Fachhochschule Vorarlberg

Date 17.03.2022

ITB
Enterprise Computing
Thomas Feilhauer

Exercise Assignment 2

2: Java RMI based Automated Teller Machine

Implement a distributed Automated Teller Machine (ATM) using Java RMI.

Functional Requirements:

The ATM will simulate a real world automated teller machine.
The ATM must support a variety of accounts.
The ATM must support the following operations:

- **deposit**: add some Euro amount to a specified account's balance
- **withdraw**: deduct some Euro amount from a specified account's balance
- **balance inquiry**: get current balance of a specified account

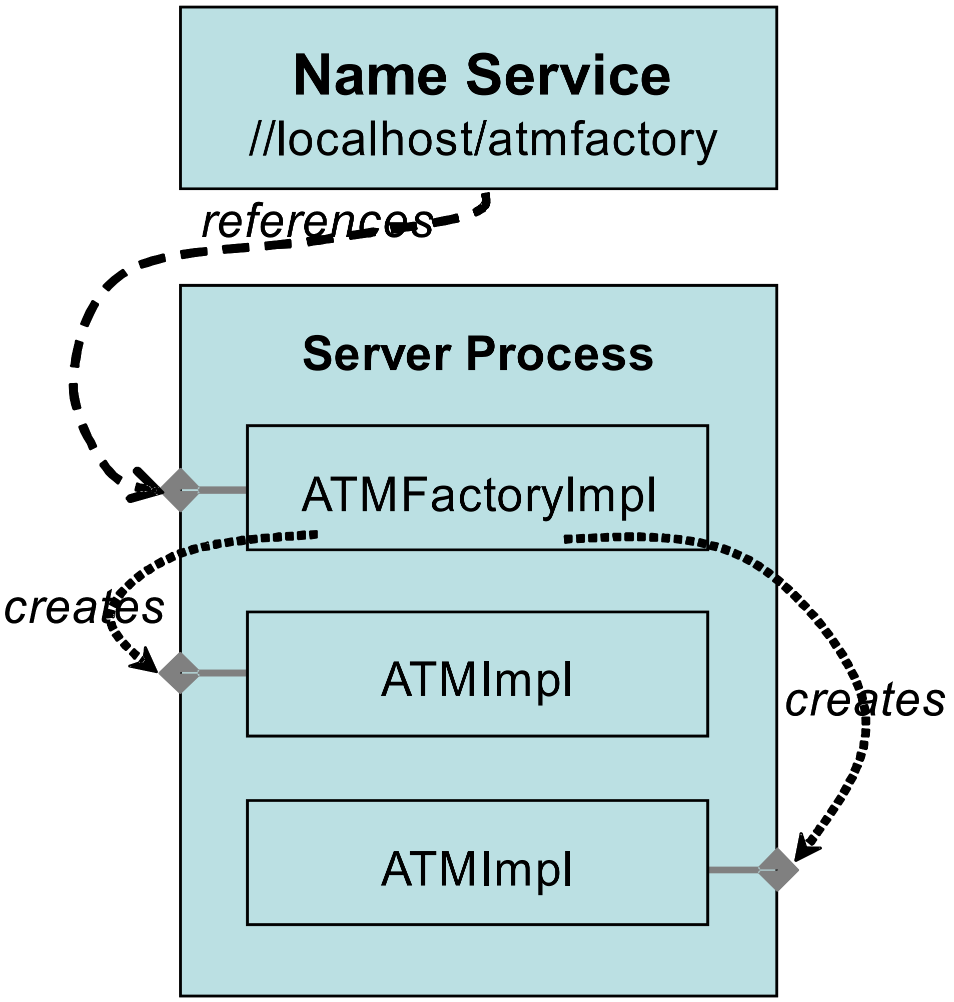
The ATM will run in its own process and will handle remote requests from a client running in some other process on a different machine.

Design

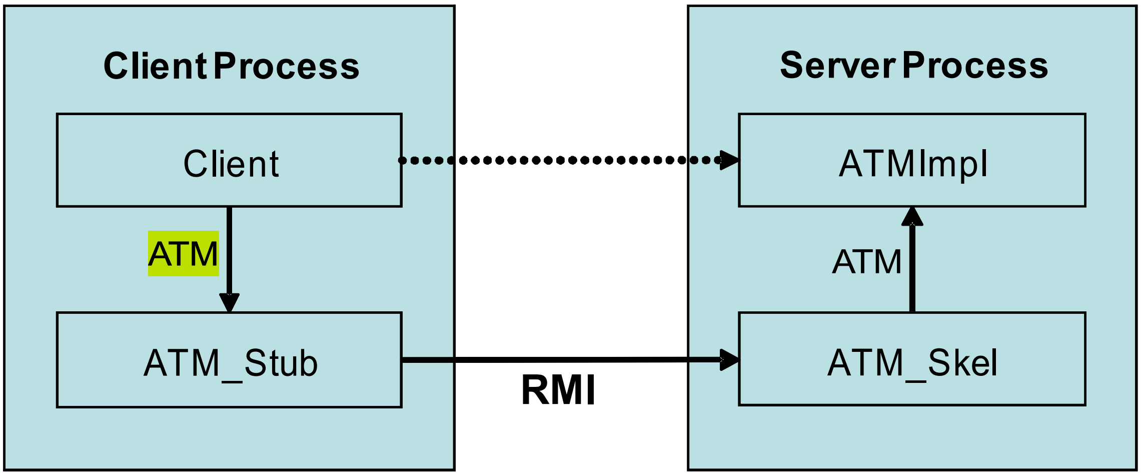
You will define an ATM interface that will be implemented by both the real ATM and a client side stub for the ATM.
The ATM will include several individual **`Account`**s and each of the ATM methods must include a parameter that allows the account number to be specified. For simplicity we'll assume the account identifier type is **`int`**.
One of the challenges of distributed computing is connecting a client to the *first* remote object. Once the first object reference is obtained it can be used to gain access to other remote components. In fact it might be a good idea if the entire job of this first object we connect to should be providing references to other objects. This design is so common it has been detailed as a common *design pattern* called *factory*. A factory is a remote object whose main job is to provide references to other remote objects, i.e. it is a navigational starting point to find our way around objects on the server side. In this system, you will create an ATM factory that the client will use to get a reference to a remote ATM.
The ATM factory is a server side object with a remote interface just like the ATM. You will create an **`ATMFactory`** interface that has a single method **`createATM()`** that returns a remote reference to the serverside **`ATM`** instance.
The server process will start up, create an **`ATMFactoryImpl`** instance, and then register it with the naming service. The client will then be able to lookup the ATM factory and connect to it. Then the client will use the **`createATM()`** method of the factory to get a remote reference to an **`ATM`** instance.

Architecture

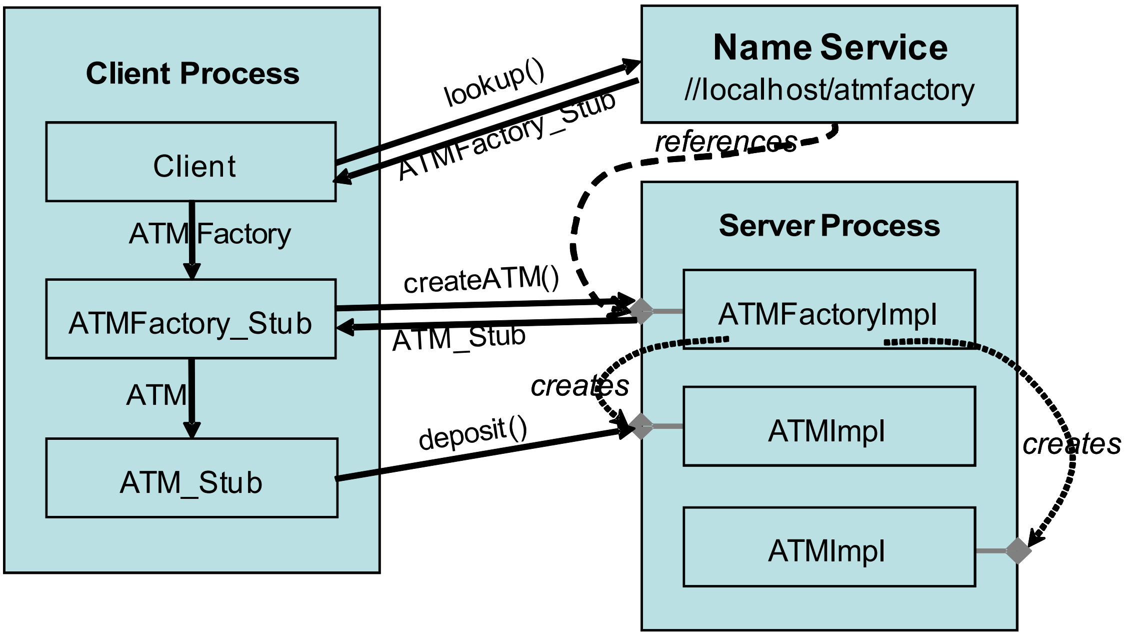
The following diagram illustrates how a factory can be used in bootstrapping a remote application. In the diagram only two **`ATM`** instances are shown, but imagine a real banking system with hundreds of **`ATM`**s. The factory is aware of the remote locations of each **`ATM`** instance and can provide any one reference to a client application. Without the factory, all of the **`ATM`** instances would need to be registered in the naming server and the client would need to know a unique name for each **`ATM`** instance. In this case a factory is extremely useful.



And the following diagram shows how the stubs and skeletons are used to dispatch method calls to remote implementation objects.



The following diagram shows the entire flow from the client's perspective. First the client looks up the **`ATMFactory`** in the name service. Once it has the factory it can call **`createATM()`** to get a reference to an actual **`ATM`**. And once it has a reference to an **`ATM`** it can call methods on it.



Account

Develop an **`Account`** class that represents an individual account. The **`Account`** should contain account specific data like a balance and methods for manipulating that data. The ATM implementation will create several **`Account`** instances to be manipulated by the client.

ATM-Interface

Write an ATM-Interface that corresponds to the following method signatures:

```
      public void deposit(int accountNo, float amount);
      public void withdraw(int accountNo, float amount);
      public float getBalance(int accountNo);
```

ATMImpl

Create a servant **`ATMImpl`** that implements the **`ATM`** Interface.
When the **`ATMImpl`** is created it should reference the accounts created by the server at startup.
When a remote request comes in, the banking transaction should be carried out on the appropriate account.

ATMFactory-Interface

Create an **`ATMFactory`** Interface that supports a single **`createATM()`** method.

ATMFactoryImpl

Create the implementation of the ATMFactory-Interface. It should return a remote reference to an **`ATM`** instance.

Server

Create a **`Server`** class. The **`Server`** is used to startup the server application. Since you've created a convenient factory object, the central task of the **`Server`** is to create an instance of the factory and bind it to the naming service. After that is has to create some accounts and store them in a collection. At the very minimum create accounts with the following initial values:

Account Number	Initial Balance
1	€ 0
2	€ 100
3	€ 500

Client

Create a client class that includes the following lines to test your application:

```
// get initial account balance
System.out.println("Initial Balances");
System.out.println("Balance(1): "+atm.getBalance(1));
System.out.println("Balance(2): "+atm.getBalance(2));
System.out.println("Balance(3): "+atm.getBalance(3));
System.out.println();
// make €1000 depoist in account 1 and get new balance
System.out.println("Depositting(1): 1000 ");
atm.deposit(1, 1000);
System.out.println("Balance(1): "+atm.getBalance(1));
// make €100 withdrawal from account 2 and get new balance
System.out.println("Withdrawing(2): 100 ");
atm.withdraw(2, 100);
System.out.println("Balance(2): "+atm.getBalance(2));
// make €500 deposit in account 3 and get new balance
System.out.println("Depositting(3): 500 ");
atm.deposit(3, 500);
System.out.println("Balance(3): "+atm.getBalance(3));
// get final account balance
System.out.println();
System.out.println("Final Balances");
System.out.println("Balance(1): "+atm.getBalance(1));
System.out.println("Balance(2): "+atm.getBalance(2));
System.out.println("Balance(3): "+atm.getBalance(3));
```

Which classes should be RMI classes?
Develop server and client applications as separate projects! The interfaces should also be provided in a separate project and should then as a JAR-file be integrated into the client and the server projects. Which files must be located in the client application, which in the server application?
Start the RMI registry in a separate process!
Test your application by running client and server on different machines!

You should see the following output from the Client:

```
      Initial Balances
      Balance(1): 0.0
      Balance(2): 100.0
      Balance(3): 500.0

      Depositting(1): 1000
      Balance(1): 1000.0
      Withdrawing(2): 100
      Balance(2): 0.0
      Depositting(3): 500
      Balance(3): 1000.0

      Final Balances
      Balance(1): 1000.0
      Balance(2): 0.0
      Balance(3): 1000.0
```