Outline of QuoCo Insurance System Scenario

Introduction

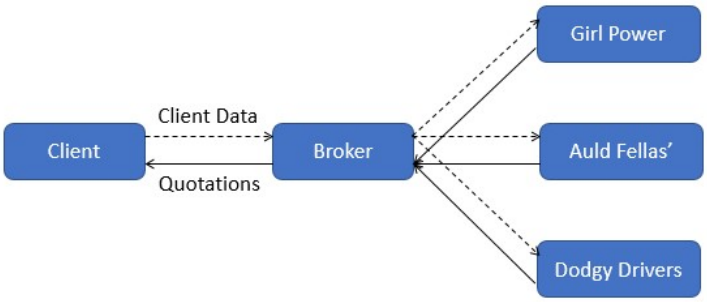
You have recently started working for an insurance broker and have joined the team responsible for developing a multi-device insurance quotation portal. This project has only just started, and after taking a distributed systems programming course at college, you are now considered the "network programming" expert.

The lead developer for the team has mocked up an outline of what they are looking for, and you have been tasked with identifying the best technology for distributing the app. Each week you are expected to deliver a prototype based on one potential technology.

Overview of the System

The purpose of the system is to allow heterogenous clients to connect to a Broker Service that will take **client info** as an input and which will produce a **list of quotations** as the output. The quotations will be generated by interacting with the Web APIs provided by the various insurance companies that the broker works with. Because all these APIs may be different, the decision has been made to wrap each external API in an internal service that will provide a standardised interface to the system you are building.

To allow you to prototype an implementation, the lead developer has provided you with 3 dummy insurance service implementations: Girl Power, Dodgy Drivers, and Auld Fellas'.



Detailed Design

The lead developer wants to adopt a service-oriented approach (where each function – the insurance quotations & the broker – is viewed as a service). They also want the system to be able to automatically handle the addition/removal of insurance quotation services (as the companies they work with change). The accepted approach to implementing such a system is to use some form of registry (although this may not always be the case), so their basic prototype has been designed to include a simple registry.

In the code provided, the following packages are specified:

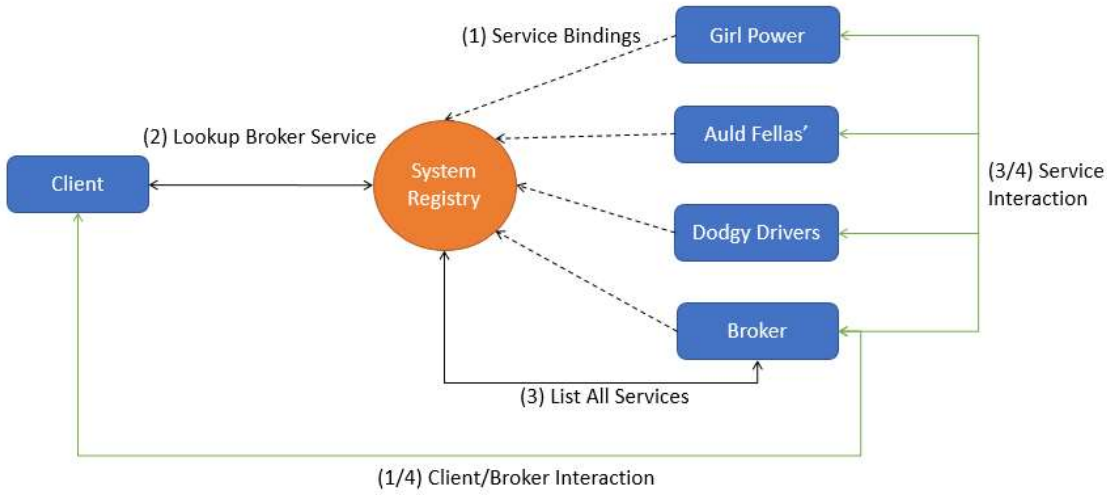
- **service.registry**: This package contains a basic service registry implementation. This consists of two classes: the registry itself (which uses static methods to implement a singleton); and an interface that objects wishing to be registered via the registry must implement.
- **service.core**: This package contains the core components of the system. This includes: interfaces for the Quotation and Broker services; the implementation of an abstract quotation

service that contains some common methods that is used by all concrete quotation services; and two data classes that represent the client info and the quotations.

- **service.broker**: This package contains the implementation of the broker service. This service basically looks up all available quotation services from the registry (notice that quotation services are prefixed by the string "qs-"), and requests a quotation from each service found. The list of retrieved quotations is then returned to the invoker of the service.
- **service.auldfellas**, **service.dodgydrivers**, **service.girlpower**: Packages that implement the dummy insurance quotation services. Each service implementation extends the AbstractQuotationService class, which implements the QuotationService interface (which itself extends the Service interface).
- **client**: The package that contains the test client implementation, which connects to the brokers and sends dummy client info data to the broker to request quotations.

The figure below gives a high-level view of the behaviour of the insurance quotation system:

1) The four main services bind to the System Registry, advertising their existence.
2) The client looks up the broker via the registry and starts submitting requests for quotations.
3) For each quotation request, the broker retrieves a list of all active services, using the "qs-" prefix to identify which services are Quotation Services, and requests a quotation from each active quotation service.
4) The quotation services return their quotations to the broker and, once all services have responded, the broker returns a list of quotations to the client.



What to do first?

Read this document and run the code. Try to become familiar with each part of the code. If useful, add some console output to the code so that you can follow the flow. You should not start trying to complete the practical work before you are comfortable with the system.

- Keep a copy of the original form of the system for reference.
- If something isn't clear – email me.
- You may need to adapt some of the classes / interfaces for specific approaches. Try to keep a (change) log of the changes you have made. Ideally, I would like it to be written in a text file (called CHANGES.TXT) that is in the project root folder. Only brief notes are required – enough to understand what changes you have made.