**, вместо того, чтобы перейти в **STARTED**
    - Вместо сообщения BundleEvent типа STARTING рассылается типа LAZY\_ACTIVATION
    - Экономит время запуска платформы и потребляемые ею ресурсы
    - Забавное, но дефолтную «агрессивную» политику активации bundle при помощи заголовка **Bundle-ActivationPolicy** задать нельзя
    - Можно потребовать отложенной активации только по загрузке классов из определенных пакетов: **Bundle-ActivationPolicy: lazy; include:="com.acme.service", exclude:=”….”**
  + «Понадобится» означает, что другой bundle попробует загрузить класс данного при помощи **BundleContext.loadClass()** или в процессе разрешения импортируемых пакетов. Загрузка ресурсов НЕ вызывает активацию bundle



* + Остановка bundle в STARTING состоянии означает возврат в RESOLVED, а уже потом STOPPING и STOPPER
  + Если отложенная активация одного bundle, требует загрузки класса из другого, тоже отложенного и далее, то активация bundle выполняется по цепочке начиная с конца
  + Невозможно программно выполнить отложенную активацию произвольного bundle, не помеченного соответствующим заголовком. В то же время посредством bundle.start() можно запустить отложенный bundle в нормальном режиме. Чтобы запустить его в отложенном режиме надо явно использовать **bundle.start(Bundle.START\_ACTIVATION\_POLICY).** Bundle декларативно не определенный как отложенный в любом случае будет активирован немедленно.
  + Проблема - иногда сложно активировать такой bundle. Например, В1 регистрирует сервис S1 с интерфейсом С2, определенном в пакете В3. В2 запрашивает сервис у реестра, но загружает только интерфейс С2 из В3, а реализация остается не созданной и соответственно не найденной



## Развертывание и обновление

* Обновление приложения
  + Деинсталировать и заново инсталлировать ручками слишком сложно, подходит только для разработки и тестирования. Для эксплуатации нужно
    - Если обновление сломалось – способ возвратить обратно
    - Защититься от кривых рук пользователя и не требовать от администраторов умения общаться с контейнером
  + Имеется **Bundle.update()** API, позволяющее автоматизировать операцию обновления
    - Вызывает **Activator.stop(),** затем инсталлирует новую версию, затем вызывает **Activator.start().** Если что-то из этого накрывается, то возвращает первоначальную реализацию
    - Если какой-то пакет был экспортирован, то он остается экспортирован до рестарта системы в целом
  + Можно, например, написать агента, который при запуске будет обновлять все bundle, старше определенного срока
    - Признак того, что какой-то bundle относится к инфраструктуре, а не к приложению - **bundle.getLocation().startsWith("file:bundle")**
    - Когда инсталлируем bundle всегда надо использовать новый экземпляр объекта Bundle чтобы не попасться на кэширование. Update можно использовать и у старого
  + По умолчанию платформа считает, что обновление происходит из того же файла, из которого был bundle установлен.
    - Это неудобно, ибо предполагается, что его кто-то его там на диске как-то подменил
    - Можно обновляться из **InputStream**
    - Можно через заголовок манифеста **Bundle-UpdateLocation** определить URL, откуда новый файл bundle будет скачан в процессе обновления
    - Последние два способа НЕ перезаписывают исходный файл, поэтому если сделать uninstall / install, то возвращаемся к первоначальной версии, понадобится обновление
* Обновление расширений приложения
  + Можно тем же механизмом, но сложно для обновляющих. Им требуется поддержка трех операций: установить, получить список установленного, удалить
  + Можно предоставить интерфейс
    - Сложный, но более удобный – веб приложение, есть стандартные, например <http://felix.apache.org/site/apache-felix-webconsole.html>
    - Проще – место на диске + агент, который будет следить за добавлением / удалением файлов, список установленного получается автоматически. Реализация – Apache Felix File Install bundle
* Подготовка экземпляра контейнера к авто запуску
  + Удалить все лишние bundle
  + Приложение можно поместить в директорию системных bundle, но это не очень хорошо:
    - Пользователь должен будет командами оперировать над ним
    - Все запуститься на одном уровне
  + Более тесная интеграция, нативная

**felix.auto.start.1= \**

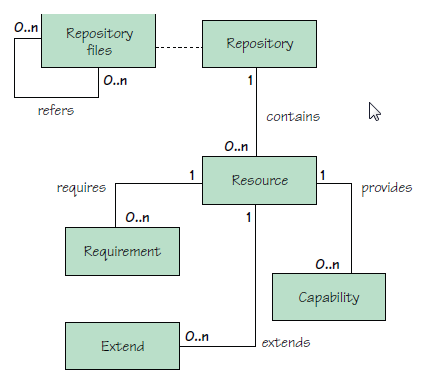
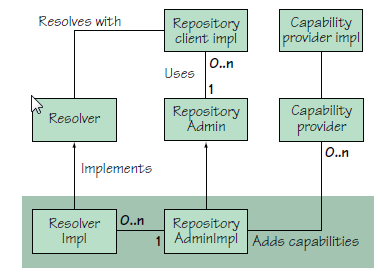
**file:modules/org.apache.felix.fileinstall-2.0.8.jar \**

**file:modules/manning.osgi.mysystembundle\_1.0.0.jar**

* + - Можно будет удалить консоль вообще
    - Установленные файлы не будут видны в стандартной директории
* Каждая реализация предоставляет свой механизм автоматического развертывания приложений, например для Felix это свойства **felix.auto.deploy.action и felix.auto.deploy.dir**. Все bundle в данной директории будут автоматически инсталлированы и запущены при старте контейнера
* Основные подходы к развертыванию сложных приложений, состоящих из множества bundles со множественными зависимостями
  + Написание собственного агента доставки и развертывания bundle
  + OSG Bundle Repository (OBR)
    - фокусируется главным образом на организации удаленного репозитория для хранения индивидуальных bundles, снабжаемых метаданными о зависимостях / ресурсах.
    - Во время процедур Install / update файлы берутся напрямую прямо оттуда с учетом необходимости автоматического разрешения их зависимостей
  + Deployment Admin Service
    - Фокусируется главным образом на развертывании подготовленного разработчиком пакета bundles и связанных ресурсов, составляющих приложение
* Дополнительный инструментарий
  + Ace – средство базирующееся на Deployment Admin, предоставляет центр управления удаленными системами, распределенными программными компонентами, конфигурационными данными и так далее. Удаленные системы обычно основаны на OSGi, но не обязательно
  + Nimble – заточек под управление ресурсами, которые предоставляют какие-то сервисы, но основываются на отличной от OSGi архитектуре. Например, для развертывания сервлетов в составе bundle необходимо развертывать вне OSGi контейнера отдельный сервлетный контейнер
  + p2 – расширение Eclipse Equinox проекта, расширяет список артефактов, которые могут быть развернуты, например Unix RPM пакеты или windows сервисы

## OSGi bundle repository

* стандартной спецификации пока нет, ведется разработка RFC 112, но уже существующие решения вполне могут быть использованы
* предоставляет сервисы
  + централизованное удаленное хранилище bundles и сопутствующих метаданных. Хранилище может ссылаться на другие хранилища – федеративный механизм
  + получение перечня bundle, которые могут быть развернуты из него
  + механизм развертывания bundle и его зависимостей (с учетом транзитивности)
* использует XML файл для описания набора bundle и простую файловую систему хранения репозитария, позволяющую помещать в него bundles без необходимости запуска файлового процесса, просто копированием.
  + Можно создавать файл ручками, но есть средства автоматизации типа BIndex (<http://www.osgi.org/Repository/BIndex>). Хорошо просто добавить его вызов в свой цикл сборки
  + Для пользователей Maven также есть соответствующая поддержка.
* корневые концепции
  + ресурсы – любой артефакт: bundle, конфигурационный файл, сертификат и т.д.
  + requirements / capabilities – соответственно для ресурса требования и удовлетворения требований других ресурсов

* проблема:
  + составляя перечень зависимостей, удовлетворение которых необходимо для развертывания bundle, ORB исходит из алгоритма сходного с алгоритмом разрешения в OSGi, но не идентичным. В частности, ORB не понимает «uses»
  + ORB начинает с некоторого перечня bundle и добавляет зависимости до удовлетворения, OSGi просто проверяет набор установленных bundle на удовлетворение всех зависимостей, не добавляя ничего
* Просмотр репозитория – **RepositoryAdmin**
  + Resource[] discoverResources(String filterExpr);
  + Resolver resolver();
  + Repository addRepository(URL repository) throws Exception;
  + boolean removeRepository(URL repository);
  + Repository[] listRepositories();
  + Resource getResource(String respositoryId);
* Установка bundle из репозитория – **Resolver**
  + void add(Resource resource);
  + Requirement[] getUnsatisfiedRequirements();
  + Resource[] getOptionalResources();
  + Requirement[] getReason(Resource resource);
  + Resource[] getResources(Requirement requirement);
  + Resource[] getRequiredResources();
  + Resource[] getAddedResources();
  + boolean resolve();
  + void deploy(boolean start);
* Типовой образ действия
  + Ищем ресурс для развертывания за счет RepositoryAdmin.discoverResources(), который поддерживает фильтрация на основе LDAP выражения
  + Добавляем ресурс для развертывания за счет Resolver.add()
  + Добавляем его необходимые зависимости за счет Resolver:resolve()
  + Если все зависимости нормально добавились, то запускаем развертывание / старт за счет Resolver.deploy()

## Deployment Admin

* ORB автоматически вычисляет необходимые зависимости и добавляет их к запрошенному на установку bundle.
  + В противовес этому Deployment Admin заточен под развертывание заранее подготовленного разработчиком пакета с bundles и другими необходимыми в процессе развертывания ресурсами
  + Мы оперируем не индивидуальными bundle, а целым набором, составляющим приложение (или его часть)
* Пакет это обычный jar, содержащий другие jar с bundle и метаописания в форме манифеста. Этот пакет передается DeploymentAdmin сервису.
  + Заголовки описания DeploymentPackage-SymbolicName и DeploymentPackage-Version
  + Bundles из него он обрабатывает самостоятельно, а остальные ресурсы соответственно их типам отдает ResourceProcessor сервисам, ответственным за их обработку.
  + Все ResourceProcessor в системе получают сообщения о добавленных / обновленных / удаленных ресурсах.
  + Если все ресурсы обработаны успешно, то DeploymentAdmin коммитит транзакцию. Транзакции поддерживаются не в строгом смысле слова, а поскольку возможно.
* Содержимое пакета расположено последовательно для обеспечения возможности его обработки по мере закачки без необходимости развертывания целиком в памяти: манифест, подписи, файлы локализации, bundles, остальные ресурсы
* Разбиение приложения на отдельные пакеты зависит от стратегии доставки и обновления (все в одном, главный пакет / расширения, все в одном / набор обновлений и так далее). Выбранная стратегия должна поддерживаться в дальнейшем. В дальнейшем перемещение bundles / ресурсов между пакетами не допускается. Каждый ресурс должен содержаться только в одном установочном пакете
* Последовательно устанавливаемые пакеты могут обновлять друг друга при условии соблюдения порядка установки.
  + Обновляющий / source и обновляемый / target ресурсы.
  + Один ссылается на другой через одинаковое символическое имя и диапазон версий, которые могут быть обновлены в заголовке DeploymentPackage-FixPack
  + В обновляющий пакет включаются только измененные bundle, остальные можно не включать, а в манифесте помечать как DeploymentPackage-Missing: true
* Управление пакетами – **DeploymentAdmin**
  + DeploymentPackage installDeploymentPackage(InputStream in) throws DeploymentException;
  + DeploymentPackage[] listDeploymentPackages();
  + DeploymentPackage getDeploymentPackage(String symbName);
  + DeploymentPackage getDeploymentPackage(Bundle bundle);
  + boolean cancel();
  + Удаление пакета – DeploymentPackage.uninstall()
* Обработка ресурсов – **ResourceProcessor**
  + void begin(DeploymentSession session);
  + void process(String name, InputStream stream)
  + throws ResourceProcessorException;
  + void dropped(String resource) throws ResourceProcessorException;
  + void dropAllResources() throws ResourceProcessorException;
  + void prepare() throws ResourceProcessorException;
  + void commit();
  + void rollback();
  + void cancel();
* **DeploymentAdmin** направляет ресурсы из загружаемого пакета соответствующему **ResourceProcessor**, основываясь на содержимом заголовка Resource-Processor, в котором мы определяем PID нужного обработчика.
  + Таким образом мы кастомизируем процесс загрузки / установки пакета. Например, можем XML описание процесса обновления базы данных направить соответствующему обработчику или осуществить конверсию данных.
  + Кастомизирующие сервисы могут уже быть установлены, а могут содержаться в самом пакете. Тогда их bundles помечаются за счет DeploymentPackage-Customizer заголовка и устанавливаются / стартуют первыми
  + Обработка всех ресурсов пакета производится в контексте одного экземпляра DeploymentSession, она не видна клиентов, но связывает все операции обработки всех ресурсов между собой.
  + Если в процессе обработки ресурса возникло исключение, то в меру возможности откатывается весь процесс. Если все нормально, то происходит двух фазное подтверждение процесса путем опроса всех **ResourceProcessor**

# JMX

## Основы

* JMX следует такой же модели, как и SNMP. Специальный java класс (managed bean / MBean) создается для ресурса, которому необходимо внешнее управление. Бин работает как управляющий фасад для этого ресурса и предоставляет метаданные, описывающие доступные операции и их атрибуты
* У каждого MBean есть имя, состоящее из пространства имен и набора свойств: **osgi.core: type=framework,version=1.5**
* В JVM запускается локальный объект, который управляет иерархией бинов и предоставляет к ним доступ через различные протоколы, например RMI
  + До JDK 1.6 его приходилось запускать специально как опции запуска основного процесса

**Java -Dcom.sun.management.jmxremote.port=9999**

**-Dcom.sun.management.jmxremote.authenticate=false**

**-Dcom.sun.management.jmxremote.ssl=false**

**-jar bin/felix.jar**

* + Начиная с JDK 1.6 он запускается по умолчанию, но только для локального доступа, то есть jmxremote свойства необходимо указывать в случае необходимости доступа по сети
* Чтобы подключиться
  + консолью: **$JAVA\_HOME/bin/jconsole localhost:port**
  + программно

**JMXServiceURL url =**

**new JMXServiceURL("service:jmx:rmi:///jndi/rmi://localhost:9999/jmxrmi");**

**JMXConnector connector = JMXConnectorFactory.connect(url, null);**

**MBeanServerConnection msc = connector.getMBeanServerConnection();**

**ObjectName mbeanName = new ObjectName("osgi.core:type=bundleState,version=1.5");**

**BundleStateMBean bundleStateMBean =**

**JMX.newMBeanProxy(msc, mbeanName, BundleStateMBean.class);**

* Данные, возвращаемые бинами, представлены в виде таблиц данных, каждая строка состоит из именованных атрибутов. Это позволяет обходиться без пользовательских типов данных.
  + Вся таблица – **TabularData**
    - тип зависит от бина, например для **BundleStateMBean** это **BUNDLES\_TYPE**
    - **BUNDLES\_TYPE** константа типа **TabularType**, задающая описание коллекции строк

**TabularType BUNDLES\_TYPE = Item.tabularType("BUNDLES", "A list of bundles",**

**BUNDLE\_TYPE,IDENTIFIER);**

* + Каждая строка - **CompositeData**
    - Тип зависит от бина, например для **BundleStateMBean** это **BUNDLE\_TYPE**
    - **BUNDLE\_TYPE** константа типа **CompositeType**, задающая описание колонок строки

**CompositeType BUNDLE\_TYPE =Item.compositeType("BUNDLE",**

**"This type encapsulates OSGi bundles",**

**EXPORTED\_PACKAGES\_ITEM, FRAGMENT\_ITEM, FRAGMENTS\_ITEM,**

**HEADERS\_ITEM, HOSTS\_ITEM,IDENTIFIER\_ITEM, IMPORTED\_PACKAGES\_ITEM,**

**LAST\_MODIFIED\_ITEM,LOCATION\_ITEM, PERSISTENTLY\_STARTED\_ITEM,**

**REGISTERED\_SERVICES\_ITEM,REMOVAL\_PENDING\_ITEM, REQUIRED\_ITEM,**

**REQUIRED\_BUNDLES\_ITEM,REQUIRING\_BUNDLES\_ITEM, START\_LEVEL\_ITEM,**

**STATE\_ITEM,SERVICES\_IN\_USE\_ITEM, SYMBOLIC\_NAME\_ITEM, VERSION\_ITEM);**

* + - Каждое описание колонки есть константа типа **Item**

**Item SYMBOLIC\_NAME\_ITEM = new Item(SYMBOLIC\_NAME,**

**"The symbolic name of the bundle", SimpleType.STRING);**

* Данные, принимаемые бинами, организованы так же, но используются специфические реализации этих двух интерфейсов, позволяющие создавать из их коллекций
  + **TabularDataSupport**, в качестве типа коллекции указываем конструктору **JmxConstants.PROPERTIES\_TYPE**
  + **CompositeDataSupport**, в качестве типа коллекции указываем конструктору **JmxConstants.PROPERTY\_TYPE**
* клиент должен импортировать только OSGi MBean API. Можно ничего не импортировать, а идти через обобщенное API, доступной посредством **MBeanServerConnection**.

## BundleStateMBean (osgi.core:type=bundleState,version=1.5)

* + получение состояния инсталлированного bundle, импортированных пакетов, заголовков, сервисов, которые они регистрируют
    - Пример

**TabularData bundlesTable = bundleStateMBean.listBundles();**

**Collection<CompositeData> bundles =**

**(Collection<CompositeData>) bundlesTable.values();**

**for (CompositeData bundleInfo : bundles) {**

**if (bundleInfo.get(BundleStateMBean.SYMBOLIC\_NAME).equals("mybundle")) {**

**System.out.println("Application state = " +**

**bundleInfo.get(BundleStateMBean.STATE));**

**}**

**}**

* + - Атрибуты строки данных
      * **BundleStateMBean.SYMBOLIC\_NAME** – наименование бина
      * **BundleStateMBean.STATE** – состояние бина
      * **BundleStateMBean.IDENTIFIER** – идентификатор бина в виде long, определяется порядком инсталляции bundle, системный имеет идентификатор «0», не переживает рестарт сервера, хороший кандидат для кэширования на клиенте
    - Основные методы бина
      * **getBundleSymbolicName(long)**
      * **getHeaders(long)**
      * **getImportedPackages(long)**
      * **getLocation(long)**

## ServiceStateMBean (osgi.core:type=ServiceState,version=1.5)

* + получение информации о сервисах, зарегистрированных в реестре OSGi
    - Пример

**long [] serviceIds = bundleStateMBean.getRegisteredServices(myBundleId);**

**mbeanName = new ObjectName ("osgi.core:type=ServiceState,version=1.5");**

**ServiceStateMBean serviceStateMBean = JMX.newMBeanProxy(msc, mbeanName, ServiceStateMBean.class);**

**for (long serviceId : serviceIds) {**

**long [] bundleIds = serviceStateMBean.getUsingBundles(serviceId);**

**System.out.println("The service '" + serviceStateMBean.getObjectClass(serviceId)+**

**"' is being used by bundles :");**

**for (long bundleId : bundleIds) {**

**System.out.println(bundleStateMBean.getSymbolicName(bundleId));**

**}**

**}**

* + - Основные методы бина
      * **getProperties(long)** – получение свойств сервиса
      * **getBundleIdentifier(long)** – переход к bundle, экспортирующему сервис

## PackageStateMBean (osgi.core:type=PackageState,version=1.5)

* + получение информации о пакетах.
    - В отличии от bundle и сервисов пакеты идентифицируются не через longId, а через сочетание имени и версии – «manning.osgi;1.0.0.0». Это слегка отличается от синтаксиса определения наименования пакета в манифесте - Export-Package: manning.osgi;version=1.0.0.0
    - Пример

**String [] packages = bundleStateMBean.getExportedPackages(myBundleId);**

**mbeanName = new ObjectName("osgi.core:type=PackageState,version=1.5");**

**PackageStateMBean packageStateMBean =JMX.newMBeanProxy(msc, mbeanName, PackageStateMBean.class);**

**for (String osgiPackage: packages) {**

**int sepIndex = osgiPackage.indexOf(";");**

**String pkg = osgiPackage.substring(0, sepIndex);**

**String ver = osgiPackage.substring(sepIndex + 1);**

**long [] bundleIds =**

**packageStateMBean.getImportingBundles(pkg,ver, myAppId);**

**System.out.println("The package '" + osgiPackage + "' is being used by bundles :");**

**for (long bundleId : bundleIds) {**

**System.out.println(bundleStateMBean.getSymbolicName(bundleId));**

**}**

**}**

## FrameworkMBean(osgi.core:type=Framework,version=1.5)

* + управление bundles как таковыми, инсталляция, запуск, остановка, обновление, деинсталляция
    - Пример

**mbeanName = new ObjectName("osgi.core:type=Framework,version=1.5");**

**FrameworkMBean frameworkMBean = JMX.newMBeanProxy(msc, mbeanName, FrameworkMBean.class);**

**long bundleId = frameworkMBean.installBundle("bundle/mybundle.jar");**

* + - При инсталляции указывается путь относительно директории в которой запущен фреймворк, можно посмотреть через user.dir
    - Поддерживаемые функции
      * **installBundle()**
      * **get/setFrameworkStartLevel() –** текущий для платформы
      * **get/setInitialBundleStartLevel()** – начальный для всех последующих установленных
      * **get/setBundleStartLevel() –** текущий для bundle
      * **restartFramework() / shutdownFramework();**
    - Пре инсталляции bundle поддерживает пакетные операции
      * необходимы, когда за раз надо, к примеру, установить несколько bundle. Делать это последовательно – излишняя нагрузка на сеть
      * пример

**mbeanName = new ObjectName("osgi.core:type=Framework,version=1.5");**

**FrameworkMBean frameworkMBean = JMX.newMBeanProxy(msc, mbeanName, FrameworkMBean.class);**

**String [] bundleLocations = {"bundle/bundleA.jar", "bundle/bundleB.jar",**

**"bundle/bundleC.jar"};**

**CompositeData batchResult = frameworkMBean.installBundles(bundleLocations);**

**Long [] installedBundleIDs = (Long[]) batchResult.get(FrameworkMBean.COMPLETED);**

**for (int i = 0; i < installedBundleIDs.length; i++) {**

**System.out.println("Bundle '" + bundleLocations[i] +**

**"' was installed successfully and its ID is " +**

**installedBundleIDs[i]);**

**}**

**if (((Boolean) batchResult.get(FrameworkMBean.SUCCESS))== false) {**

**String culpritBundleLocation =**

**(String) batchResult.get(FrameworkMBean.BUNDLE\_IN\_ERROR);**

**String reason = (String) batchResult.get(FrameworkMBean.ERROR);**

**System.out.println( "Bundle '" + culpritBundleLocation +**

**"' failed to install because of: " + reason);**

**String [] remainingBundleLocations =**

**(String []) batchResult.get(FrameworkMBean.REMAINING);**

**System.out.println("The remaining bundles “+**

**”still need to be installed: ");**

**for (String remaingBundleLocation : remainingBundleLocations)**

**System.out.println(remaingBundleLocation);**

**}**

**}**

* + - * Это НЕ транзакционная операция, выполнение проводится до первого ошибочного результата. Все остальные определяются в REMAINING
      * Соответственно в строке данных
        + **FrameworkMBean.SUCCESS** – общий результат операции
        + **FrameworkMBean.COMPLETED** – перечень благополучно установленных
        + **FrameworkMBean.BUNDLE\_IN\_ERROR** – расположение первого же bundle, чья инсталляция вызвала ошибку
        + **FrameworkMBean.ERROR** – описание ошибки
        + **FrameworkMBean.REMAINING** – перечень bundle, которых даже не попытались установить

## ConfigurationAdminMBean (sgi.compendium:service=cm,version=1.3)

* + работа с конфигурацией сервисов, необязательный для поставщика платформы сервис
    - Пример конфигурирования

**ConfigurationAdminMBean cmBean =**

**JMX.newMBeanProxy(msc, mbeanName, ConfigurationAdminMBean.class);**

**TabularData properties = new TabularDataSupport(JmxConstants.PROPERTIES\_TYPE);**

**Map<String, Object> propertyValue = new HashMap<String, Object>();**

**propertyValue.put(JmxConstants.KEY, "port");**

**propertyValue.put(JmxConstants.VALUE, 9000);**

**propertyValue.put(JmxConstants.TYPE, JmxConstants.INTEGER)**

**CompositeData property = new CompositeDataSupport(JmxConstants.PROPERTY\_TYPE,propertyValue);**

**properties.put(property);**

**cmBean.update("manning.enterpriseosgi.notification.broker",properties);**

* + - Пример запроса конфигурации

**TabularData newProperties = cmBean.getProperties("manning.enterpriseosgi.notification.broker");**

**CompositeData portProperty = newProperties.get(new String[] {"port"});**

**System.out.println("The value of the new port configuration**

## Оповещения

* + Получив **BundleStateMBean** и the **ServiceStateMBean** можно зарегистрировать удаленные слушатели на события, связанные с bundle и сервисами. Фактически это подписка на сообщения из очередей **org/osgi/framework/BundleEvent и org/osgi/framework/ServiceEvent**
  + Пример подписки (Последний параметр true определяет, что создаваемый прокси будет также реализовывать **NotificationEmitter**)

**mbeanName = new ObjectName("osgi.core:type=bundleState,version=1.5");**

**BundleStateMBean bundleStateMBean =**

**JMX.newMBeanProxy(msc, mbeanName, BundleStateMBean.class, true);**

**MBeanServerNotificationFilter filter =**

**new MBeanServerNotificationFilter();**

**filter.enableObjectName(new ObjectName("osgi.core:type=bundleState,version=1.5"));**

**((NotificationEmitter) bundleStateMBean).addNotificationListener(**

**new BundleEventListener(), null, null);**

* + Пример слушателя

**public class BundleEventListener implements NotificationListener {**

**public void handleNotification( Notification notification,Object handback) {**

**CompositeData bundleEvent = (CompositeData) notification.getUserData();**

**String bundleSymbolicName = (String) bundleEvent.get(**

**BundleStateMBean.SYMBOLIC\_NAME);**

**Integer bundleState = (Integer) bundleEvent.get(**

**BundleStateMBean.EVENT);**

**if (bundleState == 1) // Installed**

**System.out.println("Bundle '" + bundleSymbolicName**

**+ "' has been installed!");**

**if (bundleState == 16) // Uninstalled**

**System.out.println("Bundle '" + bundleSymbolicName**

**+ "' has been un-installed!");**

**}**

**}**

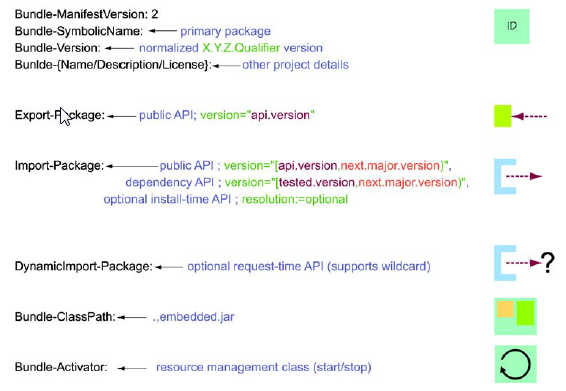
* + Работа с сообщением
    - Содержимое сообщения – уже знакомая нам таблица данных **CompositeData**
    - Тип сообщения - **Notification.getType(),** возвращает строковый тип полученного сообщения, например в случае **BundleEvent** это будет **org.osgi.jmx.framework.BUNDLE\_EVENT\_TYPE**
    - **BundleStateMBean.SYMBOLIC\_NAME** – наименование bundle источника событий
    - **BundleStateMBean.EVENT** – тип **события (INSTALLED=1, STARTED=2, STOPPED=4, UPDATED=8, UNINSTALLED=16)**
    - **ServiceStateMBean.OBJECT\_CLASS** – String[] интерфейсов, которые реализует сервис
    - **ServiceStateMBean.EVENT** – состояние сервиса **(REGISTERED=1, MODIFIED=2, UNREGISTERING=3)**

## Наименования

* Фреймворк
  + osgi.core: type=framework, version=1.5 и osgi.core: type=bundleState, version=1.5
  + Пространство имен osgi.core
  + реальный тип бина задается свойством type: framework, bundleState, serviceState, and packageState
* Enterprise сервисы
  + osgi.compendium: service=cm, version=1.3 и osgi.compendium: service=useradmin, version=1.1
  + Пространство имен osgi.compendium
  + Реальный тип бина задается атрибутом service, который соответствует пакету, в котором располагается интерфейс - org.osgi.service.**cm**.ConfigurationAdmin, реальное имя класса НЕ включается

# Формирование приложения

## Общая стратегия действий



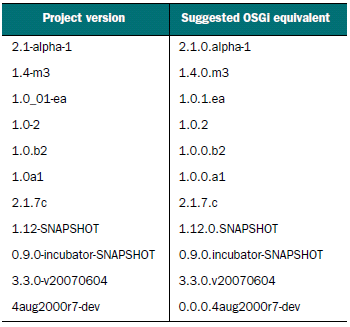
* В простейшем случае это просто добавление метаданных (уникальный идентификатор, перечень импортов / экспортов). Для большинства прикладных jar это так, но проблемы начинаются со сторонними библиотеками и фреймворками, оказывается необходим вдумчивый дизайн. Типовые вопросы:
  + Где грань между публичной и приватной частью
  + Какие импорты обязательны, а какие опциональны
  + Какие версии с какими совместимы

## Идентификация bundle

* OSGI не накладывает ограничений на формат идентификатора, главное, чтобы ваши одновременно загружаемые bundle не пересекались по именам
* НО рекомендует использовать reverse domain naming соглашение. Варианты:
  + Если bundle главным образом экспортирует один пакет, то его имеет смысл использовать как идентификатор
  + Можно использовать доменное имя проекта, который поставляет jar файл, <http://kxml.sourceforge.net/kxml2/> => net.sourceforge.kxml.kxml2
  + Можно использовать Maven groupId + artifactId
* Если мы имеем дело с особенной реализацией какого-то API, то можно использовать предыдущие стратегии (символическое имя API, наименование главного пакета и так далее) и суффикс, обозначающий уникальную имплементацию - slf4j.log4j и slf4j.jcl
* Если мы делаем собственную версию какой-то библиотеки (не обязательно меняем, возможно просто пакуем для использования в OSGi), то можно использовать схему <свой домен>+ <символическое имя библиотеки>, подчеркивая, что это не официальная поставка, а именно мы ответственны за ее метаданные - com.springsource.slf4j.api
* В принципе потом всегда можно переименовать bundle, так как основной вариант связывания – по пакетам. Именно поэтому плохо использовать зависимости от bundle.

## Версионность bundle

* + К сожалению далеко не все проекты используют стандартную для OSGi схему наименования major.minor.micro.qualifier, приходится конвертировать.
  + Стандартного способа нет, типовые варианты



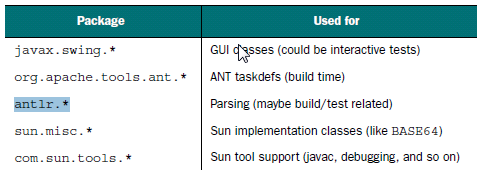
* + Есть утилита bnd, она пытается это делать исходя из некоторого набора интуитивных предположений
  + Некоторые пакеты меняются чаще чем другие, версия bundle меняется при любом изменении версии одного из экспортируемых пакетов или даже чаще, когда фиксим ошибки, оставляя API неизменным

## Экспорт пакетов

* Если bundle предоставляет реализацию сервиса через реестр, то он не должен ничего экспортировать, а только импортирует реализуемое API
* Типовая проблема – библиотека внедряет в себя для удобства распространения часть другой библиотеки, которая может встретиться в вашем проекте.
  + Пример: Apache Bean Utils включает в себя часть Apache Commons Collections
  + Достаточно распространенный вариант, причины
    - Не хотят раскрывать свои классы для окружения
    - Хотят иметь все потенциальные зависимости в одном файле и облегчить установку для клиента
    - Используют средства типа Jar Jar Link, которые переупаковывают зависимости, используя различные пространства имен, чтобы избежать потенциальных конфликтов. Попытка закрыть слабую модульность в Java
  + В не OSGi окружении это не исключение, так как в итоге все свалится в одну кучу и загрузится одним classloader-ом. Однако и там это может служить источником головной боли, если версии загружаемых библиотек окажутся несовместными.
  + В OSGi возникает проблема:
    - Мы вынуждены экспортировать org.apache.commons.collections, так как основное API библиотеки использует его классы
    - Но он не полный и если кто-нибудь его импортирует, то он потенциально не увидит нужные ему классы
    - Одни и те же классы встречаются импортированными из разных мест, что ведет к ClassCastException
* Выходы
  + В OSGi импорт перекрывает локальные классы, можно импортировать пакет опционально. Если он есть – будет использовать bundle библиотеки коллекций, если нет – локальная реализация. Однако остается проблема, когда BeanUtils и его клиент привяжутся к различным экспортерам пакета (неконсистентный импорт)
  + Чтобы сохранить консистентность импортируемых классов BeanUtils придется использовать «uses» при экспорте своего API
    - Пример: Export-Package: org.apache.commons.beanutils; uses:="org.apache.commons.collections"
    - Правда есть BeanUtils bundle будет установлен и resolved первым, то это все равно не поможет – импорт останется неконсистентным и получим ClassCastException
  + Единственный полноценный вариант – перепаковка BeanUtils, удаление из нее всех следов Commons Collections и постановка правильного импорта из полноценной реализации
  + Лучше связываться с разработчиками и просить их исправить ситуацию

## Импорт пакетов

* + Для автоматизации построения перечня импортируемых пакетов можно использовать анализаторы байткода типа bnd (<http://aqute.biz/Code/Bnd>) с последующей ручной доводкой.
    - Средство от создателей OSGi, набор ANT задач, которые управляются дополнительными заголовками в файле манифеста, а также файлом конфигурации, который входит в состав исходников проекта.
    - В отличии от стандартных, пакетные инструкции bnd могут содержать «\*», также команды исключения тех или иных пакетов из иерархии при импорте / экспорте
    - Автоматически импортирует экспортируемые пакеты, что является рекомендованной практикой. Можно запретить.
    - В частности им можно автоматизировать перепаковку неправильных jar с исключением фрагментов
  + При импорте
    - используемых пакетов задавать рекомендуется задавать версию – от минимально приемлемой, до но НЕ включая следующий major версию, что означает, что версии больше не совместимы
    - имплементируемых пакетов – до но НЕ включая minor версию. Если в интерфейс добавляются методы, то клиенты еще могут работать, а вот реализации уже не совместимы
    - можно не вручную указывать диапазоны версий, а доверять определение bnd за счет макросов типа [${version;==;${@}},${version;+;${@}}) . Подробнее - http://aqute.biz/Code/Bnd#versionpolicy
  + Ручная доводка обязательна – в перечень импортируемых попадают пакеты, используемые самими инструментальными средствами, кроме того они могут опрашивать окружение на наличие третьих библиотек и так далее
    - Частые случаи:



* + - Для пакетов, которые никогда не встретятся в окружении исполнения – ставим optional или вообще исключаем из перечня
    - Если понятия не имеем что это – динамический импорт, но вообще это грубое средство заставить систему работать хоть как-нибудь

## Embedding

* Реализация – jar составляющий bundle, внутри него еще один jar, который за счет манифеста подключен в classpath. Альтернатива – пустой bundle, в который внедрены оба jar, и основной и используемый.
* Причины - если могут работать только вместе, если не хотим шарить статические переменные с остальными bundles, если хотим оставить использование закрытой подробностью реализации основного функционала.
* Плюс – легче реализовать, не надо возиться с импортом / экспортом пакетов. Минус – внедренное содержание останется недоступным в не OSGi среде, придется перепаковывать
* Пример CGLIB включает в себя ASM для процессирования кода
* Особенно часто применяется, когда возникает необходимость поддерживать ЖЦ используемой библиотеки (наличие у нее состояния, внешних соединений, пула нитей и так далее)

## Поддержка ЖЦ библиотек

* Сложно, если используемая библиотека имеет состояние: пул соединений с БД, кэш на файловой системе, пул нитей исполнителей и так далее. Необходимо корректно запускать и останавливать совместно с ЖЦ bundle, в который упаковывается библиотека.
* Пример: Apache HttpClient, пул нитей инициализируется по требованию, а при выгрузке библиотеки необходимо его остановить
* Средство – добавляем активатор, который доставляет события ЖЦ bundle до библиотеки

## Преобразование приложения в целом

* Простейший способ – все в один bundle, запаковываем все jar составляющие приложение в один bundle и прописываем их в определение Bundle-ClassPath
  + Плюсы
    - Легчайший первый шаг, чтобы заставить его заработать
    - практически ничего не надо импортировать, разве что только javax пакеты из JDK или пакеты, предоставляемые контейнером. В этом очень помогает свойство org.osgi.framework.bootdelegation
  + минусы
    - Не получаем никаких плюсов от модульности, фактически OSGi не работает, остается неактивной прослойкой
    - Для нескольких подобных приложений в контейнере много содержания будет продублировано
    - Не можем запускать / останавливать приложение, только вместе с контейнером
* Каждый jar в отдельный bundle
  + Стратегии
    - Используем bnd для генерации метаданных перед созданием jar файлов.
    - Создаем jar как обычно и потом препроцессируем за счет bnd
    - Используем bnd для генерации jar файлов вместо ant задачи
  + Ключевое отличие от Ant
    - Ant имеет дело с файлами и директориями, копируя их в один jar файл
    - Bnd оперирует classpath и перечнем инструкций, определяя какие классы и ресурсы надо запаковать в jar файл
* Частые проблемы
  + java.net.MalformedURLException: Unknown protocol
    - Причина – OSGi регистрирует собственную URLStreamHandlerFactory, ожидая что остальные bundle будут подключать к ней реализации протоколы как сервисы через реестр
    - Выход – пишем активатор
  + Приложение не может определить свою корневую директорию, пытаясь сделать это через анализ пути из загрузчика классов. Придется как-то сообщать ее приложению явно, например через –D
  + если раньше каждый класс видел ВСЕ остальные, то теперь только ТЕ, которые были выявлены на этапе разделения и доступны через экспортированные / импортированные пакеты. В результате имеем кучу runtime ошибок о не найденных классах или ресурсах
  + Утилиты типа bnd, анализирующие исполнимый код, принципиально НЕ отслеживают зависимости от ресурсов, например от файлов runtime конфигурации или скриптов, располагающихся в тех или иных пакетов. А значит не генерируют инструкции импорта данных пактов. Выход
    - Написать к ним свой собственный custom analyzer
    - Выявлять нереализованные зависимости за счет тестов, использующих данные ресурсы и добавлять их ручками
    - Силовые методы
      * Использовать DynamicImport-Package, но не всех классов, а ТОЛЬКО внутри верхнего пакета приложения
      * Использовать org.osgi.framework.system.packages.extra или даже org.osgi.framework.bootdelegation, чтоб ы сделать какие-то части приложения доступными всем bundle без необходимости явного прописывания зависимостей в их метаданных
      * Так как все равно при этом оказывается невозможным применение OSGi фич типа одновременного использования разных версий или динамического развертывания, то проще сливать подобные проблемные budle вместе и не мучаться
* Стратегии разделения на отдельные bundle. Общий подход: выделение каждого конкретного bundle должно что-то давать в плаке облегчения процессов разработки и развертывания.
  + По отдельным компонентам и их границам. Применимо, если есть внятный дизайн системы
  + По областям ответственности. Как минимум бизнес-логика, доступ к данным и интерфейс
  + По тому, какие части должны обновляться / исправляться / поставляться независимо
  + Выделение опциональных частей, общих наборов утилит, используемых всеми остальными
  + По тому, как разработка приложения в целом разделяется между отдельными разработчиками / командами разработчиков.
  + Исходя из необходимости часто перезагружать какую-то часть приложения при отладке
* Если набор bundle всегда устанавливаются / обновляются вместе НЕ имеет смысл их разделять, проще слить вместе и не возиться с поддержкой рзделения.
* Стратегии построения перечня экспортируемых пакетов
  + От структуры кода и понимания перечня зависимостей. Помогают анализаторы типа ([www.headwaysoftware.com/products/structure101](http://www.headwaysoftware.com/products/structure101)
  + Читаем заголовки Import-Package из сформированных манифестов
  + Много раз переразвертываем приложение и вычищаем ошибки
* После разделения приложения по отдельным JAR, каждый из которых bundle обеспечить совместимость при обычном запуске из JVM можно
  + Командный файл, строящий classpath из всех jar для запуска главного, содержащего main класс
  + В главный jar помещаем заголовок манифеста Class-Path, в нем перечисляем остальные, которые автоматически должны быть добавлены в classpath. Недостаток – если коллекция jar будет разложена по-другому, то главный придется перепаковывать
* OSGi это НЕ система «все или ничего»
  + разделение всего приложения на отдельные Bundle нужно проводить ТОЛЬКО пока это приносит конкретные плюсы и не приносить больших проблем.
  + Зачастую действительно проще поместить тяжелые библиотеки сторонних производителейв глобальный claspath контейнера до того момента как либо дойдут руки разобраться как именно их лучше разделить, либо за вас это сделает поставщик библиотек.

## Использование сервисов

* В принципе опционально, приложение вполне может продолжать полагаться на старые механизмы удовлетворения внутренних зависимостей: конструировать их непосредственно, использовать фабрики, DI и так далее. Потенциальная проблема:
* Пример необходимости введения сервисов
  + у приложения есть механизм подключения плагинов путем добавления JAR файлов в определенную директорию. JAR файлы упаковались в один из bundle и механизм работать перестал.
  + Чтобы восстановить работу механизма надо обернуть старый механизм подключения в обертку, которая отслеживает появление bundle – реализаций плагинов и передает их данные старому механизму подключения плагинов. Следовательно, нужны сервисы

# Тестирование

* Тесты являются важным инструментом миграции приложения в OSGi – необходимо установить, что приложение ведет себя одинаково как при традиционном запуске, так и при запуске через OSGi. Соответственно надо прогонять весь набор тестов дважды
* Unit тесты сервисов и клиентов можно писать без проблем, так как SOGi платформа не накладывает на их интерфейсы ограничений в виде необходимости что-то реализовывать или наследовать. Проблема начинается, когда нам надо протестировать все вместе. А поднимается все вместе активатором. Соответственно две возможности: либо делаем mock эмуляцию OSGi инфраструктуры, либо делаем интеграционный тест с полноценным запуском контейнера
* Варианты расположения тестов
  + boot class path
  + system bundle export
  + внутри тестируемого bundle
  + вне тестируемого bundle

## Unit тесты

* Private классы тестируемого bundle могут быть протестированы только тестами, которые попадают в тот же classpath, то есть располагаются внутри тестируемого bundle. Публикуемые классы могут и должны быть протестированы как изнутри, так и извне чтобы имитировать реальные условия
* Как развертывать собственно JUnit или аналогичное средство тестирования
  + Развернуть как bundle
    - Необходимо пакетировать. Лучше всего через bndwrap. Чтобы увидел все тесты без необходимости перечисления пакетов можно использовать DynamicImport-Package: \*. Запуск тестов осуществляется при старте bundle за счет Main-Class: junit.textui.TestRunner. Аргументы передаем тут же.
  + Сделать доступным через внешний classpath
    - Это проще – не надо пакетировать
    - Недостаток – Junit не увидит тестовые классы, если конечно их не развернуть через тот же classpath или напрямую не передать ему уже созданные экземпляры. Последний способ означает, что придется писать собственное средство опроса развернутых bundle, инстанцирования тестовых объектов и скармливания их JUnit, вместо того, чтобы положиться на стандартный стартер тестов
* Если хотим тесты в отдельном bundle, но они разделяют одни пакеты с экспортируемыми классами, что бывает очень часто, то имеем большую проблему со split package.
  + Внешние для тестируемого кода тесты можно спокойно мигрировать в другие пакеты, они очевидно не полагаются на default область видимости
  + Можно идти сложным образом через сливание пакетов или импорт с указанием обязательного квалификатора
  + Можно JUnit bundle сделать фрагментом bundle с тестами, тогда пакеты с тестами не придется экспортировать, так как для них и JUnit будет один classpath
* Средства контроля покрытия (Cobertura)
  + Этапы работы: инструментирование / выполнение теста / анализ результатов
  + Инструментирование производится стандартными средствами на этапе сборки
  + На этапе выполнения две проблемы
    - Доступность классов инструментатора, три стандартных пути:
      * оформляем как bundle
      * даем доступ через org.osgi.framework.system.packages.extra или rg.osgi.framework.bootdelegation (в последнем случае необходимо убрать сгенерированные bnd импорты из заголовков тестовых bundle)
      * внедряем jar средства внутрь bundle с тестами и делаем его доступным через Bundle-ClassPath
    - Доступность базы данных где накапливается информация о выполнении. Через системное свойство net.sourceforge.cobertura.datafile

## Bundle тесты

* Когда хотим протестировать не класс или группу классов, а функционал bundle в целом
* Хорошо если зависимости поставляются bundle декларативно за счет DI, тогда ситуация ничем не отличается от предыдущей. Хуже, когда зависимости обеспечиваются посредством активатора и OSGi API. Тогда единственный способ – mock тесты
* Mock object – симуляция, а не реальная реализация объекта. Предоставляет тот же самое API, но его методы рассчитаны на то, что их вызывают в определенной последовательности, возвращают предопределенные значения или другие mock объекты
* Особенно удобны, когда надо протестировать реакцию bundle на поведение контейнера, которое трудно воспроизвести само по себе. Например ситуацию с race condition, когда context.getServiceReference() возвращает нормальную ссылку (сервис еще есть), а context.getService(reference) возвращает null (сервиса уже почему-либо нет). Без mock объектов обеспечить подобное поведение контейнера практически невозможно
* Типовые этапы работы с mock средствами:
  + Создаем прототипы mock объектов
  + Задаем ожидаемые последовательности поведения
  + Подготавливаем объекты к использованию – загоняем в начальное состояние
  + Выполняем тесты, передавая им в качестве аргументов / зависимостей mock объекты в начальном состоянии
  + Проверяем, что mock объекты подверглись ожидаемой последовательности воздействий
* Работа с Easy Mock (<http://easymock.org/>, <http://www.ibm.com/developerworks/java/library/j-easymock/index.html> )
  + Создание объектов

**BundleContext context = createStrictMock(BundleContext.class);**

**ServiceReference serviceRef = createMock(ServiceReference.class);**

**LogService logService = createMock(LogService.class);**

* + Задаем поведение

**expect(context.getServiceReference(LogService.class.getName())).andReturn(serviceRef);**

**expect(context.getService(serviceRef)).andReturn(logService);**

**logService.log(and(geq(LogService.LOG\_ERROR), leq(LogService.LOG\_DEBUG)), isA(String.class));**

* + Загоняем в начальное состояние

**replay(context, serviceRef, logService);**

* + Проигрываем ЖЦ тестируемого bundle

**BundleActivator logClientActivator = new Activator();**

**logClientActivator.start(context);**

**Thread.sleep(1000);**

**logClientActivator.stop(context);**

* + Проверяем, что mock объекты были использованы правильным образом

**verify(context, serviceRef, logService);**

* + Вместо createStrictMock можно использовать createNiceMock, который позволяет вызывать методы в любом порядке с любыми аргументами – объект-заглушка
* Работа с mock библиотекой от Spring ( <http://www.springsource.org/osgi>)

**public void testAuctionnerActivator() throws Exception {**

**MockBundleContext mockContext = new MockBundleContext();**

**SealedFirstPriceAuctioneerActivator activator =**

**new SealedFirstPriceAuctioneerActivator();**

**activator.start(mockContext);**

**ServiceReference reference =**

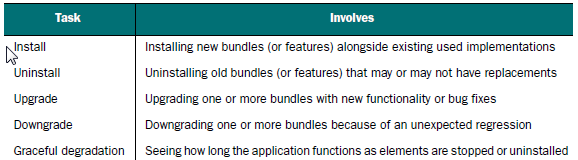
**mockContext.getServiceReference(Auctioneer.class.getName());**

**assertNotNull(reference);**

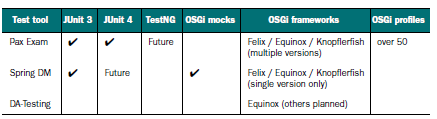
**}**

## Интеграционные тесты

* Обычно данные тесты не должны увеличивать степень покрытия тестами, которая должна обеспечиваться unit и mock тестами, обрабатывающими все основные критические пути. Вместо этого тестируется
  + совместимость с различными контейнерами
  + правильность пакетирования и следования программной модели контейнера
  + взаимодействие развернутых bundle между собой
  + управляемость bundle, совместимость приложения в целом с операциями обновления / запуска / остановки и так далее



* Средства тестирования



* + Open Participation Software for Java’s (OPS4J) Pax Exam (<http://wiki.ops4j.org/display/paxexam>)
    - Позволяет прогонять наборы тестов в разных OSGi фрейморках, просто добавляя по строчке в описание тестовой конфигурации. Фрейморк сам выкачивает необходимые файлы, конфигурирует и разворачивает OSGi контейнеры
    - Преимущественно рассчитан на Maven, но в принципе может использоваться и за счет Ant
    - Каждый тест сообщает фрейморку конфигурацию в которой он должен быть запущен, в описание конфигурации входит:
      * Maven наименования и версии ядра и compendium
      * Перечень bundle, которые должны быть развернуты
      * Наименование OSGi контейнера
    - Методы получения конфигурации и тестирования запускаются средством в разных потоках, передача информации между ними только через системные свойства
    - Unit и Bundle тесты также можно запускать при помощи данного средства
    - Для проведения bundle теста код получает для начала работы ссылку на BundleContext тестового bundle
    - Для проведения интеграционного теста он наследуется от ConfigurationBaseTest, ему доступны методы базового класса для
      * Определения конфигурации
      * Инсталляции / удаления bundle
      * Ожидания, пока предыдущая команда управления контейнером будет выполнена
    - Если мы запускаем update bundle, то текущая хранимая конфигурация bundle сохраняется (хотя может оказать и не доступной для использования новой версией).
  + Spring DM’s test support (<http://static.springsource.org/osgi/docs/1.2.1/reference/html/testing.html>)
    - тест наследуется от **AbstractConfigurableBundleCreatorTests**, который берет на себя старт контейнера, регистрацию тестирующего bundle, имеющего доступ ко всем средствам платформы, его запуск и остановку платформы
    - Пример активатора тестирующего bundle

**public class IntegrationTestAuction extends**

**AbstractConfigurableBundleCreatorTests {**

**public void testSealedFirstPriceAuction() {**

**ServiceReference reference =**

**bundleContext.getServiceReference(**

**Auctioneer.class.getName());**

**Auction auction = (Auction)**

**bundleContext.getService(reference);**

**// Place bids, asks...**

**}**

**protected String getPlatformName() {**

**return Platforms.FELIX;**

**}**

**}**

* + - Тест должен быть запущен из Felix директории, чтобы загрузить все его потроха. Предварительно в контейнере должны быть установлены все необходимые bundle, это делается вручную
    - Главным образом оптимизирован для тестирования на платформе Spring DM сервера
  + Dynamic Java’s DA-Testing ([www.dynamicjava.org/posts/da-testing-introduction](http://www.dynamicjava.org/posts/da-testing-introduction)).
    - Имеет свое собственное API описание тестов, подходящее для специальных ситуаций типа, например, race condition
* Типовой ЖЦ подобных средств
  + Подготовка OSGi контейнера, выбранного одного или нескольких по кругу, когда тестируем тесты на наборе контейнеров
  + Устанавливаем необходимые bundle
  + На ходу пакетируем bundle с тестами
  + Устанавливаем тестовый bundle и запускаем тесты
  + Останавливаем контейнер

# Отладка

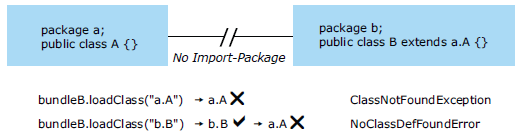
* Дополнительный плюс от модульности: вместо полностью неработоспособного приложения получаем проблемы только в одной части. Вот только как выяснить в какой?
* Основные проблемы: множество версий классов сосуществует одновременно, ошибочные импорты / экспорты дают в итоге невидимость классов или несовместимость между собой, утечки памяти вследствие удержания ссылок уже выгруженными

## Диагностика ошибок

* Видим текст исключения, какие-то классы и пакеты
  + Как определить в какой bundle уходит стен
  + Как определить версию этого bundle, если их в системе установлено несколько. Особенно это важно не в девелоперской, а в уже эксплуатирующейся системе
* Обычные отладчики не помогают, так как не в курсе множественности версий классов и не могут подключать множественные исходники. Существуют специализированные, например в составе Eclipse
* В принципе можно не подключать специально исходники, а упаковывать их в пакет с bundle. Они должны быть размещены в директории OSGI-OPT/src, там их ожидает увидеть отладчик
* После отладки
  + HotSwap.
    - Во-первых, не всегда может заменить реализацию класса, например Sun JVM не способна этого сделать, когда изменился набор методов класса, только IBM JVM.
    - Во-вторых, будут заменены ВСЕ версии классов, загруженные в JVM на текущий момент, что сделает остальные версии bundle неработоспособными
  + JRebel
    - Когда перекомпилируем класс заменяет только соответствующие версии класса
    - Проблемы с производительностью. Кроме того, исходники должны быть правильно разложены, поддерживается только Equinox OSGi контейнер
    - Так как это java агент, то надо курочить строку запуска JVM, что не всегда возможно, особенно в случае серверов приложений. Также могут быть против безопасники
  + Основной способ – пересборка и обновление bundle.
    - Если приложение правильно разделено на модули, то отладка одной фичи затрагивает только связанные между собой классы в одном или нескольких легко заменяемых bundle и не затрагивает все приложение
    - Проблема, если у bundle есть состояние, его придется сохранять / загружать при остановке / запуске bundle

## Проблемы с классами

* **ClassNotFoundException** – загрузчик классов не способен найти его по имени ни сам, ни делегируя эту задачу другим загрузчикам
  + Часто проблема вызвана package-private классами, по определению они видны только классам из того же пакета И из того же загрузчика. В случае, когда пакет разделен между несколькими bundle, то такие классы из разных bundle не будут видеть друг друга.
  + Как правило разрешается проверкой манифестов и конфигурационных файлов. Но может упираться в специальные загрузчики классов и тогда придется смотреть в их исходники в поиске места, откуда они пытаются считать информацию
* **NoClassDefFoundError** – загрузчик нашел класс, начал его грузить, но загрузить не смог, так как он использует (наследуется от или имеет члены) недоступный класс
  + Основная причина – класс был скомпилирован в присутствии какого-то API, которое на момент загрузки оказалось недоступным
  + Стек в этом случае указывает на класс, который на самом деле присутствует, что вводит в заблуждение. Основная причина проблемы указана в самом низу стека.
* Разница: **ClassNotFoundException** указывает на недоступность класса, а **NoClassDefFoundError** на проблемы с загрузкой все же найденного класса вследствие невозможность разрешить его зависимости



* **ClassCastException** - проблемы с приведением классов
  + Типовые причины
    - ServiceTracker.open(true) выдает все сервисы определенного типа без проверки совместимости их версий с версиями, определенными запрашивающим bundle. Соответственно будут проблемы
    - Включение приватной копии какого-либо API в bundle и одновременно развертывание bundle, предоставляющего данное API, с высокой вероятностью даст эту проблему
    - Экспорт API из bundle, которое использует другое API, несколько реализаций которого доступно в системе
  + Методы разрешения:
    - у несовместимых объектов взять getClassLoader().toString() – они будут указывать на разные bundle
    - использовать команды консоли OSGi, которые для пакета выдают перечни экспортеров / импортеров
    - использовать «uses», чтобы обеспечить согласованность пространства классов. В случае невозможности это сделать, сообщение об ошибке будет включать ссылки на конфликтующие версии пакетов, что является дополнительным бонусом
    - установить более детальный уровень выдачи сообщений контейнера, в логе будут указания на проблемы с зависимостями. Иногда есть специальные команды для получения перечня неудовлетворенных зависимостей
* Загрузка классов посредством **Class.forName()**
  + Часто используется в предположении, что все классы находятся в одном пространстве. Часто не работает в высокосегментированном модульном окружении
    - классы, чтобы быть доступными должны содержаться в bundle, быть импортированными или поставляться загрузчиком классов самого OSGi контейнера
    - можно лечить за счет динамического импорта, проблемы
      * все равно работает только тогда, когда нужны классы были экспортированы, это далеко не всегда
      * связывает наш bundle с большим количество других, если кто-то из них будет обновлен, то обновление затронет наш и так далее
      * bundle может быть связан только с одной версией пакета одновременно.
  + Лучший способ: использовать для загрузки не свой загрузчик, а загрузчик того bundle который гарантированно имеет доступ к нужному классу.
    - Например, для обработки полученного сообщения нужно загрузить какой-то ресурс и он лежит рядом с классом сообщения. Соответственно, для загрузки ресурса надо использовать message.getClass().getClassLoader().loadClass(className)
    - Еще хуже практика использования Class.forName() вместо classloader.loadClass(), так как в первом случае информация кэшируется в двух загрузчиках: который загружает и который в действительность содержит определение класса.
* Использование ThreadContextClassLoader (TCCL)
  + Получается как Thread.currentThread().getContextClassLoader(), полезен, когда мы вниз по стеку имеем динамическую работу с классами, но не хотим протаскивать специальный загрузчик через параметры всех функций
  + Если в каком-то месте его меняем, то обязательно надо использовать try-finally для того, чтобы в любом случае возвратить предыдущий. Таких вложенных замен может быть несколько.
  + Соответственно таким образом можно передать загрузчик классов из одного bundle в другой: вызывающий операцию bundle устанавливает свой загрузчик как TCCL, реализующий операцию использует его для получения доступа к ресурсам первого bundle

## Утечки памяти

* Основная проблема: при перезагрузке bundle, если кто-то продолжает удерживать ссылку на порожденный объект, на этой ссылке повисает загрузчик классов уже выгруженного bundle со всеми своими потрохами и определениями классов. Кроме того истощается PermGen, который вынужден содержать множественные определения один и тех же классов, причем со всеми их статическими членами, подвешенными на них коллекциями, кэшами и так далее
* ThreadLocal требует обязательной ручной очистки
  + в 5-ой Java это хранилище не чистилось с завершением работы нити
  + могут использоваться долгоживущие пулы нитей, когда одна и та же нить используется много раз для обработки все новых и новых запросов
* Хороший инструмент анализа памяти - Eclipse Memory Analyzer (MAT, <http://eclipse.org/mat/>), в числе прочего выдает коллекцию стеков, приводящих к появлению анализируемых экземпляров
* Болтающиеся (dangling) сервисы
  + Типовая ошибка: получив сервис из ссылки прикапываем его для долговременного хранения. Если поставщик сервиса в это время перегрузился, то мы продолжаем удерживать и саму реализацию сервиса, и загрузки классов bundle зарегистрировавшего сервис и все остальное – большой и тяжелый граф объектов
  + Решения
    - Не хранить ссылки на реализации сервисов!
    - Можно использовать для диагностики bundle.getServiceInUse()
      * к сожалению пользователи часто не вызывают вовремя ungetService(), что порождает ложные свидетельства утечки.
      * это все равно не дает нужную информацию о прикопанных ссылках.
    - Можно использовать ServiceReference.getUsingBundles() - те же проблемы
    - Специализированные средства типа OSGi inspector agent (<http://wiki.github.com/mirkojahn/OSGi-Inspector/>)
    - Для доставки сервисов клиентам использовать не OSGi API, а DI и декларативное описание зависимостей
    - Регистрировать в реестре не сам сервис, а его прокси. Даже если клиент сохранит надолго полученную ссылку, то повиснет экземпляр прокси, что тоже утечка, но гораздо меньше. Типовая реализация:

**Proxy.newProxyInstance(LogService.class.getClassLoader(),**

**new Class[] { LogService.class },**

**new InvocationHandler() {**

**@Override**

**public Object invoke(Object proxy, Method method, Object[] args)**

**throws Throwable {**

**LogService cachedService = (LogService)**

**loggerMap.get(bundle);**

**if (cachedService != null) {**

**return method.invoke(cachedService, args);**

**}**

**throw new IllegalStateException("LogService has been deactivated");**

**}**

**});**

# Работа с компонентами внутри bundles

## Основы

* Модули имеют дело с упаковкой кода и зависимостями между фрагментами кода. Обращаем внимание на статический код и зависимости времени компиляции.
* Компоненты имеют дело с реализацией высокоуровневой функциональности и зависимостями между отдельными функциями. Компоненты, как правило, нуждаются в сервисе управления зависимостями между ними, но в принципе им не требуется модульность. Обращаем внимание на экземпляры, их ЖЦ и зависимости времени выполнения.
* Плюсы компонентной модели
  + Улучшаю повторное использование, так как ограничены зависимости и скрыты детали реализации
  + Четкая спецификация требуемых / предоставляемых интерфейсов способствует повторному использованию и применению композиции как архитектурного принципа
  + Так как компонентная модель требует специфических шаблонов дизайна, то код становится более однообразным и легче для понимания
  + Однообразие дизайна способствует внешней управляемости приложений, основанных на компонентной модели
  + Внешние агенты, основываясь на компонентной модели, могут больше узнавать о коде и выполнять по отношению к нему различные задачи
* Минусы компонентной модели
  + Компонентные фреймворки обычно тяжелы и могут быть излишне глобальны для маленьких приложений. Большое время вхождения для нового разработчика.
  + Отладка приложений в подобной архитектуре затруднена, так как зачастую не понятно что происходит в ситуации, когда компонент не был создан и не стал в итоге доступен для других компонент. Длинные стеки, загрязненные кодом фреймворка, трудный для понимания
  + Проблемы времени сборки / выполнения вызванные ошибками в конфигурационных файлах, сложность отладки конфигурационных файлов
* Тип компонентной модели OSGi – сервис ориентированная компонентная модель, компонент либо публикуют сервисы / либо потребляют сервисы. Каждый bundle это один / несколько компонент, их интерфейсы публикуются как сервисы. В то же время каждый bundle это физический JAR файл, то есть модуль.
* На уровне модулей bundle это способ описания куска кода, его потребностей и возможностей. На компонентном уровне спецификация OSGi не предоставляет для bundle формальных средств описания предоставляемых сервисов, их потребностей и возможностей. Вместо декларативного описания – динамический реестр сервисов, управляемый программно.
* Недостатки OSGi с точки зрения компонентной архитектуры
  + Код приложения свои зависимости от сервисов удовлетворяет программно, зачастую достаточно сложная модель на основе обработки событий и много повторяющегося кода.
  + не предоставляет формальных способов описания приложения, составляемого из набора отдельных компонент-сервисов
  + нет возможности формирования компонент из POJO объектов, их конфигурировать, связывать с сервисами OSGi, кастомизировать жизненный цикл и так далее
* общий подход
  + переход от концепции bundle==компонент к концепции bundle – контейнер для множества компонент
  + добавление новых типов метаданных для описания содержащихся в bundle компонент. Эти новые типы данных обрабатываются bundle компонентного фреймворка на основе шаблона «extender».
  + Мета описания компонент в частности содержат описания публикуемых ими сервисов, которыми будут удовлетворяться зависимости других компонент, а также зависимостей данной компоненты от сервисов, предоставляемых другими компонентами. Основные свойства типовой зависимости
    - Тип требуемого сервиса
    - Опциональность – может ли компонента функционировать без удовлетворения данной зависимости (например, опциональное логирование)
    - Количество элементов – зависит ли данная компонента от одного экземпляра сервиса данного типа или от нескольких (1:1 или 1:N)
    - Влияние изменений – они могут обрабатываться компонентой или при изменении внешних условий компонент просто сбрасывается и возможно создается новый экземпляр
  + bundle компонентного фреймворка, коллекционирующий у себя описания компонент, ответственен за
    - создание экземпляров компонент и публикацию содержащихся в них сервисов
    - удовлетворение их зависимостей от других сервисов. В зависимости от типа зависимостей и доступности требуемых сервисов при добавлении / удалении сервисов из системы возможно как активация / деактивация компоненты, так и просто установка / обнуление каких-то ее свойств
    - управление жизненным циклом
  + Дополнительные сервисы
    - Декларативная доставка до компонента свойств конфигурации, многоуровневое конфигурирование – от статических значений в дескрипторе до runtime значений в конфигурации.
    - Аспектное провязывание публикуемых / потребляемых сервисов: транзакционность, безопасность, аудит
* Стандартная модель удовлетворения зависимости компоненты от сервиса, много кода: достаточно сложного, но однотипного
  + создаем наследника от ServiceTracker, регистрируем его для получения сообщений о добавляемых / удаляемых сервисах системы определенного типа
  + при добавлении сервиса:
    - прикапываем ссылку на ServiceReference в коллекцию доступных сервисов
    - если компонент не функционирует из-за отсутствия сервиса, то инициируем его полученным сервисом и запускаем
  + при удалении сервиса
    - удаляем его из коллекции доступных
    - если компонент функционирует на основе полученного экземпляра сервиса, то либо пересаживаем компонент на другой сервис из доступных, либо останавливаем его функционирование
* Проблемы с множественными самопальными расширителями
  + Не предоставляют достаточной гибкости в части создания и конфигурирования объектов, которые будут создаваться расширителем и подключаться к инфраструктуре SOGi. Например, хочется передавать аргументы конструктору, устанавливать значения свойств
  + Сложно комбинировать возможности, навешиваемые разными расширителями на один класс – каждый расширитель норовит создать отдельный экземпляр класса и с ним дальше работать.
  + Тривиальное решение – сделать большой монолитный расширитель, ответственный за все (как минимум, persistence, transaction, security). Противоречит фундаментальной идее OSGi о том, что возможно пользовать только необходимыми сервисами и комбинировать различных их поставщиков в одной системе
* Способы обработки динамической природы зависимостей от сервисов
  + Использование синхронизирующей логики: синхронизированные блоки, AtomicReference, CopyOnWriteArraySet. Это позволяет компоненте при асинхронном обновлении ее свойств в ответ на события фреймворка видеть корректные значения полей
  + Обработка null для опционального сервиса или специального типа исключения, обозначающего, что сервис не доступен
  + Использование timeout при доступе с потенциально отсутствующему сервису
  + Объявление зависимости от сервиса как обязательной, если сервис опускается, то компонента становится неактивной, а значит больше к сервису не обращается и своих сервисов не предоставляет
  + Использование mock заглушки, которой передается управление, если запрашиваемый сервис не доступен. Заглушка принимает все вызовы, но ничего не делает.
* Решения не рассмотренные в данной книге
  + Google Guice peaberry (http://code.google.com/p/peaberry/)
  + ScalaModules (<http://wiki.github.com/weiglewilczek/scalamodules/>)
  + Beanome ([www.humbertocervantes.net/papers/ISADS2002.pdf](http://www.humbertocervantes.net/papers/ISADS2002.pdf))
  + Service Binder ([www.humbertocervantes.net/papers/ESEC2003.pdf](http://www.humbertocervantes.net/papers/ESEC2003.pdf))

## Declarative Services

* Спецификация разработана OSGi альянсом, как часть R4 compendium спецификации.
* Предназначен для управления компонентами в части публикации сервисов и удовлетворения зависимостей от сервисов. Убирает необходимость в активаторах, поддерживает отложенное создание компонент (вне зависимости от типа инициализации bundle), за счет этого уменьшает время старта системы. Избавляет компоненты от необходимости в однообразном коде интеграции в OSGi инфраструктурой. За счет этого снижает количество классов и объем требуемой памяти.
* Подключение bundle к фреймворку – за счет заголовка Service-Component, указывающего на файл описания содержащихся в bundle компонент.
  + Файлов может быть несколько, они могут располагаться где угодно. В заголовке они перечисляются через «,», может использоваться «\*»
  + Файлы могут располагаться в bundle-фрагментах, при этом заголовок должен быть в родительском bundle. Кастомизация компонент через перечень публикуемых фрагментов
* Формат публикации компоненты

**<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0">**

**<property name="simple.shape.name" value="Circle" />**

**<property name="simple.shape.icon" value="circle.png" />**

**<scr:implementation class="org.foo.shape.circle.Circle" />**

**<scr:service>**

**<scr:provide interface="org.foo.shape.SimpleShape"/>**

**</scr:service>**

**</scr:component>**

* + Свойства могут использоваться для конфигурирования экземпляра компоненты, также они могут использоваться для публикации сервиса как свойства сервиса. В последнем случае их наименование должно быть в виде <имя сервиса>.target, значение <имя свойства>=<значение свойства>
  + Свойства могут быть только примитивных типов. Таким образом, если надо передать клиенту компонента ресурс, то ему передаем URL, а он за счет classloader сервиса его самостоятельно грузит. Некоторая инверсия по сравнению с классическим подходом.
* Формат использования компоненты, точнее опубликованного ею сервиса

**<scr:component xmlns:scr=**[**http://www.osgi.org/xmlns/scr/v1.1.0**](http://www.osgi.org/xmlns/scr/v1.1.0)

**name="paint" immediate="true">**

**<scr:implementation class="org.foo.paint.PaintFrame" />**

**<scr:reference interface="org.foo.shape.SimpleShape"**

**target=”(name=main)”**

**cardinality="0..n" policy="dynamic"**

**bind="addShape" unbind="removeShape"/>**

**<scr:service>**

**<scr:provide interface="java.awt.Window"/>**

**</scr:service>**

**<scr:property name="name" value="main"/>**

**</scr:component>**

* + Отслеживание сервисов – bind / unbind методы, вызываются когда сервисы соответствующего типа появляются / исчезают из системы. Методы вызываются асинхронно в многопоточном режиме
    - Типы
      * void <method-name>(ServiceReference) – позволяет исследовать сервис не получая его, для использования нужен ComponentContext
      * void <method-name>(<parameter-type>) – ссылку можно прикопать, но надо вовремя ее сбросить в ответ на вызов unbind метод, ля поддержания многопоточности для хранения используем AtomicReference и CAS методы установки / удаления
      * void <method-name>(<parameter-type>, Map) – вместе со ссылкой на сам сервис передаются его свойства
    - видимость для фрейморка
      * public и protected – доступ разрешен
      * package private – разрешен, если метод расположен в реализующем классе или его наследнике в том же пакете
      * private – разрешен, если метод расположен в реализующем классе
    - в общем случае надо защищать данные методы за счет модификаторов доступа, чтобы другие сервисы не могли посредством них получить / подменить сервисы, к которым у них в нормальной ситуации не должно быть доступа
  + Свойство **cardinality** управляет как опциональностью сервиса (левая часть), так и количеством доступных сервисов (правая часть).
    - Если сервис опциональный, то просто вызывается unbind метод, если обязательный – вслед за этим экземпляр компонента сбрасывается. Когда сервис вновь появится, то экземпляр компонента будет создан заново
    - Если кардинальность N, то bind метод будет вызван для всех сервисах подходящего типа, если 1 – только один раз
  + Policy – динамическая компонента уведомляется о появлении / исчезновении требуемых сервисов, сложнее писать, например, из-за многопоточности. Статическая – связывается с сервисом один раз, как только сервис сбрасывается – сбрасывается и экземпляр компонента, который его использовал
* Жизненный цикл компоненты
  + Состояния, типовой проход: enabled -> satisfied -> activated
    - Enabled – флаг определяет, доступен ли компонент для управления.
      * Можно выставлять за счет одноименного атрибута в <component> элементе описания или выставляем программно через ComponentContext.
      * Используется чтобы быстро стартовать с малым количеством компонент, а затем программно запустить остальные
    - Satisfied – обязательные потребности удовлетворены, сервисы опубликованы, но компонент еще не стартовал
    - Activated – экземпляр компонента стартовал, так как кто-то запросил публикуемый им сервис
    - Modified – конфигурация компонента изменена и экземпляр компонента должен быть об этом оповещен
    - Deactivated – он был disabled или его зависимости ушли. Публикуемые им сервисы дерегистрируются, он становится доступным для сборщика мусора
  + Обработчики состояний описываются соответствующими атрибутами элемента component: activate, deactivate, modified.
    - Activate / deactivate методы можно не описывать, дефолтные наименования activate / deactivate. Modified метод должен быть описан явно
    - Допустимые типы параметров
      * ComponentContext
      * BundleContext
      * Map
      * Int – причина события
        + 0—Unspecified.
        + 1—The component was disabled.
        + 2—A reference became unsatisfied.
        + 3—A configuration was changed.
        + 4—A configuration was deleted.
        + 5—The component was disposed.
        + 6—The bundle was stopped.
  + Для каждого активизированного компонента создается экземпляр ComponentContext
    - public Dictionary getProperties();
    - public Object locateService(String name);
    - public Object locateService(String name, ServiceReference reference);
    - public Object[] locateServices(String name);
    - public BundleContext getBundleContext();
    - public Bundle getUsingBundle();
    - public ComponentInstance getComponentInstance();
    - public void enableComponent(String name);
    - public void disableComponent(String name);
    - public ServiceReference getServiceReference(); - доступ к ссылке на публикуемый компонентой сервис
    - методы семейства locateService – альтернативный подход к DI сервиса фрейморком, lookup стратегия, можно отложить получение (а значит и активацию) сервиса до последнего момент. В качестве String аргумента используется имя сервиса из декларативного определения зависимости.
* Конфигурирование
  + Компонент становится Satisfied когда доступны не только необходимые сервисы, но и данные конфигурации, необходимые для функционирования. Управляется свойством компонент configuration-policy
    - Optional – дефолтный, данные ConfigurationAdmin используются если доступны
    - Require – данные из ConfigurationAdmin должны быть доступны для старта компоненты. Используется, когда нет разумного дефолтного значения, например URL соединения с СУБД
    - Ignore – используются только значения из конфигурационного файла
  + Порядок применения к компоненте конфигурационных значений, верхние перезаписывают нижние.
    - Свойства, переданные при динамическом создании за счет ComponentFactory.newInstance()
    - Свойства, переданные сервису за счет ConfigurationAdmin – настраиваемые администратором системы
    - Свойства, описанные в дескрипторе компоненты – дефолтные
  + Если ConfigurationAdmin использует PID сервиса для создания единичной конфигурации, то создается один экземпляр объекта. Если используется **createFactoryConfiguration**, то создается набор компонент – легковесный путь создания множественных экземпляров компонента из одного метаописания, альтернативный – фабрика компонент
* Если компонента предоставляет сервис, то она активируется когда сервис кому-то понадобится. Если нет, то придется в описании компоненты ставить immediate=”true”. Это опционально для сервис-компонент и обязательно для компонент, которые не регистрируют сервис.
* Фабрика компонент – программный способ создания нескольких экземпляров компоненты из одного описания
  + В описании компоненты должно быть описано наименование фабрики factory=<factoryName>, свойство «name» по прежнему обязательно указывать для разрешения зависимости
  + Клиент определяет зависимость от ComponentFactory и создает новый экземпляр компонента за счет сomponentFactory .newInstance(Map переопределяемых свойств). В обязательном для переопределении свойстве «name» надо указывать наименование нового компонента

## Blueprint

* Вводится в OSGi R4.2 Enterprise спецификации базирующейся на Spring/OSGi интеграции, произведенной в Spring Dynamics Modules. Есть несколько реализаций, включая Apache Aries и Eclispe Gemini.
* OSGi альянс поддерживает одновременно два стандартных механизма: Declarative Services и Blueprint. Но возможно их одновременное совместное использование. С точки зрения альянса лучше иметь несколько сообществ, стандартизировавших свои подходы, чем определять единый подход для всех. Разница:
  + Declarative Services для построения легковесных компонент с быстром временем вхождения в технологию
  + Blueprint для высоко конфигурируемых enterprise ориентированных компонент.
* Особенность Blueprint:
  + каркас расширителя, в который встраиваются отдельные модули, каждый из которых ответственен за расширение в каком-то аспекте.
  + Blueprint ответственен за декларативное описание и инициализацию объектов, а модули поставщики обрабатывают эти объекты, подключая их к OSGi.
  + Каждый модуль получает инструкции за счет XML элементов в своем пространстве имен.
* Каждый Blueprint компонент содержит ряд элементов, каждым из которых управляет отдельный менеджер в рамках контейнера
  + **Bean** – примерно то же самое, что и в Spring.
    - Конструкторы и статические методы фабрики.
    - Синглетоны и прототипы
    - DI аргументов конструктора и значений свойств
    - Callback фаз ЖЦ – активация и деактивация
    - Доступные средства Spring для определения свойств бинов – value, ref. Idref, map, props, list, set, array
  + **Reference** – доставляет один сервис из OSGi реестра до компонента, базируясь на интерфейсе и фильтре на свойства сервиса
  + **ReferenceList** – доставляет компоненту не один сервис, а набор подходящих по интерфейсу / фильтру сервисов
  + **Service** – позволяет компоненту опубликовать сервис
  + **Environment** – предоставляет компоненту доступ к контекстам OSGi и Blueprint
* Blueprint bundle – содержит XML файл с описанием процесса создания, обработки и подключения компонентов друг к другу и к инфраструктуре OSGi. XML файлы конфигурации указываются в заголовке манифеста Bundle-Blueprint, возможны следующие варианты
  + Абсолютный путь к файлу или файлам в bundle, например OSGI-INF/cicrcle.xml, могут использоваться «\*»
  + Директория, значение должно кончаться на «/»
  + Части описания компонента, так же как и в Spring, может располагаться в различных файлов
  + Файлы могут располагаться как в основном bundle, так и в его фрагментах. Способ иметь разные конфигурации компонентов в зависимости от перечня загруженных bundles
* Регистрация сервиса

**<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">**

**<bean id="circle" class="org.foo.shape.circle.Circle" />**

**<service id="shape" interface="org.foo.shape.SimpleShape" ref="circle">**

**<service-properties>**

**<entry key="simple.shape.name" value="Circle"/>**

**<entry key="simple.shape.icon">**

**<bean class="org.foo.shape.circle.IconFactory" factory-**

**method="createIcon"/>**

**</entry>**

**</service-properties>**

**</service>**

**</blueprint>**

* + В отличии от DI значения свойств сервиса могут быть произвольными классами, в частности этот производится при помощи фабрики

**public class IconFactory {**

**public static ImageIcon createIcon() {**

**return new ImageIcon(IconFactory.class.getResource("circle.png"));**

**}**

**}**

* + Интерфейсы публикуемого сервиса могут задаваться в атрибуте, во вложенных элементах <interfaces><value> и при помощи атрибута auto-export, который позволяет контейнеру автоматически вычислить перечень:
    - Disabled – дефолтное значение
    - Interfaces – все имплементированные публичные интерфейсы класса и суперкласса
    - Class-heirarchy – класс реализации и все супеклассы до но не включая Object
    - All-classes – все классы, суперклассы, интерфейсы класса и суперклассов
    - Вообще крайне не рекомендуется использовать классы как сервисы, так как Blueprint доставляет сервисы компонентам в виде динамических прокси ((java.lang.reflect.Proxy) для создания которых необходимы интерфейсы
* Потребление сервиса через прокси
  + Вместо передачи компонентам прямых ссылок на сервисы, Blueprint создает прокси для сервисов, позволяя
    - получать сервисы по мере их реальной необходимости
    - прозрачно для компонент защищаться от поднимающихся / опускающихся сервисов, оперативно переключая ссылки на активный в данный момент сервис / набор сервисов
  + потребление сервиса

**public class B {**

**private A a;**

**public void setService(A a) { this.a = a }**

**public void someAction() { a.doit(); }**

**}**

**<reference id="a" interface="A"/>**

**<bean id="b" class="B">**

**<property name="service" ref="a"/> или <argument ref="a"/>**

**</bean>**

* + потребление набора сервисов

**public class D {**

**private List<A> list;**

**public void setServices(List<A> list) { this.list = list }**

**public void someAction() {**

**for (A a : list) {**

**a.doit();**

**}**

**}**

**}**

**<reference-list id="a" interface="A"/>**

**<bean id="d" class="D">**

**<property name="services" ref="a"/>**

**</bean>**

* + у reference / reference-list есть атрибут availability=optional | mandatory, определяющий необходимость наличия сервиса для жизнеспособности компонента
  + Такой подход не может решить все проблемы, например
    - Сервис может опуститься а взамен ему может никто не появиться. В этом случае прокси заблокирует нить вызывающего метода и через некоторый таймаут выдаст ServiceUnavailableException. Время ожидания до возбуждения исключения регулируется атрибутом timeout в миллисекундах, 0 – ждать вечно
    - То же самое относится к списку сервисов, но там дополнительно возникает проблема, когда hasNext() / getNext() уже вызван, а кандидат опустился. Вызывающий код получает прокси, смотрящий на отсутствующий сервис и соответственно ServiceUnavailableException. А вот во время итерации по меняющемуся на ходу набору напротив не получает исключения ConcurrentModificationException
* Reference listeners
  + Позволяют компонентам получать ссылки на сервисы посредством bind / unbind методов
  + Пример

**public class E {**

**private volatile A a;**

**public void addService(A a) { this.a = a }**

**public void removeService(A a) { this.a = null }**

**public void someAction() {**

**A a = this.a;**

**if ( a != null ) a.doit();**

**}**

**}**

**<bean id="e" class="E"/>**

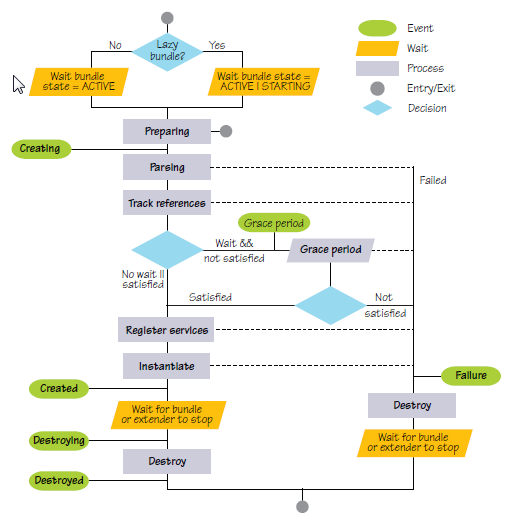
**<reference id="a" interface="A">**

**<reference-listener bind-method="addService"**

**unbind-method="removeService" ref="e"/>**

**</reference>**

* + Допустимые сигнатуры методов
    - public void <method-name>(ServiceReference)
    - public void <method-name>(<parameter-type>)
    - public void <method-name>(<parameter-type>, Map)
  + В отличии от DS bind / unbind методы должны быть только public. Возникает опасность, что сторонний код используя отражение может воспользоваться данными методами, даже если менеджер безопасности это запрещает
  + Reference listener в отличии от reference не налагает обязательное / опциональное требование на наличие сервиса для жизнеспособности компоненты
* Жизненный цикл



* + В определении бина есть init-method / destroy-method, вызываемые как callback при переходе компонента по ЖЦ
  + Если обязательная reference зависимость от сервиса становится битой, то и сам компонент через какое-то время становится невалидным. Blueprint контейнер автоматически останавливает все сервисы, публикуемые данным компонентом. Любые попытки других компонент использовать эти сервисы приводят к блокировке потоков и возбуждению ServiceUnavailableException
    - Плюсы – такой подход ликвидирует волны publish / unpublish событий, порождаемых обновлениями bundles
    - Минусы – приводит к возникновению непредвиденных блокировок, что особенно неприятно в GUI
  + Примерно такая же проблема с доступностью обязательных сервисов во время старта. Время ожидания регулируется свойством в заголовке bundle.

**Bundle-SymbolicName**: com.acme.foo; **blueprint.graceperiod**:=true; **blueprint.timeout**:=10000

Можно установить в false, тогда компонента будет создана без удовлетворения зависимостей, но ее сервисы опубликованы не будут. Соответственно обращающиеся к ним потоки будут блокированы.

* + В DS компоненты lazy по умолчанию, то есть они будут созданы тогда, когда будет первое обращение к публикуемому компонентой сервису. Здесь регулируется атрибутом bean.activation=eager|lazy.
* Связь с окружением
  + Посредством элемента <property name=”” ref=””> и <argument ref=””> стандартные элементы окружения доставляются компоненте
    - blueprintContainer – доступ к blueprint контейнеру и метаописаниям компонент. Основные методы
      * void Set getComponentIds();
      * Object getComponentInstance(String id);
      * ComponentMetadata getComponentMetadata(String id)
      * Collection getMetadata(Class type);
    - blueprintBundle – bundle содержащий компоненты
    - blueprintBundleContext - bundleContext
    - blueprintConverter – стандартный конвертер значений свойств для <property>
  + внутри контейнерных тэгов определений значений свойств можно использовать определения свойств за счет <service>
  + scope – singleton / prototype
    - для бина – либо один экземпляр для всех, либо создание нового экземпляра на каждый вызов контейнера getComponentInstance()
    - для сервиса – один экземпляр сервиса для всех или для каждого bundle, аналог ServiceFactory
  + кастомная конвертация строковых значений в объекты
    - зарегистрировать свой конвертер

**<type-converters>**

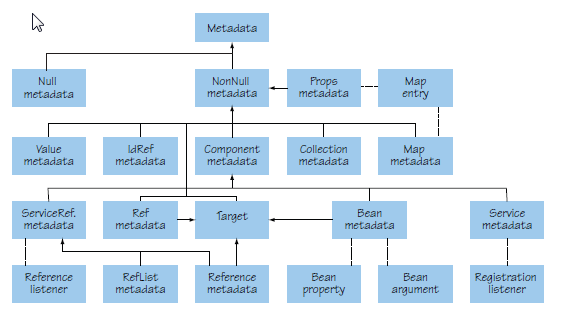
**<bean class="AtomicConverter">**

**<argument ref="blueprintConverter"/>**

**</bean>**

**</type-converters>**

* + - реализовать методы
      * canConvert(Object, ReifiedType)
      * convert(Object, ReifiedType)
    - blueprintConverter используется, чтобы сначала выполнить дефолтные преобразования, а потом уже попытаться преобразовать результат к тому, что необходимо получить
    - кастомные конвертеры разумно использовать, когда надо адаптировать компоненты третьих производителей
  + метаданные контейнера
    - в Spring используются чтобы создавать на лету необходимые компоненты, тут возможность ограниченна, так что скорее в справочных и отладочных целях



* Кастомные расширения для создания собственных компонент
  + Пример конфигурирования использования расширения за счет пространства имен и применения сконфигурированного расширения

**<?xml version="1.0" encoding="UTF-8" standalone="no"?>**

**<blueprint xmlns=**[**http://www.osgi.org/xmlns/blueprint/v1.0.0**](http://www.osgi.org/xmlns/blueprint/v1.0.0)

**xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"**

**xmlns:sab="http://www.manning.com/xmlns/blueprint-sab/v1.0.0"**

**xsi:schemaLocation="http://www.osgi.org/xmlns/blueprint/v1.0.0 http://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd http://www.manning.com/xmlns/blueprint-sab/v1.0.0**

[**http://www.manning.com/**](http://www.manning.com/) **xmlns/blueprint-sab/v1.0.0/sad.xsd" >**

**<bean id="handler" class="manning.enterpriseosgi.SimpleEventHandler" />**

**<!—создание нового компонента и регистрация его как слушателя очереди -->**

**<sab:subscriber id="sub1"**

**topic="org/osgi/framework/BundleEvent/\*" ref="handler" />**

**<bean id="handler1" class="manning.osgi.SimpleEventHandler" >**

**<!—декорирование метода существующего компонента транзакционным атрибутом-->**

**<tx:transaction method="handleEvent" />**

**</bean>**

**</blueprint>**

* + Чтобы добавить расширение в Blueprint
    - надо зарегистрировать сервис **NamespaceHandler** cо свойством **osgi.service.blueprint.namespace,** ссылающимся на соответствующее пространство имен

**<service interface="org.osgi.service.blueprint.namespace.NamespaceHandler">**

**<service-properties>**

**<entry key="osgi.service.blueprint.namespace"**

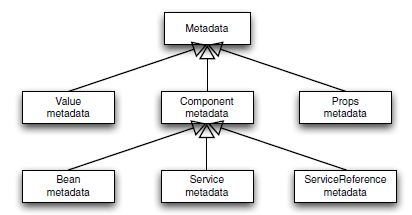
**value="http://www.manning.com/xmlns/blueprint-sab/v1.0.0"/>**

**</service-properties>**

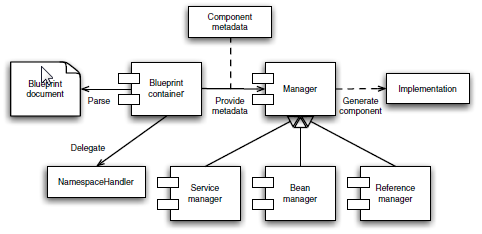
**<bean class="manning.enterpriseosgi.sab.SabNamespaceHandler" />**

**</service>**

* + - Дальше реализуем **NamespaceHandler** со стандартными методами
      * **public URL getSchemaLocation(String schema)** – возвращает XSD для контроля синтаксиса конфигурирования расширения. Этот файл храним в bundle расширения и возвращаем за счет **this.getClass().getResource("schemas/blueprint-sab.xsd")**
      * **public ComponentMetadata parse(Element element, ParserContext context)** - обработка целого XML элемента, то есть процесса создания нового компонент
      * **public ComponentMetadata decorate(Node node, ComponentMetadata component, ParserContext context)** – обработка декорирования компонента, создаваемого кем-то еще
  + Процесс работы Blueprint контейнера
    - Контейнер читает описания из всех подходящих bundle и генерирует метаданные



* + - Менеджеры компонент (билдеры) используют эти мета описания для создания экземпляров соответствующих компонент



* + Создание обработчиком пространства имен нового компонента
    - Например, если мы хотим на основании своего мета описания создать слушателя событий. В этом случае наше мета описание в виде XML элемента должно содержать всю необходимую информацию о создаваемом компоненте. Методу **ComponentMetadata parse(Element element, ParserContext context)** она прилетает в виде **Element**
    - Чтобы создать компонент надо создать его описание и отдать реестру Blueprint. Это такое же описание, которое мы создаем за счет <bean> или <service>. Например, для создание бина создаем его описание в виде **BeanMetadata**:
      * заполняем его свойства
        + **Scope** – singleton | prototype
        + **ClassName**
        + **InitMethod / DestroyMethod**
      * Определяем значения свойств создаваемого объекта за счет набора экземпляров **ValueMetadata**, которые будут возвращены **ComponentMetadata.getProperties().** Каждый экземпляр сообщает о себе тип и значение
      * За счет типа **RefMetadata** можно определить ссылку на другой бин. При этом используется либо ИД этого бина, либо одно из предопределенных стандартных значений
        + **blueprintBundleContext** – контекст bundle, содержащего blueprint XML описание
        + **blueprintContainer** – внутренний реестр описаний метаданных и компонентов
        + **blueprintBundle** – описание bundle, содержащего blueprint XML описание
        + **blueprintConverter** – конвертер Blueprint компонент и свойств
    - Дальше Blueprint по полученному описанию инстанцирует экземпляр класса и проведет его инициализацию
    - Реальная работа производится методами **init / destroy** класса, описание экземпляра которого создается методом **parse().** Например в init мы подписываем объект на события, а в destroy отсоединяем слушателя.
  + Расширение обработчиком пространства имен уже существующего компонента
    - Например, если мы хотим добавить транзакционность к вызову метода уже существующего компонента.
    - Делается это в методе

**public ComponentMetadata decorate(Node node, ComponentMetadata component,**

**ParserContext context)**

за счет интерцептора, регистрируемого на мета данных компонента при помощи метода

**context.getComponentDefinitionRegistry().registerInterceptorWithComponent(component, interceptor);**

мы НЕ создаем новые метаданные, но подправляем переданные нам мета данные внешнего по отношению к нам описания

* + - Интерцептор должен реализовывать методы интерфейса Interceptor
      * **public Object preCall(ComponentMetadata metadata, Method m, Object... args)**
      * **public void postCallWithException(ComponentMetadata metadata, Method m, Throwable t, Object correlator)**
      * **public void postCallWithReturn(ComponentMetadata metadata, Method m, Object ret, Object correlator)**
    - На одном методе может быть определено множество интерцепторов, сначала срабатывают все preCall, затем все пост-методы. Корреляция между пре и пост методами осуществляется за счет объектов-корреляторов, возвращаемых пре-методами и передаваемых пост-методам
    - Интерцептор можно создавать полностью внутри обработчика, а можно использовать ссылку на отдельно определяемый бин-интерцептор

## Apache Felix IPOJO

* Разница между IPOJO и предыдущими решениями
  + IPOJO работает на основе инструментирования байт кода, что облегчает многие вещи
  + Метаданные компонент могут быть определены на основе любого из возможных подходов (XML, аннотации, API)
  + Высокая степень расширяемость фреймворка, разработчики могут дополнять решение своими хэндлерами, добавляя недостающую функциональность
  + В DS и Blueprint в метаданных описываются экземпляры компонент, тут отдельно описываются типа компонент, а отдельно их экземпляры на основе ранее описанных типов
* Так как применяется инструментирование кода, то кроме компиляции появляется дополнительный этап с сборке проекта, есть ANT задача на вход которой подается jar файл с будущими компонентами, подлежащий обработке, и опционально XML файл, в котором перечислены создаваемые экземпляры компонент (сами описания типов компонент могут быть в любом из трех видов)
  + Инструментированию подвергаются вызовы методов и доступ к полям классов
  + Инструментирование не меняет поведение компонент, они могут как и раньше использоваться обычным образом вне контейнера, правда возникают дополнительные зависимости кода
  + Кроме инструментирования на этом этапе производится сборка и запись описаний типов компонент в стандартную форму
    - Если мы используем XML для описания, то парсер нужен только на время сборки, во время выполнения он уже не нужен
    - Если мы используем аннотации, то они нужны только во время сборки, результат может быть использован в ранних версиях JVM, не поддерживающих аннотации
    - Загрузка заранее подготовленных метаданных во время выполнения осуществляется быстрее и более эффективно
* Описание компонента, публикующего сервис

**@Component(immediate=true)**

**@Provides**

**public class Circle implements SimpleShape {**

**@ServiceProperty(name=SimpleShape.NAME\_PROPERTY)**

**private String m\_name = "Circle";**

**@ServiceProperty(name=SimpleShape.ICON\_PROPERTY)**

**private ImageIcon m\_icon = new ImageIcon(this.getClass().getResource("circle.png"));**

**public void draw(Graphics2D g2, Point p) {**

**...**

**}**

**}**

* + Значения свойств, помеченных как @ServiceProperty, автоматически проносятся в реестр сервисов OSGi. Более того, если компонент изменяет значения этих свойств во времени выполнения, то описания сервиса в реестре обновляется
  + Если компонент создается отложенно, когда кто-то запрашивает сервис, то с получением свойств для регистрации сервисов возникает логическая проблема. Выходы:
    - Первоначально сервис регистрируется с пустыми свойствами и только после первого запроса они обновляются
    - В аннотации можно задать дефолтные значения свойств
    - Посредством immediate можно подавить создание по требованию
  + В наименованиях свойств, сервисов и так далее можно использовать статические константы, что повышает согласованность кода с метаданными по сравнению с XML описаниями и прочими способами
  + По умолчанию сервис регистрируется со всеми возможными интерфейсами, посредством @Component(specification) можно задать конкретный интерфейс / интерфейсы
* Потребление сервисов
  + Возможно через внедрение методов и полей. Оба подхода могут использоваться одновременно. В большинстве случаев пофигу что использовать для внедрения методы или поля. Но если хотим, чтобы насуведомляли о появляении / исчезновении сервисов, то приходится ограничиваться методами. Но можно оба подхода совместить
  + Возможности: фильтр на интерфейс сервиса, опциональность / обязательность, один / много сервисов, LDAP фильтр на свойства запрашиваемых сервисов, каким образом будет обрабатываться динамическая природа сервиса (binding policy), использование прокси при внедрении свойств
  + Проксирование сервисов
    - Создание прокси может осуществляться как за счет инструментирования кода, так и за счет динамических прокси. На большинстве платформ используется первый подход, а значит можно пренебрегать интерфейсами и выше скорость работы. На андроиде, к примеру, инструментирование не возможно, а значит интерфейсы обязательны
    - В отличие от Blueprint эти прокси не блокируют вызывающий поток в случае отсутствия сервиса. Типовой шаблон – null или дефолтный объект, выступающий mock заглушкой.
  + Внедрение через методы определяется с помощью @Bind / @Unbind
    - Поддерживаемые сигнатуры методов
      * void <method-name>()
      * void <method-name>(ServiceReference ref)
      * void <method-name>(<S> svc)
      * void <method-name>(<S> svc, ServiceReference ref)
      * void <method-name>(<S> svc, Map props)
      * void <method-name>(<S> svc, Dictionary props)
    - в первых двух методах тип желаемого сервиса определяется через атрибут specification аннотации, в остальных случаях тип может быть определен автоматически
    - атрибут аннотации filter ответственен за LDAP фильтрацию потребляемых сервисов
    - атрибут аннотации aggregation ответственен за то, внедряется ли один подходящий сервис или все подходящие по очереди
    - парность методов bind / unbind определяется автоматически, по остатку наименования (начало должно быть стандартное bind или unbind). Если схема наименования отличается или адаптируются унаследованные методы, то применяется атрибут id
  + внедрение через поля класса определяется с помощью @Requires и атрибута filter, который задает фильтр на желаемый сервис
    - может применяться к коллекциям и массивам
    - код фреймворка сам корректно обрабатывает асинхронную природу события, собственные игры с AtomicReference и так далее не требуются
    - по умолчанию если подходящий сервис более не существует, то внедряется автоматически создаваемый null объект, методы которого ничего не делают, а возвращают false / 0 / null в зависимости от сигнатуры метода.
      * Можно запретить посредством атрибута nullable=false. Тогда при наличии прокси надо быть готовым к исключению, без прокси – к null значению
      * Посредством атрибута default-implementation можно предложить собственную реализацию null объекта
    - IPOJO гарантирует, что значение поля компонента будет одним и тем же в течении всего времени, которое поток проводит в вызванном методе компонента, даже если управление покидает на время этот метода / компонент. Это сильно облегчает взаимодействие со statefull сервисами
* Жизненный цикл
  + Динамическая природа ЖЦ по отношению к потребляемому сервису определяется атрибутом @Requires(policy)
    - Static – когда компонент создается он снабжается конкретными экземплярами доступных сервисов. В дальнейшем он не замечает появления других экземпляров, а если эти опускаются, то погибает. Код компоненты проще – не надо заморачиваться реакцией на изменения.
    - Dynamic – вариант по умолчанию, компонент получает события о появлении новых сервисов без учета приоритетов
    - Dynamic-priority – получает события, только если появляющиеся сервисы более приоритетны
  + Временные зависимости от сервисов
  + Выбиваются из деления обязательные / опциональные. Например – сервис нужен на время инициализации, а потом нужда в нем отпадает – не опциональный, но и не обязательны.
    - iPOJO поддерживает временные зависимости от сервисов, которые не влияют на ЖЦ – вызывающий поток блокируется до получения сервиса. Вместо аннотации org.apache.felix.ipojo.annotations.Requires надо использовать аннотацию rg.apache.felix.ipojo.handler.temporal.Requires.
      * Атрибут timeout определяет период ожидания
      * Атрибут onTimeout позволяет вернуть null значение, null объект или дефолтную реализацию
  + Аннотации @Validate / @Invalidate позволяют отреагировать на активацию / деактивацию (с последующей сборкой мусора) компонента в ответ на пропавший обязательный сервис.
    - С @Invalidate надо быть осторожным, в этом состоянии какие-то сервисы уже могут отсутствовать
    - За счет @Controller на private boolean поле класса можно изнутри компонента программно его активировать / деактивировать просто за счет присвоения полю значения.
    - Аналогично @ServiceControl на поле позволяет управлять публикацией сервиса, предоставляемого компонентом. По умолчанию это относится ко всем регистрациям, за счет атрибута specification можно поднимать / опускать только определенные сервисы
  + Если у компонента есть дефолтный конструктор с BundleContext, то в процессе создания будет передан соответствующий параметр
* Создание компонентов IPOJO
  + В отличии от Declarative Services и Blueprint, где совмещены в одном описания типа компоненты и экземпляра компоненты, тут они разнесены. Разница примерно такая же, как между классом и объектом.
  + Статический XML
    - ANT задаче ipojo передается путь к файлу описания перечня создаваемых компонент.
    - Файл не обязательно должен содержать описания компонент, определенных в этом же bundle. Так что можно несколько bundle с компонентами, а потом один с описаниями экземпляров на основе всех предыдущих определений.
    - В файле

**<ipojo>**

**<instance component="идентификатор определения компоненты">**

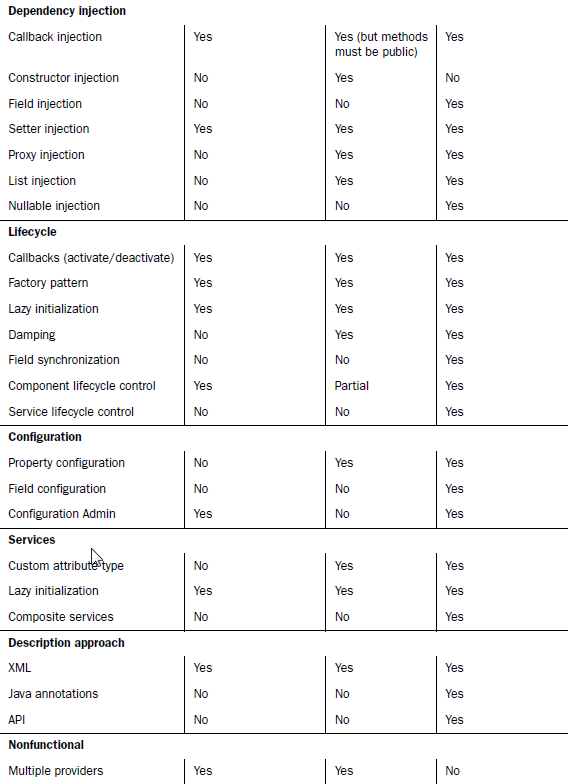
**<property name=”” value=””/>**

**</instance>**

**</ipojo>**

* + - На основе одного определения можно сделать несколько экземпляров, у каждого надо определить уникальное свойство «name»
    - Для работы со свойствами поддерживаются list, map, arrays, set, dictionary
  + Аннотация @Instance
    - Подобно определению синглетона в коде
    - Недостаток – страдает повторное использование, потому что фиксируется число и свойства компонент, нельзя варьировать в зависимости от сценариев и конфигурации
  + Программно через IPOJO фабрику
    - Для каждого определения компоненты IPOJO регистрирует сервис с интерфейсом org.apache.felix.ipojo.Factory и уникальным свойством factory.name.
    - Дальше мы получаем сервис из реестра OSGi и создаем экземпляр компоненты используя метод factory.createComponentInstance(props). Не забываем в числе прочих свойств передать уникальное имя
    - В ответ получаем ComponentInstance, который используем для управления созданным компонентом.
      * instance.reconfigure(props)
      * instance.dispose()
    - минус – привязываемся к проприетарному Felix API
  + Программно через ConfigurationAdmin - создаем одиночную конфигурацию или фабрику конфигураций, заполняем свойства

## Сводные характеристики



* Можно в одном приложении мешать компоненты разных типов, так как основа интероперабельна
* Тем не менее, есть некоторые проблемы
  + DeclarativeServices – свойства компонент могут быть только примитивами, чтобы просунуть ресурс надо чтобы клиент по строковому url и классу сервиса самостоятельно его грузил
  + Blueprint – использует proxy объекты, поэтому клиенты не смогут получить реальный класс сервиса, перейти к его classlodaer и что-то загрузить из bundle поставщика сервиса
  + Blueprint требует, чтобы только интерфейсы публиковались как сервисы

# Встраивание контейнера

## Архитектура

* По отдельности Apache Felix, Equinox и Knopflerfish имеют свои проприетарные механизмы запуска и встраивания. Обычно это всякие комбинации свойств авторазвертывания, механизмы инстанцирования, оболочки с текстовым / графическим интерфейсом. Начиная с R4.2 спецификации альянс предлагает стандартные способы решения данной задачи
* Причины самостоятельно запускать контейнер
  + Специфические требования к запуску со стороны приложения, коорые не поддерживаются стандартными механизмами
  + Мы не хотим конвертировать все приложение в набор bundle, а хотим использовать контейнер для запуска плагинов
* В процессе функционирования OSGi контейнер представлен специальным bundle – системным, имеющим стандарный идентификатор 0.
  + Интерфейс стандартного bundle
    - Доступ к содержимому
      * String getSymbolicName();
      * Version getVersion();
      * Dictionary getHeaders();
      * Dictionary getHeaders(String locale);
      * URL getEntry(String path);
      * Enumeration getEntryPaths(String path);
      * Enumeration findEntries(String path, String pattern, boolean recurse);
      * URL getResource(String name);
      * Enumeration getResources(String name) throws IOException;
      * Class loadClass(String name) throws ClassNotFoundException;
      * Map getSignerCertificates(int signersType);
    - Контроль ЖЦ
      * void start() throws BundleException;
      * void start(int options) throws BundleException;
      * void stop() throws BundleException;
      * void stop(int options) throws BundleException;
      * void update() throws BundleException;
      * void update(InputStream input) throws BundleException;
      * void uninstall() throws BundleException;
    - Получение информации
      * int getState();
      * BundleContext getBundleContext();
      * long getBundleId();
      * String getLocation();
      * long getLastModified();
      * ServiceReference[] getRegisteredServices();
      * ServiceReference[] getServicesInUse();
      * boolean hasPermission(Object permission);
  + Системный bundle реализует расширенный интерфейс Framework, добавляющий методы
    - void init() throws BundleException;
    - FrameworkEvent waitForStop(long timeout) throws InterruptedException;
    - При этом спецификаций НЕ требует, чтобы интерфейсы Framework и системного bundle c идентификатором 0, видимого изнутри, были реализованы одним и тем же объектом

## Основные операции

* Создание экземпляра контейнера
  + Основная проблема – мы не знаем какой класс инстанцировать и как его запускать.
  + Решение стандартное для Java платформы через определение провайдера сервиса в META-INF/services.
    - Там набор файлов, каждый из которых носит наименование интерфейса сервиса и содержит внутри наименование файла реализации интерфейса. Например, файл META-INF/services/java.text.spi.DateFormatProvider содержит наименование класса реализации org.foo.CustomDateFormatProvider
    - Вместо того, чтобы впрямую создавать фреймворк используется фабрика с интерфейсом FrameworkFactory и единственным методом Framework newFramework(Map config), которому передаются свойства создаваемого инстанса.
    - Фабрика аналогичным образом создается через механизм сервисов

**ServiceLoader<FrameworkFactory> factoryLoader =**

**ServiceLoader.load(FrameworkFactory.class);**

**FrameworkFactory factory = factoryLoader.iterator().next();**

**Framework fw = factory.newInstance(Map)**

* + - Если используем более раннюю Java, то через ClassLoader.getResource() получаем доступ к файлу META-INF/services/org.osgi.framework.launch.FrameworkFactory и за счет отражения создаем экземпляр, у которого
  + Как правило, при создании фреймворк надо сконфигурироваться, хотя бы задать расположение кэша и текущую директорию. Это делается через свойства, наименования большинства специфично для реализации, но некоторые стандартизированы
    - org.osgi.framework.storage – расположение кэша
    - org.osgi.framework.storage.clean – если onFirstInit - очистить кэш перед первым использованием
    - org.osgi.framework.system.packages и org.osgi.framework.system.packages.extra – перечень пакетов, коорые экспортируются из системного bundle
    - org.osgi.framework.startlevel.beginning – начальный уровень запуска контейнера
    - org.osgi.framework.bootdelegation – перечень пакетов, которые видны всем bundle без указания в импорте, по умолчанию java.\*. Лучше не использовать
    - org.osgi.framework.bundle.parent – какой рутовы загрузчик классов использовать для контейнера
      * boot – дефолтное значение
      * app – загрузчик приложения
      * ext – загрузчик расширений
      * framework – загрузчик, которым грузили фреймворк
    - org.osgi.framework.library.extensions – дополнительные библиотеки для поиска нативного кода
    - org.osgi.framework.command.execpermission – команды операционной системы для установки разрешений нативному коду
* Запуск экземпляра контейнера
  + Два основных метода:
    - init() переводит системный bundle в STARTING состояние. Цель этапа – сделать доступным BundleContext, чтобы можно было управлять контейнером до запуска основных bundles.
      * разрешается обработка событий
      * устанавливается менеджер безопасности
      * уровень устанавливается в 0
      * закэшированные bundle загружаются и устанавливаются в INSTALLED
      * создается BundleContext для контейнера
      * системные сервисы становятся доступными
    - start() в ACTIVE
      * если конейнер еще не в STARTING состоянии, то вызывается init()
      * стартовый уровень повышается до заданного значения, в процессе этого активируется ряд bundle
      * рассылается сообщение FrameworkEvent.STARTED
* Остановка экземпляра – Bundle.stop() на системном bundle. Метод асинхронный, подождать пока реально остановится (а также просто подождать остановки в любом случае, даже сразу непосредственно после запуска) можно через Framework.waitForStop(). Он принимает timeout и возвращает причину остановки контейнера
  + STOPPED – был остановлен
  + STOPPED\_UPDATE – был обновлен
  + ERROR – ошибка вынудила остановиться или ошибка во время остановки
  + WAIT\_TIMEOUT – таймаут истек, но фреймворк еще работает
* Цикл start / init <-> stop можно повторять несколько раз, если onFirstInit кэш будет создан только один раз
* Типовые трюки
  + Запустив контейнер можно посредством Runtime.getRuntime().addShutdownHook установить системный хук, который его остановит при остановке JVM. Надо не только отдать команду остановиться, но и дождаться конца.
  + Запуская контейнер в свойства, передаваемые ему, добавить все системные и аргументы командной строки
  + Сначала все bundle приложения инсталлируем, затем уже стартуем, чтобы не вляпаться в зависимости. Автоматизируя старт bundles приложения надо не забывать про фрагменты, которые самостоятельно не могут быть запущены. Их отделяем через bundle.getHeaders().get(Constants.FRAGMENT\_HOST)
  + Аналогично можно найти все bundle, в манифесте которых упомянут main класс, и запустить main процедуры из них – альтернативный способ запуска без активатора
  + Поток, запустивший контейнер, сразу после запуска становится на waitForStop(0), на выходе вызывает System.exit(0), чтобы гарантированно выйти из JVM, даже если кто-то из прикладных bundle создал поток не-демона и забыл его собрать

## Встраивание OSGi контейнера внутрь приложения

* Основаная причина – удобное средство управления плагинами, берет на себя решение типовых проблем, авторы плагинов работают в стандартном окружении
* Приложение управляет плагинами через системный bundle и полученный им BundleContext. Также через BundleContext приложение может публиковать свои сервисы для плагинов, а также получать доступ к сервисам и ресурсам плагинов. Так как сервисы имеют динамическую природу, то приложение должно использовать средства типа Tracker для обработки событий
* Существенное ограничение: приложение и плагины должны использовать одни и те же классы и их версии. Варианты:
  + Используем только основные JVM пакеты
  + Экспортируем пакеты из приложения в контейнер за счет org.osgi.framework.system.packages и org.osgi.framework.system.packages.extra. При этом плагины должны явным образом импортировать эти пакеты
  + Используем отражение. Получаем ссылку на сервис через BundleContext.getAllServiceReferences() вместо of BundleContext.getServiceReferences(), чтобы контейнер не отфильтровал подходящие нам по доступным пакетам.
* Потенциальные проблемы
  + Приложение может использовать системные синглетоны, например, для обработки URL и загрузки ресурсов
  + Приложение может использовать Thread Context Class Loader, а код полагаться на его правильную установку и даже использовать его вместо своего собстенного. Разные OSGi контейнеры обходятся по-разному, некоторые его насильно заменяют на загрузчик bundle

# Безопасность

## Общие места

* Недостаток гибкости и открытости – компоненты приложения доступны для атаки со стороны третьих bundles – могут перехватывать вызовы, переопределять зависимости, изменять конфигурацию и так далее. К счастью Java содержит механизм песочницы, который может быть задействован средствами OSGi
* Если мы не учитываем безопасность в коде компонент, то лишаем разработчиков возможности использования их в защищенном окружении. Если учитываем, то выполняемые проверки и работа с безопасностью повлияет на производительность и сложность кода
* Управление правами зачастую настолько сложное, что разработчики не в состоянии сопоставить их с выполняемыми кодом задачами и правильно распределить. В этом случае безопасность выступает скорее как препятствие
* Ключевые моменты безопасности
  + идентификация провайдера модуля, которому предоставляются привилегии. Основные подходы: на основании цифровой подписи кода ина основании места из которого загрузили код
  + формирование политики на основе привилегий, связывание наборов привилегий с идентифицированными модулями
  + выполнение проверок на наличие у вызывающего кода необходимых привилигий

## Java безопасность

* два типа безопасность
  + основанная на доменах безопасность – совокупность кода одинакового происхождения (code base). Полностью поддерживается OSGi с некоторой надстройкой поверх
  + основанная на аутентификации ролей или процессах, которым выдаются роли. Стандартный подход – JAAS, но OSGi предоставляет собственных механизм основанный на User Admin Service
* Базовые объекты
  + фрагментам кода присваиваются привилегии в виде наследников java.security.Permission.
    - Обычно конструктор принимает два параметра: name – имя объекта и actions – перечень действий, которые могут быть с ним совершены.
      * Например, для создания java.io.FilePermission необходимо указать путь и набор допустимых действий - READ, WRITE, EXECUTE или DELETE
      * В OSGi данная привилегия имеет свои особенности
        + В JVM путь интерпретируется относительно директории, где процесс был стартован.
        + В OSGi это не имеет смысла и указывается относительно личной директории bundle. Вообще-то, контейнер и так предоставляет bundle все права на эту директорию, так что имеет смысл добавлять только права на выполнение файлов
    - Кроме того даный класс реализует метод Permission.implies(Permission1), который проверяет является ли Permission1 подмножеством исходного Permission. Объекты Permission используются и для выдачи и для проверки привилегий
  + фрагментам кода представлены в системе как java.security.ProtectionDomain, описывающие свойства фрагмента относительно концепции безопасности.
    - Привилегии присваиваются ProtectionDomain и всем классам в него входящим.
    - В OSGi домен один к одному соответствует bundle, так как весь код одного bundle грузится из одного jar
  + Привилегии и домены в JVM складываются в файле определения политики безопасности (policy file), который выглядит примерно так:

**grant signedBy "sysadmin" {**

**permission java.security.AllPermission;**

**};**

**grant codeBase "file:/app/main.jar" {**

**permission java.security.AllPermission;**

**};**

**grant {**

**permission java.io.FilePermission "/tmp/\*", "read,write";**

**};**

* Когда вызывается код JVM анализирует ProtectionDomain вверх по стеку. Чтобы действие могло быть выполнено код каждого элемента стека должен иметь необходимые для этого привилегии.
  + Данная проверка вызывается посредством AccessController.checkPermission(Permission p) или методами SecurityManager
  + Принцип проверки всех элементов стека интуитивен, но черезчур ограничивающи. Например, в стеке неизбежно могут попадаться вызовы кода утилит третьих производителей, которым никто не выдавал необходимых привилегий. Выход – механизм привилегированного вызова (privileged call)
    - В данном случае проверяется не весь стек вызова вверх, а только до первого класса, обладающего данным свойством
    - Реализуется посредством AccessController.doPrivileged(), которому передается экземпляр PrivilegedAction, который подобно Runnable имеет метод run(), в котором выполняются необходимые действия
    - Таким образом, только привилегии домена защиты кода, запустившего PrivilegedAction, определяют возможность выполнения затребованных действий
  + Если мы хотим сделать сервис, выполняющий запись в лог, но не хотим предоставлять привилегию записи в файл всем его клиентам, то мы должны выполнить следующие действия
    - Определить привилегию AppendPermission, которую будем выдавать клиентам, которые хотят писать в лог
    - Предоставить реализации сервиса право записи в файл за счет FilePermission
    - Реализация должна, во-первых, проверять у вызывающего кода наличие AppendPermission, а, во-вторых, выполнять запись в файл за счет PrivilegedAction.

**public void append(String msg) {**

**SecurityManager sm = System.getSecurityManager();**

**if (sm != null) {**

**sm.checkPermission(new AppendPermission());**

**AccessController.doPrivileged(new PrivilegedAction() {**

**public Object run() { doFileAppend(); }});**

**} else {**

**doFileAppend();**

**}**

**}**

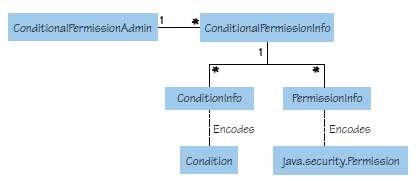
* + - Таким образом, если безопасность включена, то только код обладающий AppendPermission сможет вызвать функцию append, а функция записи в файл будет проверять стек только до кода сервиса, который имеет необходимые привилегии по записи в файл. Если же безопасность выключена, то влияния на производительность данный код иметь не будет

## OSGi безопасность

* OSGi определяет ряд стандартных наследников Permission, которые определяют поведение bundle на различных уровнях системы
* Уровень модулей
  + PackagePermission – определяет возможность bundle экспортировать / импортировать пакеты,
    - можно запретить bundle использовать какие-то классы или наоборот, предоставить только определенным bundle возможность экспортировать жизненно важные для системы сервисы
    - параметры
      * String name – наименование пакета или пакетов, допустимо использование «\*»
      * String actions – export | import (дает возможность и import указанный пакет) | exportonly
  + BundlePermission – контролирует возможность bundles зависеть друг от друга и использовать фрагменты
    - Параметры
      * String symbolicName – имя bundle, к которому применяется ограничение
      * String actions – что можно делать: provide (автоматически дает require), require, host, fragment
* Уровень ЖЦ
  + AdminPermission – дает возможность управлять другими bundle
    - Параметры
      * String filter – LDAP фильтер, определяющей перечень bundle, к которым применяются ограничения. В фильтре можно использовать не все свойства bundle, а только – signer, location, id, name
      * String action
        + \* это все действия
        + Class (загружать классы из), Execute (запуск, остановка, установка уровня), extensionLifecycle, lifecycle (инсталляция, обновление, удаление), listener ( работа с синхронным слушателем), metadata (получение заголовков и расположения), resolve ( Resolve / refresh), resource (загрузка ресурсов), startlevel (установка текущего и начального уровня), context (получение контекста от bundle)
* Уровень сервисов
  + ServicePermission – определяет какие сервисы bundle может публиковать или использовать
    - Параметры
      * String name – имя класса / интерфейса сервиса
      * String actions – get, register.
        + В имени можно определить несколько значений, можно использвать \*
        + Для get можно определить LDAP фильтр на используемый сервис, можно использовать те же самые свойства - signer, location, id, name
* В JVM присваивание привилегий коду статическое через файл политики, в OSGi привилегии присваиваются bundle на основании факта выполнения / не выполнения ряда условий над их свойствами. Такой подход лучше согласуется с динамическим характером OSGi окружения
  + Условия собираются в группы, которые привязываются к bundle.
  + Возможность выполнения кода определятся на основании анализа элементов стека, каждый из которых, относится к тому или иному bundle.
  + Код может быть выполнен, если проверка условий, относящихся к каждому bundle в стеке, дала в результате необходимый набор привилегий для выполнения действия
  + Каждый раз проверка всех условий, связанных с bundle, для каждого элемента стека, может быть весьма дорогим занятием. К счастью, ConditionalPermissionAdmin сервис позволяет снизить затраты за счет определения mutable / immutable условий, последние вычисляются один раз для каждого загруженного bundle
  + Проверки, применимые к одному bundle на основании условий, выполняются по очереди. Либо очередное вычисленное (а не отложенное) применительно к bundle дает необходимую привилегию, либо отнимает его.

## ConditionalPermissionAdmin

* Основные свойства
  + ConditionalPermissionInfo addConditionalPermissionInfo(ConditionInfo[] conds, PermissionInfo[] perms);
  + AccessControlContext getAccessControlContext(String[] signers);
  + ConditionalPermissionInfo getConditionalPermissionInfo(String name);
  + Enumeration getConditionalPermissionInfos();
  + ConditionalPermissionInfo setConditionalPermissionInfo(String name, ConditionInfo[] conds, PermissionInfo[] perms);
  + public ConditionalPermissionUpdate newConditionalPermissionUpdate();
  + public ConditionalPermissionInfo newConditionalPermissionInfo(String name, ConditionInfo[] conditions, PermissionInfo[] permissions,String access);
  + public ConditionalPermissionInfo newConditionalPermissionInfo(String encodedConditionalPermissionInfo);
* модель данных



* + ConditionalPermissionInfo – основной элемент политики безопасности, сочетает в себе набор условий, которые должны выполниться, и набор привилегий, которые в этом случае будут предоставлены
  + Вспомогательные классы ConditionInfo и PermissionInfo с текстовыми значениями (имен классов условий / привилегий и значений параметров конструкторов) задействованы, так как вследствии модульной природы bundle может не иметь доступ к соответствующим классам.
  + Пример создания PermissionInfo
    - * new PermissionInfo(AdminPermission.class.getName(), "(id>10)",

AdminPermission.EXECUTE);

* + - * право на запуск / остановку bundle с id>10
  + пример создания ConditionInfo
    - * + new ConditionInfo(BundleLocationCondition.class.getName(),

new String[] { "\*://foo.org/\*" });

* + - * + проверить, что код загружен с домена foo.org используя любой протокол
* Основные стандартные условия
  + основные типы условий BundleLocationCondition и BundleSignerCondition соответствуют аналогам в файле политики JVM
  + BundleLocationCondition
    - В конструкторе BundleLocationCondition можно использовать «не из расположения» - new String[] { "file:bundle/foo.jar", "!" });
  + BundleSignerCondition
    - Позволяет выдать права не конкретному bundle, а всем bundle одного поставщика
      * повышает уровень абстракции
      * однозначно идентифицирует поставщика и предохраняет код от модификации
    - условие вычисляется посредством сравнения фильтра на DN и DN сертификата, которым подписан bundle

**ACCEPT {**

**[org.osgi.service.condpermadmin.BundleSignerCondition "CN=core,O=baz,C=de"]**

**...**

**}**

* + - * опционально в конструкторе можно исползовать «!», аналогично предыдущему случаю
      * для сравнения используется максимально уточненный сертификат из цепочки, сертификаты выше него в сравнении не используются
    - Можно использовать «\*»
      * CN=\*,O=baz,C=de и даже CN=\*,O=baz,C=\* - подпись организации baz в германии или в любой стране
      * \*,O=baz,C=de – подпись либо сертификатом baz в германии с любыми уточняющими атрибутами
      * \*;CN=core,O=baz,C=de – условие на цепочку сертификатов, в данном случае это любой сертификат. «-» выступает в качестве «0 или более»
    - В сравнении участвуют только доверенные сертификаты или сертификаты подписанные доверенными сертификатами.
      * До R4.2 стандартного определения источника доверенных сертификатов не было, сейчас это org.osgi.framework.trust.repositories
      * Данное хранилище только для чтения и НЕ должено быть с паролем, а потому в него можно помещать только сертификаты (и не помещаться какие-либо ЗК)
* Программное определение политики безопасности через ConditionalPermissionAdmin
  + Когда OSGi контейнер стартует в первый раз все bundle первоначально имеют AllPermission, это значит, что любой из них может использовать ConditionalPermissionAdmin чтобы установить какую-то политику, в том числе запрещающую остальным bundle сделать то же самое. Будучи установленной, политика переживает перезагрузки контейнера. Bundle, намеревающийся установить политику безопасности, должен учитывать возможность conditional race и отлупа, так как у него уже отобрали необходимый для использования ConditionalPermissionAdmin сервиса AllPermission
  + Шаблон использования
    - Посредством организации сессии, представленной объектом ConditionalPermissionUpdate
    - Сначала открываем сессию за счет ConditionalPermissionAdmin.newConditionalPermissionUpdate()
    - У ConditionalPermissionUpdate есть методы, позволяющие редактировать текущую политику безопасности
      * getConditionalPermissionInfos()
      * clear() – рекомендуется вызывать с самого начала, чтобы создавать политику с чистого листа
      * add(ConditionalPermissionInfo)
      * commit
    - объекты ConditionalPermissionInfo создаются за счет ConditionalPermissionAdmin. newConditionalPermissionInfo(), которой передаются параметры
      * наименование элемента политики – уникальное значение, используемое для идентификации и нигде в дальнейшем не применяющееся. Если передать null, то будет сгенерировано автоматически
      * перечень условий в виде ConditionInfo[], элементы которого создаются через new ConditionInfo
      * перечень предоставляемых (или отбираемых) привилегий в виде PermissionInfo [], элементы которого создаются через new PermissionInfo
      * ConditionalPermissionInfo.ALLOW или ConditionalPermissionInfo.DENY – позволяет не только предоставить привилегию, но и отобрать ее.
        + Пример: предоставление bundle возможность импортировать все пакеты, кроме перечисленных. Без этого флага сделать невозможно
        + Становится важным порядок определения условий. Их выполнение производится по порядку пока хотя бы одно не удовлетворится, соответственно DENY условие должно быть определено первым
* Декларативное определение политики безопасности
  + методы чтения / записи политики безопасности в строковом представлении
    - ConditionalPermissionInfo.getEncoded()
    - ConditionalPermissionAdmin.newConditionalPermission(String)
  + Формат строки access { conditions permissions } name
    - Формат conditions - [type "arg0" "arg1" ...]
    - Формат permissions - (type "name" "actions")
  + Пример

**ALLOW {**

**[org.osgi.service.condpermadmin.BundleLocationCondition "file:foo.jar"]**

**(org.osgi.framework.PackagePermission "\*" "IMPORT")**

**} "allow-all-packages"**

* + Основная стратегия: файл, отдельные определения разделены пустыми строками, читаем в List<String>, по одной строке скармливаем ConditionalPermissionAdmin
* Достаточно сложно определить какие именно привилегии надо предоставить какому-либо bundle, поэтому вводится понятие “local permission” – максимальный перечень привилегий, которые используются bundle для работы.
  + Располагаются в файле OSGI-INF/permissions.perm
  + Служит для поддержки инструментальных средств администрирования и аудита
  + Bundle никогда не получает больше возможностей, чем в этом файле. Может получить меньше, должен корректно справляться с этой ситуацией

## Кастомные условия

* Возможность определить собственные условия, отличные от BundleLocationCondition и BundleSignerCondition, например, условия, зависящее от времени, или запрашивающее разрешение на доступ bundle к системе у пользователя
* Классы таких условий должны поставляться из системного classpath, то есть из рутового загрузчика или системного bundle.
  + Лучший способ это сделать – определить класс в bundle, который просоединиться к системному bundle в качестве фрагмента за счет Fragment-Host.
  + У системного bundle есть алиас символического имени «system.bundle», также должен быть опредлен атрибут extension:=framework
* Класс кастомного условия должен содержать
  + Либо статический метод фабрики public static Condition getCondition(Bundle bundle, ConditionInfo info)
  + Либо конструктор public X(Bundle bundle, ConditionInfo info)
* Класс кастомного условия должен реализовывать интерфейс Condition
  + public static Condition TRUE и public static Condition FALSE – если метод фабрики на основании переданных параметров понимает, что условие реально константа, то он может вернуть два прдопределенных условия, которые всегда TRUE / FALSE
  + public boolean isPostponed();
    - признак того, что реальное вычисление откладывается до того момента, когда действительно понадобится значение. Применяется когда вызов занимает много времени или ресурсов (например, опрос пользователя)
    - false – для вычисления используется isSatisfied() прямо в момент проверки условия
    - true – для вычисления используется длинный аналог метода isSatisfied() в конце, когда все не-отложенные условия уже проверены и вычисление данного может изменить ситуацию.
    - Данный метод ВСЕГДА должен возвращать одно и то же значение
    - С точки зрения контейнера НЕ как вычислялось условие, отложенно или непосредственно, это только способ оптимизации
    - Пример
      * Последовательсность условий: отложенное на операцию А, запрещающее операцию А, разрешающая все
      * Если запрошена не операция «А», то условие 1 откладывается, условия 2 не применяется, условие 3 разрешает – результат положительный
      * Если запрошена операция А, то первая откладывается, вторая запрещает, значит для финального вычисления результата надо выполнять отложенную, так как она логически идет ДО запрещающей и может разрешить выполнение
  + public boolean isSatisfied();
    - вычисление условия на месте
  + public boolean isMutable();
    - признак того, что один раз вычисленное значение условия можно / нельзя кэшировать. Если false, то метод isSatisfied будет вызван только один раз
  + public boolean isSatisfied(Condition[] conditions, Dictionary context);
    - отложенное вычисление условия
    - массив conditions всегда содержит только один элемент – само условие, дать совместимости со старыми реализациями
    - единый объект context передается между всеми вычисляемыми отложенными условиями
* Сложный код в кастомном условии должен выполняться из-под AccessController.doPrivileged(), чтобы избежать избыточного контроля. Так как условие в системном bundle у него есть AllPermission, которым надо воспользоваться

## Включение безопасности

* Надо либо заставить контейнер установить менеджер безопасности, либо установить свой
* Контейнерный через стандартные свойства org.osgi.framework.security=osgi и org.osgi.framework.trust.repositories=<путь к хранилищу доверенных сертификатов>
* Свой собственный черевато – какие-то фичи OSGi могут не работать, особенно отложенные условия

# Веб приложения

* Можно выдавать: статический контекст в виде ресурсов, сервлеты / JSP, артефакты других технологий, например, GWT.
* Плюсы разделения приложения на несколько слоев
  + Явно выделяется логическая и физическая структура приложения
  + Управление отдельными веб-приложениями: инсталляция, запуск / остановка, деинсталляция и обновление без помощи сервисов контейнеров
  + Можно перемещать bundle между машиными ез изменения кода и настроек
  + Общие ресурсы, используемые всеми веб-приложениями, развернутыми в одном контейнере, можно вынести в отдельные bundle. Уменьшается объем занимаемой памяти, централизуется управление, убираются дубликаты и так далее
* Ключевой компонент - org.osgi.service.http.HttpService
  + Изначально определен как Compendium. R4.2 enterprise спецификация стандартизирует Web application bundle (WAB) – стандартный War архив с OSGi метаданными, поведение которого расширяется специальными инфраструктурными bundle.
  + Позволяет регистрировать / дерегистрировать вышеперечисленные артефакты
    - HttpContext createDefaultHttpContext();
    - void registerResources(String alias, String name, HttpContext context);
    - void registerServlet(String alias, Servlet servlet, Dictionary initparams, HttpContext context);
    - void unregister(String alias)
  + разработчик не имеет контроля над тем, как именно будет поставляться контент, с какого порта или относительного урла. Можно только частично управлять за счет немногочисленных стандартных свойств
    - org.osgi.service.http.port
    - org.osgi.service.http.port.secure

## Работа со статическими ресурсами

* Регистрируем несколько деревьев ресурсов, подсоединяя их к разным контекстам
  + service.registerResources("/", "/html", null)
  + service.registerResources("/images", "/images", null);
* Последний параметр (выше он null) это интерфейс HttpContext, позволяющий bundle осуществить контроль доступа, кастомную подгрузку ресурса по его имени (по умолчанию грузит из самого bundle) и управление MIME типами для отдаваемых ресурсов
  + boolean handleSecurity(HttpServletRequest req, HttpServletResponse res) throws IOException;
  + URL getResource(String name);
  + String getMimeType(String path);
* Частичная автоматизация
  + Делаем специализированный bundle ResourceTracker
  + Трекер смотрит за поднимающимися / опускающимися bundle, смотрит в их метаданные, считывает пары <контекст>-<путь к ресурсам> и публикует
  + Разнося разные части одного веб приложения по отдельным bundle можно оперативно модульно управлять содержимым, перегружать локализацию и так далее
  + Трекер должен при публикации использовать специальный ProxyHttpContext, который будет перенаправлять запросы не в bundle трекера, а в bundle, источник ресурса
  + Кроме того можно осуществлять управление доступом за счет метаданных bundle, а также маппить локальные файлы bundle в веб сервис

## Работа с сервлетами

* Регистрируем за счет void registerServlet(String alias, Servlet servlet,Dictionary initparams, HttpContext context);
  + Реализация сервлета стандартно строится на основе Servlet
  + Если вместо HttpContext в вызове передаем null, то для сервлета создается уникальный дефолтный ServletContext. А так создается по одному ServletContext для каждого переданного уникального HttpContext
* Частичная автоматизация
  + Различные Bundles регистрируют свои сервлеты как сервисы, относительный контекст сервлета в приложении можно передать как свойство сервиса
  + Инфраструктурый bundle подписан за счет трекера или bind метода на эти сервисы и публикует их за счет HttpService
* Одно из стандартных средств - Pax Web project (<http://wiki.ops4j.org/display/paxweb/Pax+Web>),
  + поставляет WebContainer сервис, расширяющий HttpService
  + дополнительные методы для регистрации JSP, фильтров, слушателей сообщений и так далее

## Работа с WAB

* До спецификации 4.2 не было стандартных способов использования готовых WAR файлов с OSGi контеейнерами
* WAB это WAR, который удовляет как минимум спецификации Servlet 2.5 / JSP 2.1 и содержит дополнительные метаданные в виде заголовков манифеста
* Bnd предоставляет стандартные методы конвертации WAR в WAB
  + Расширение набора классов: Bundle-ClassPath: WEB-INF/lib/gwt-servlet.jar,WEB-INF/classes
  + Определение контекста: Web-ContextPath: /stockwatcher/stockPrices
* Сервлеты могут получать доступ к контейнеру через свой контекст как атрибут со стандартным названием:
  + Пример - getServletContext().getAttribute("osgi-bundlecontext");
  + Дальше уже стандартным образом ко всем сервисам и ресурсам контейнера
* В enterprise приложениях сервлеты стандартно получают ресурсы окружения через JNDI. В данном случае это тоже возможно - [www.osgi.org/Download/Release4V42](http://www.osgi.org/Download/Release4V42)
* можно изготовлять WAB не статически на этапе сборки, а динамически, указывая их как источник для загрузки bundle и конвертируя на ходу
  + делается за счет bundle, публикующих URLStreamHandlerFactory и ContentHandlerFactory сервисы, соуществляющий доступ и загрузку ресурсов
  + для инсталляции указываем путь к WAR как URL со специальным пространством имен. Параметры динамически формируемого WAB указывает тоже в URL. Пример: install webbundle:http://www.acme.com/acme.war?Bundle-SymoblicName=com.example&Web-ContextPath=acme
  + Допустимые параметры URL: Bundle-SymbolicName, Bundle-Version, Bundle-ManifestVersion, Import-Package, Web-ContextPath

# Распределенные сервисы

* Необходимость удаленно зарегистрировать сервис, например, когда удаленный подписчик хочет получать сообщения, путем регистрации своего сервиса с интерфейсом **EventHandler**. Реализация данного сервиса должна вызываться поставщиком сообщений независимо от его расположения
* Достигается за счет OSGi Remote Service. Основной инструмент – провайдеры распределения (импортирующие и экспортирующие), которые создают прокси, представляющие удаленные системы
* OSGi R4.2 спецификация ввела дополнительные к RemoteService сервисы: Remote Services Admin и SCA Configuration Type

## Основные принципы

* Экспорт сервиса, осуществляется service provider
  + Пример:

**Dictionary props = new Hashtable();**

**props.put("service.exported.interfaces", "\*");**

**props.put("service.exported.configs","org.apache.cxf.ws");**

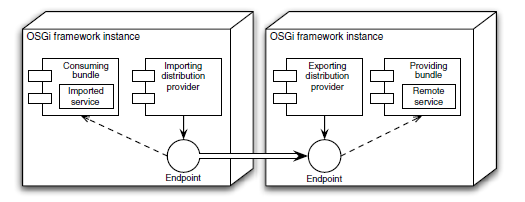
**props.put("org.apache.cxf.ws.address","http://localhost:9090/eventhandler");**

**bundleContext.registerService(EventHandler.class.getName(),subscriber, props);**

* + Через свойства указываем, что
    - экспортируем все зарегистрированные интерфейсы сервиса – «\*»
    - Конфигурационные свойства позволяют провайдеру сервиса передать необходимые параметре distribution провайдеру. В даном случае конфигурацию используем дефолтную для экспорта через Apache CXF (<http://cxf.apache.org/distributed-osgi.html>)
    - реализацию экспортируемого сервиса вешаем на адрес – элемент конфигурации специфичный для выбранного механизма экспорта
    - свойства вместо аннотаций используем, чтобы быть способным работать с Java5
  + Важно, что реализация сервиса НЕ меняется и вообще не в курсе, что его теперь будут юзать удаленно
  + Реальный экспорт осуществляет инфраструктурный distribution provider bundle, его можно выбирать по вкусу и менять, но он должен быть установлен в системе. Например можно использовать Apache CXF (<http://cxf.apache.org/distributed-osgi.html>)
  + Remote Service спецификация определяет, что ключи конфигурации для определенного distribution провайдера должны начинаться с его типа (например, org.apache.cxf.ws). Это не значит, тем не менее, что только Apache CXF distribution провайдер сможет их использовать
  + Провайдер сервиса может указать в наборе свойств конфигурации для нескольких distribution провайдеров, это улучшит его совместимость с различными рабочими конфигурациями контейнеров
  + OSGi R4.2 спецификация определила стандартный набор свойств, которые должны одинаково пониматься всеми провайдерами распределенных сервисов. Наименования свойств должны начинаться с org.osgi.sca, а параметр org.osgi.sca.bindings указывать на наименование провайдера сервиса
* Импорт сервиса
  + Код получения и использования сервиса не меняется, все тот же **context.getServiceReference(<наименование интерфейса сервиса>).** Обращения отправляются к автоматически создаваемому прокси, который ответственен за взаимодействие с сервисом
  + Однако надо учитывать удаленный характер реализации
    - Может неожиданно падать или выдавать непредсказуемые задержки. Соответственно при вызове методов данного интерфейса потребляющей стороной надо уметь обрабатывать **ServiceException**
    - Объекты, получаемые через удаленный интерфейс, являются только копиями объектов, порожденными реализацией (call-by-value vs call-by-reference). Соответственно надо менять семантику интерфейса:
      * вместо set методов на таком объекте делаем конструктор создания нового объекта с новыми свойствами (*шаблон Value Object)*
      * метод update на интерфейсе сервиса, который передаст объект с новыми данными на сторону реализации сервиса
    - Некоторые транспорты, например Corba, поддерживают далеко не все java типы
  + Узнать об удаленном характере сервиса можно посредством свойства **service.imported**.
    - Чтобы запросить у реестра только локальную реализацию сервиса надо включить условие **(!(service.imported=\*)**
    - Конфигурация сервиса, относящаяся к процедуре его импорта, доступна через свойство **service.imported.configs**
    - У импортируемого сервиса НЕТ никаких свойств типа **service.exported.\*.** В отличии от экспортируемых.

## Провайдеры

* Importing / exporting distribution providers реализуют транспорт между bundle, экспортирующим удаленный сервис, и bundle, его потребляющим из локального реестра.
  + Провайдер осуществляет экспорт опубликованного сервиса выбирая и комбинируя
    - Протоколы - SOAP, REST, RMI
    - Безопасность - SSL, DSA, Blowfish, LDAP, Kerberos
    - Трансопрты - HTTP, JMS,Unicast, Multicast, and P2P
  + Конкретный выбор будет осуществлен на основе конфигурации и требований совместимости, налагаемых как потребителем, так и получателем сервиса
  + Спецификация не определяет набор транспортов, могут добавляться еще какие-то виды в зависимости от реализации провайдера
* Провайдеры находятся на обоих взаимодействующих сторонах: на одной стороне экспортирует конечные точки, на другом конце их ищет и создает прокси, маскирующие удаленный характер сервиса



* + Удаленный сервис может экспортировать несколько конечных точек, каждую со своей конфигурацией, клиент может указать какую именно он собирается использовать
  + Так как импортирующий провайдер ответственен за поиск конечных точек, то клиентский bundle сам не обязан указывать адрес реализации, запрашиваемого им сервиса
* Выбор подходящей конечной точки осуществляется совместно обоими сторонами на основании указываемых ими в процессе импорта / экспорта свойств
  + Экспортируемые сервис может указать множественный тип экспорта при регистрации **service.exported.configs = net.rmi, net.soap, net.socket**. Это определит перечень создаваемых конечных точек.
  + При импорте можно
    - выбрать единственный за счет прямого указания требуемого типа в **service.imported.configs**

**ServiceReference ref = context.getServiceReference(EventAdmin.class.getName(),**

**"(&(service.imported.configs=net.soap))");**

* + - указать перечень поддерживаемых типов точек за счет **remote.configs.supported**. Можно запросить, например, все сервисы, поддерживающие конечные SOAP точки

**ServiceReference ref = context.getServiceReference(null,**

**"(&(remote.configs.supported=net.soap))");**

* Импортирующая сторона может осуществлять поиск реализации различными способами – Zeroconf, SSDP, UDDI, Jini, централизованный рестр различной природы, мультикаст TCP / DNS и так далее. Поставщик поставщик distribution провайдера может определить любую собственную стратегию
  + R4.2 спецификация определяет в частности статическую версию поиска реализации за счет XML файл описания, который необходимо указать в заголовке манифеста **Remote-Service**

**<endpoint-descriptions xmlns="http://www.osgi.org/xmlns/rsa/v1.0.0">**

**<endpoint-description>**

**<property name="objectClass" value="org.osgi.service.event.EventAdmin" />**

**<property name="service.imported.configs" value="org.apache.cxf.ws" />**

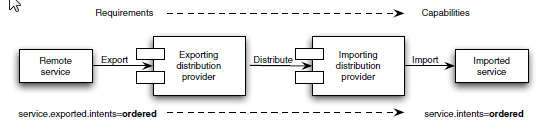
**<property name="org.apache.cxf.ws.address"**

**value=="http://localhost:9090/eventhandler" />**

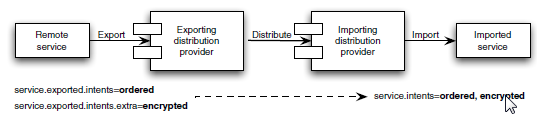
**</endpoint-description>**

**</endpoint-descriptions>**

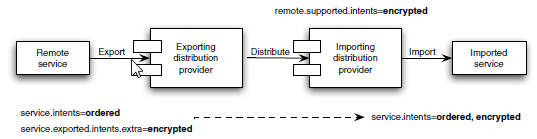
* + Другие возможности зависят от выбранного distribution провайдера, например CXF поддерживает проект Apache Hadoop Zookeeper (http://hadoop.apache.org/zookeeper/),
* экспортирующая и импортирующая стороны договариваются о протоколе вызова, совместимостях и требованиях обеих сторон. Совместимости и требования выдвигаются всеми участниками цепочки: реализация сервиса, экспортирующий провайдер, импортирующий провайдер, потребитель сервиса
  + реализация сервиса определяет свои требования к совместимости как
    - **service.exported.intents** – требования к возможностям, который должны быть в конечном итоге обеспечены при вызове сервиса, без них клиент НЕ будет работать вовсе. Для него не важно кто предоставит эти возможности публикующий distributed провайдер или импортирующий
    - **service.exported.intents.extra** – требования к возможностям, которые не всегда должны, но могут быть обеспечены
  + реализация сервиса может определить УЖЕ реализованные им самим требования как **service.intents.**
    - * Провайдеры не будут заботиться о том, чтобы эти требования удовлетворялись промежуточными компонентами.
      * Значение этого свойства в свою очередь сложится со значением свойства **service.intents** на стороне потребителя
  + intents имеют иерархичную природу, authentication.transport – требование аутентификации как минимум на транспортном уровне.
    - Общая практика, что сервис, требующий такого, может быть экспортирован только провайдером, гарантирующим authentication.transport или authentication. Однако подобное требование сервиса может быть удовлетворено НЕ только подобным провайдером
    - intents могут быть самыми различными, перечень в [www.osoa.org](http://www.osoa.org). В общем случае перечень тербуемых совместимостей должен быть как можно короче, чтобы обеспечить публикущему / импортирующему провайдерам максимум гибкости
  + экспортирующий провайдер использует значения свойства **service.exported.intents,** чтобы убедиться, что он способен правильно экспортировать данный сервис.
    - Он должен удовлетворить требование, но может добавить любое поведение, разумное по его мнению
    - импортирующий провайдер также выполняет проверку совместимости и проносит для потребителя данное свойство как УЖЕ удовлетворенные совместимости в виде **service.intents**



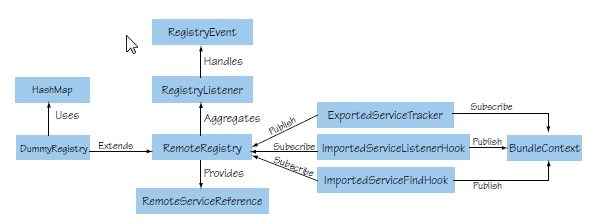
* + потребитель сервиса получает опубликованные провайдером сервиса свойства **service.intents** и удовлетворенные distribution провайдером свойства **service.exported.intents** и **service.exported.intents.extra** сложенными вместе в виде **service.intents,** что в свою очередь позволяет ему отобрать реализации сервиса, которые он способен использовать



* + - Чтобы обеспечить возможность наложения фильтра в LDAP синтаксисе на удовлетворенные значения совместимостей, каждое иерархическое значение при этом расширяется вверх: service.intents=confidentiality.message превращается на стороне клиента в service.intents=confidentiality,confidentiality.message. Так как ему не важно как именно поддерживается конфиденциальность
    - Импортирующий провайдер также может определить свои требования в **remote.supported.intents,** которые будут служить фильтром и попадут в результирующие свойства сервиса



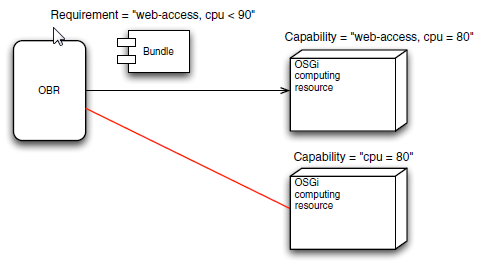
* Внедрение собственного механизма поиска реализаций сервисов
  + Модель классов



* + RemoteRegistry – интерфейс реестра удаленных сервисов, устроенный наподобие реестра сервисов OSGi
  + RegistryListener – получает уведомления о сервисах пбликуемых / удаляемых из реестра удаленных сервисов
  + ExportedServiceTracker – получает уведомления о bundle, которые заявляют сервисы, которые публикуются как внешние. Проверяет свойства таких сервисов, могут ли быть они удовлетворены и публикует их в RemoteRegistry
  + За счет ListenerHook и FindHook реестр удаленных сервисов может заметить, что кто-то ищет сервисы, которые в нем зарегистрированы, и подсунуть вместо отсутствующих локальных реализаций, прокси, которые смотрят на удаленные реализации

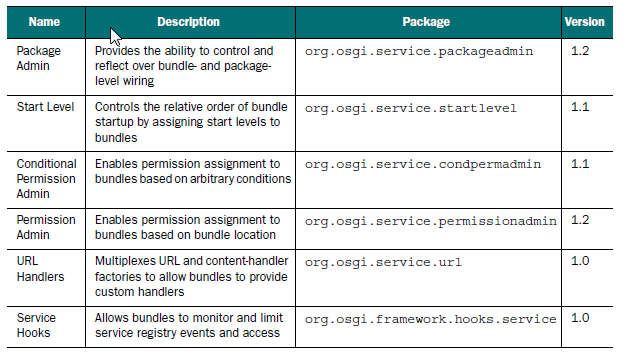
## Облако

* Чтобы эффективно воспользоваться услугами облака приложения должно иметь соответствующую модульную распределенную архитектуру
  + Должны выделяться отдельные специализированные компоненты
  + Компоненты должны публиковать свои требования к узлам облака, которые могут их исполнять
  + Должны поддерживаться удаленные сервисы
  + Клиенты сервисов должны учитывать их динамическую природу
* Выгодность разделения приложения на отдельные компоненты, размещаемые по разным узлам облака:
  + Узлы становится проще в смысле управления. Если два веба и четыре аппа, то веб-сервер к примеру надо обновлять и конфигурировать только на двух машина. Если все узлы идентичны – на всех шести
  + Узлы можно делать с разными характеристиками, более заточенными под задачи в смысле требований к железу и базовому ПО. Таким образом, они становятся дешевле.
  + Возможность выделения специальных машин под отдельные высоко требовательные задачи
  + При обновлении приложения можно обновлять (а значит тестить ПО и пакетировать для узла) только некоторые узлы, меньше объем и сложность изготовления обновления
  + Можно использовать правила динамического расширения / сужения облака, например, если загрузка веба больше>80%, то поднять новый узел с вебом, если <20% опустить узел
* Основные недостатки облака – безопасность и задержки
* OSGi как основа для облака
  + Позволяет обновлять систему с минимальной гранулярностью без необходимости генерировать образы машин или JVM
  + Архитектура приложения на основе OSGi идеально подходит для размещения в облаке (модульность, абстрактный транспорт, декларативно определяемые требования, управления зависимостями). А разрабатывать можно на столе
  + Принципы устройства:
    - Единый для облака депозитарий bundles – OBR
    - Автор приложения помещает свои bundle в OBR, а оттуда они расползаются по узлам в соответствии с декларативно определенными
      * Требованиями к узлам (железу, базовому ПО, предоставляемым сервисам) описанными в bundle: **Require-Capability**: com.cloudprovider; **filter**:="(&(web-access)(cpu<90))"
      * Описаниями узлов: **org.osgi.framework.system.capabilities.extra**= "com.cloudprovider; web-access:Boolean=true; cpu:Long=80"

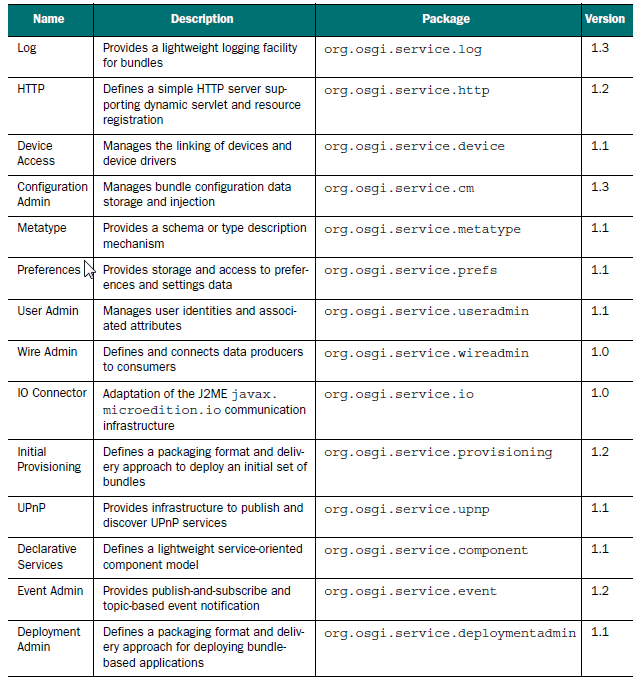


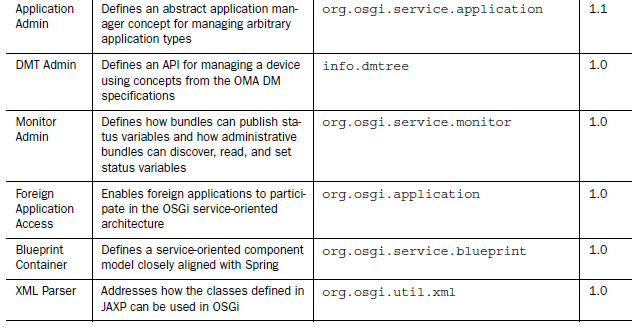
# Сводка OSGi сервисов

## Core сервисы



## Compendium сервисы





## Enterprise сервисы

