# INB/INN374 Assignment 2

# Service-Oriented Architecture for Automotive Maintenance Activities

**Due:** Friday 24 October 2014 (Week 13)

**Teams:** 1-4 members

**Worth:** 30%

## Task

Design and implement a collection of inter-connected software services and front-end application(s) that simulates the operations of an automotive maintenance company.

## Scenario

*Servicing Our Autos* is a company that provides various automotive maintenance activities for its customers. Within *Servicing Our Autos*, the *sales* department is the entity responsible for ordering maintenance activities. The sales department interacts from time to time with these five entities:

- *The customer:* A customer is any person interested in placing an order for maintenance of their vehicle. Sales interact with customers for the purpose of scheduling a maintenance activity (also called a 'work order').
- *The warehouse:* The sales department may interact with the warehouse to check for the availability of parts that are needed for a work order.
- *The supplier:* For parts that are not immediately available from the warehouse, sales may interact with a supplier to order parts that are required in a work order.
- *The garage:* Members of sales may interact with the garage to book a service bay and a mechanic to perform the required work.
- *The insurance department:* a customer's vehicle can be insured. For the purpose of scheduling a work order for an insured car, the sales department interacts with the insurance department.

Sales, warehouse and insurance are all departments within *Servicing Our Autos*. The garage is also part of *Servicing Our Autos*.

### *Business Process*

The business process starts with the receipt of a request for work from a customer. It finishes when either the sales department schedules an appointment with the customer or the customer rejects the quote for work.

Upon the receipt of a request for work, the sales department estimates the expected cost of the required supplies, parts and labour, and prepares a quote with the estimated total cost for the maintenance activity. If the customer's vehicle is insured, the sales department will then interact with the insurance department to retrieve the details of the customer's insurance plan. Depending on

the plan, the customer's insurance may cover the full maintenance costs, or a partial contribution may be required from the customer.

The sales department then presents the quote to the customer, who can either accept or reject the quote by notifying the sales department. If the customer accepts the quote, the sales department contacts the warehouse to check if the required parts are in stock. If some parts are not in stock, the sales department interacts with the supplier to provide an estimate for when the parts will be available, and informs the customer of the delay. The customer may agree to wait the required time for parts or reject the quote at this point. If the customer has decided to wait for the parts, sales will place an order with the supplier for the parts.

Once all required parts are in stock or have been ordered, the sales department schedules an appointment for the customer to bring their vehicle in and interacts with the garage to book a suitably-equipped service bay and a suitably-qualified mechanic to perform the work. Finally, a confirmation of the appointment is sent to the customer directly from the garage.

It is not within the scope of this business process to deal with the provision of the maintenance work, such as performing the planned activities and those ones that become apparent when the vehicle is inspected.

As is usually the case, this scenario description may be incomplete or contain inconsistencies. You are encouraged to seek clarification from lecturer and/or tutors. If necessary, an FAQ will be put together from questions asked and will be posted on Blackboard for the benefit of all interested students.

# Assignment Requirements and Infrastructure

## *Software Services*

An SOA system is made up of one or more software services that support one or more front-end client applications. You are required to implement a number of software services that fulfil the functional needs of the entities listed in the scenario. These software services should be exposed as web services, using the WS-* and/or RESTful architectural styles. Each software service must be implemented in either Java or C#, and each language must be used at least once (that is, your assignment must implement some service(s) using Java, and some service(s) using C#). For C#, it is recommended that the WCF API is used (see Tutorial 6), and for Java the JAX-WS API is recommended (see Tutorial 5).

## *Front-end Client Applications*

You are also required to implement a front-end client application to access your web services, simulating an in-store portal. The front-end client application may be implemented as a web-based application (hosted on a web server and accessed via a browser) or as a rich client application (run from the desktop). You may implement either a .NET-based or a Java-based solution, and may use any IDE you have access to.

It is expected that your solution will display all information in a visually appropriate format. System users may be computer novices, so the application should be easy to use and reveal an understanding of the user perspective. Higher degrees of usability and overall professionalism will be expected of larger teams.

## *Heterogeneity*

Two of the important principles to showcase are service heterogeneity, i.e. variety in languages, frameworks and platforms, and that your services are physically distributed. Your system should be deployed over at least two machines, ideally with different operating systems. Two servers have been made available to you for this assignment: \\fastapps04.qut.edu.au (Windows with Microsoft

IIS and MS SQLServer) and \\fastws.qut.edu.au (Linux with Oracle Glassfish and MySQL). You may use these machines to deploy your services and web applications. If you wish to access these servers from outside the QUT network, you can do so but you will need to use the QUT VPN. Due to QUT's security policy, these servers cannot access any service or web application that is external to the QUT network. If you want one of your services/applications to access a service situated externally, the only possible solution is for you is to deploy *all* of your services and your front-end application to servers external to QUT. This is a valid workaround, provided the servers you use are accessible from the student labs.

During development, you will most likely need to update your services/web applications a number of times. Please first deploy them locally to your development machine and test them thoroughly. Once you are satisfied that they are working correctly, you may then deploy them to the QUT servers. If deploying to the Glassfish server, do not forget to create your server descriptor file (see Tutorial 5) and to tick the *Verifier* option. Moreover, please catch all exceptions within your service/application implementation as close to their source as possible, rather than escalating them to parent classes. If you fail to do so, these exceptions may result in a crash of the entire server. Finally, please follow the naming conventions prescribed in the various tutorials: *always* use your student number as the prefix of your services/web applications' name. Any service/application that does not follow these conventions may be removed immediately and omitted from assessment.

Deploying your web applications and services to "localhost" will not meet the assignment's objectives, although you will still obtain part marks if you deploy all the services and front-end applications in a single machine (see the attached "criteria sheet"). Any rich clients you create need to be able to be run from the student labs and be able to connect to all the services that they need to use.

## Simple Services

This assignment is primarily about enterprise software architecture – not coding. So, don't waste a lot of time coding any individual service or client, or adding any functionality not explicitly required by the scenario. Each service and client should be relatively simple. It is the overall architecture that will be assessed, not the functionality of individual components. For assessment purposes, we will be treating each service and client application as a black box. We are looking for interesting service-oriented architectures, not complex individual services or applications.

## Business-Orientation

You should aim to make your service interfaces and your overall service-oriented design interesting, but most importantly it must be a sound and appropriate design for the scenario you face. The principles of service-oriented architecture dictate that each service should be business-oriented in both scope and function and the operations should be of appropriate granularity. When designing your system, you should consider various ways of decomposing your system into services, and choose the design that best meets the above criteria as well as best supporting reuse that you can reasonably be expected to foresee.

## Service Layers

With reference to a logically layered architecture, larger teams that implement the basic service layer only will do no better than a bare pass. Larger teams aiming for higher grades may wish to implement intermediary and process-oriented services. All teams should however aim to implement both logic-centric services, i.e. services that encapsulate some logic, and data-centric services, i.e. services that encapsulate an underlying persistent data repository such as a relational database. For an example on how to build data-centric services in .NET see Tutorial 7, or for Java see Tutorial 8.

It is intended that teams create their own datasets. For example, you might create a small number of simple customer records, and/or some simple warehouse records. Only a handful of records is

required – enough to demonstrate your services. Also, you are allowed to use someone else's dataset, provided that you acknowledge that in your report. Small teams (1-2 members) should aim for at least one data-centric service, while large teams (3-4 members) should aim for at least two, on at least two different platform types. The remaining services may use and/or return randomised, non-persistent data.

### *Documentation*

You will need to thoroughly document your software services (from a black-box perspective). The documentation for a given service should provide a potential client with all the information they would need to decide if the service was suitable for their needs and to be able to use it (they should not need to look at your source code or talk to you in order to be able to use your services). This documentation should be made available as a Service Repository.

The Service Repository should be browsable via a web interface. A set of HTML pages is expected: one page per service that you have implemented. On each page, include the name of the service, a textual description of the purpose of the service, and of its functionality (briefly describe what each operation does), and any constraints on usage (e.g. service protocols and description of preconditions for each operation). Also, include a URL to the service itself and a URL to the deployed WSDL for that service.

To link all the HTML pages together, you can prepare a separate web page with your name(s), your student number(s), and links to the HTML pages corresponding to each of your web services.

You can deploy your Service Repository on one of the unit's servers, or you can use a server of your choice (provided it is accessible from the student labs). Alternatively, you can simply provide a collection of HTML pages within the zip file you will submit with your assignment.

In addition to documenting each service individually, you will need to document the design and deployment of the entire system through the following diagrams/models:

- Logically Layered Architecture Diagram (see Lecture 7), where you show the structure of your system being made of services and front-end application(s), and for each service you specify its type (e.g. data-centric, process-centric, etc.)
- Interaction Diagram (see Lecture 8),
- Choreography Model (see Lecture 10),
- Orchestration Model showing the internal behaviour of any process-centric service that you may have implemented (see Lecture 10),
- Deployment Diagram (aka Physical Diagram – see Lecture 10), where you illustrate the physical machines (including URLs and operating systems) your services and applications have been deployed to.

These diagrams/models should be included in the report (see below), not in the Service Repository.

## What to Submit

Assignments must be submitted via Blackboard, using the Assignment 2 link on the unit's *Assessment* page. You must submit a single zip file named "Assignment2.zip", containing the following items:

- A report, in PDF or Word format, containing exactly the following sections, each on a new page:
  1. A title page including the full names and student numbers of all team members [1 page max] – a table of contents is not required.
  2. A brief textual description of your system – scope and basic functionally, don't go into structure yet. [1 page max].

3. A Links page containing the URL(s) for your web-based Service Repository (if you have deployed it to a server), and any front-end web application(s). Also the file names (and usage) for any rich client applications. [1 page max]
4. A logical layered architecture, an interaction diagram and a choreography model. If you have implemented any process-centric service, you also need to submit an orchestration model [5 pages max].
5. A brief discussion of the SOA analysis and design methodology you chose to follow (e.g. bottom-up or top-down): the steps you followed and the reasons why you chose this methodology. Also, you should indicate whether you considered different methodologies and why these were deemed not to be suitable for the type of systems you planned to build. Finally, you need to specify which assumptions (e.g. simplifications) you took into account when designing and implementing your system. These may be justified with reference to the number of team members. [1 page max].
6. Bullet points listing the aspects of service orientation that you have showcased. Describe which types of services you implemented (e.g. process-centric, basic, etc.) and why you chose each of these services to be implemented so. Moreover, you have to showcase which SOA principles (e.g. loose-coupling, reusability – see Lecture 7) your services fulfil and why. For example, you have to show that your service interfaces were built in a way that is general enough to be reused outside your SOA. [1 page max]
7. A deployment diagram [1 page max].
8. A table listing for each of your services and applications, the platforms, languages and APIs/technologies used to implement them, a measure of their complexity (e.g. lines of code) and their author(s). Please note that this table will be used to check whether or not there was an equal distribution of workload among your team members. [1 page max]
9. If there is any other documentation that you feel would support your submission, please discuss it with the unit coordinator. No other sections should be included without permission [12 pages for the entire report maximum].

- Any rich application that you have created, in executable form (e.g. exe or jar file). If you have not developed any rich applications (i.e. your front-end application is web-based), then you do not need to include this item.
- The HTML files of your Service Repository, only if the repository hasn't been deployed to a web server.

You are expected to submit a working system. It is vastly better to submit a working system that meets some of the requirements, rather than a non-working attempt at the entire system. The best approach is to start small – develop a core system that implements some basic requirement, then build on it. Using this modular approach, at any stage you will always have a working system.