Службы провайдеров

Проблема

```
class CheatSheet {
  public void getAnswer(String question) {
    Map source =
    String answer = source.get(question);
    if (answer != null) return answer;
    else return "Epic fail";
  }
}
```

Бывает так, что несколько классов выполняют идентичные функции, но реализация у них разная, при этом могут появиться новые реализации, поэтому логично создавать их в виде внешнего модуля.

Представим, что нужно написать приложение для подготовки к экзамену — что-то типа электронной шпаргалки Создадим класс с методом получения ответа getAnswer(), который ищет ответ на заданный вопрос. Если ответ найден, то возвращаем его, если не найден, то делать нечего — экзамен провален.

Осталось только понять, где мы можем искать ответы.

Проблема

```
class CheatSheet {
  public void getAnswer(String question) {
    Map source = knowledge.getSource();
    String answer = source.get(question);
    if (answer != null) return answer;
    else return "Epic fail";
  }
}
interface Knowledge {
  Map<String,String> getSource();
}
```

Все просто — нам нужен какой-то источник знаний. Чтобы не привязываться к конкретному источнику, а сделать программу универсальной, определяем интерфейс Knowledge с методом getSource(). Далее где-то получаем объект этого интерфейса, который сможет предоставить нам источник ответов.

Проблема

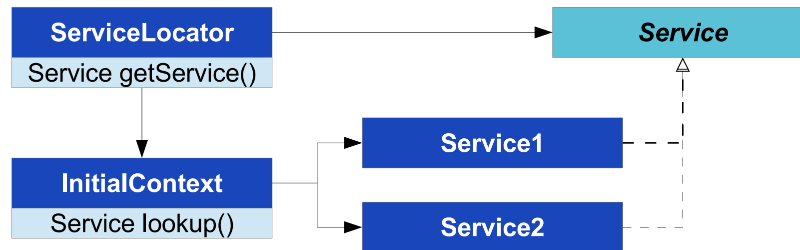
```
class CheatSheet {
  public void getAnswer(String question) {
    Map source = knowledge.getSource();
    String answer = knowledge.get(question);
    if (answer != null) return answer;
    else return "Epic fail";
  }
}
interface Knowledge {
  Map<String,String> getSource();
}
class Magic implements Knowledge
class CallFriend implements Knowledge
class Lectures implements Knowledge
class Memory implements Knowledge
```

Источник знаний может быть любым. Можно воспользоваться магическим знанием, реализовав класс Magic. Можно позвонить другу, который уже сдал экзамен. Можно написать класс, который будет вытаскивать ответы из лекций. И, наконец, можно использовать свою собственную память и запомнить ответы.

Осталось решить одну проблему — надо как-то универсально передать объект знания в наш класс, при этом сохраняя независимость от реализации.

Как передать объект?

- Шаблон ServiceLocator



В этом случае может помочь шаблон ServiceLocator. Он подразумевает, что есть некий интерфейс, представляющий сервис — в нашем случае это Knowledge. У него есть несколько реализаций, нам нужно получать объект сервиса универсальным способом.

Для этого служит класс ServiceLocator, который пользуется классом InitialContext для поиска конкретных классов, реализующих сервис. ServiceLocator хранит у себя ссылки на найденные объекты и предоставляет их при вызове метода getService().

Реализация в Java

- Класс java.util.ServiceLoader<Service>
  - static ServiceLoader<Service> load(Service.class)
  - Iterator<Service> iterator()

В Java данную схему реализует класс ServiceLoader. Класс имеет метод load(), который позволяет загрузить конкретную реализацию сервиса, и метод iterator(), возвращающий итератор по загруженным службам.

Можно пройтись по службам в цикле, и выбрать нужную, которая подходит в данный момент.

ИТМО ВТ

Реализация в Java

- interface spi.Service
  - execute()
- class spi.DefaultServiceImpl
  - public DefaultServiceImpl()
- service.jar
  - META-INF/services/
    - spi.Service
      - spi.DefaultServiceImpl

Для того, чтобы внешний подключаемый модуль успешно был обнаружен ServiceLoader-ом, необходимо в jar-файл добавить следующие компоненты:

1) Интерфейс Service, представляющий сервис с методом, позволяющим выполнить действие. Например, execute.
2) Один или несколько классов, которые реализуют данный интерфейс, например, DefaultServiceImpl. Классы должны иметь конструктор без параметров.
3) В каталоге META-INF/services должен быть файл с именем, совпадающим с именем интерфейса сервиса, содержащий список классов, реализующих интерфейс Service.

Провайдеры

- java.nio.file.spi.FileSystemProvider
- java.nio.channels.spi.AsynchronousChannelProvider
- java.nio.channels.spi.SelectorProvider
- java.nio.charset.spi.CharsetProvider
- java.text.spi.DateTimeFormatProvider
- java.text.spi.NumberFormatProvider
- java.util.spi.CalendarDataProvider
- java.sql.DriverManager

По этому принципу реализованы провайдеры служб в стандартной библиотеке Java, в том числе FileSystemProvider, AsynchronousChannelProvider, SelectorProvider и другие.

В JDBC локатором служб является DriverManager, а модули с драйверами обычно распространяются в виде jar-файлов с изученной структурой.