In diesem Kursteil

Jini - Erklärungen zum Start

1.1. Kursübersicht

Nun möchte ich Ihnen auch noch einige Hintergrundinformationen zum Startbeispiel (HelloWorld) geben. Dabei geht es nicht darum, Ihnen zu erklären, wie Sie die Umgebung aufsetzen müssen. Das finden Sie im Start-Kapitel.

Das Beispiel verwendet weder JavaSpace noch den Transaktions-Manager. Es werden lediglich http für den Transport von Archiven und Klassen, der RMI Daemon, der Jini Lookup Dienst (Reggie) sowie der Service selbst, plus dem Client eingesetzt.

Dabei muss der Aktivierungs-Daemon auf demselben Rechner laufen, wie der Lookup-Dienst, weil die Sun-Implementierung des Lookup-Dienstes (RMI-)Aktivierung verwendet.

Welches sind kritische Grössen in diesem einführenden Beispiel?

- PATH und CLASSPATH müssen entsprechend der SetEnvironment.bat Datei eingerichtet werden.
- Der RMI-Daemon muss vor dem Lookup-Dienst und auf dem selben Rechner gestartet werden.
- Der Lookup-Dienst verwendet die Multicasting-Discovery-Protokolle. Jini sucht also zuerst in seiner Nähe und benötigt einen gültigen TCP/IP Stack.
- Der Jini-HTTP-Server liefert herunterladbaren Code für den Jini-Lookup-Dienst. Als Root wird das Lib-Verzeichnis von Jini gewählt. Damit wird sichergestellt, dass Programm Code aus der Datei reggie-dl.jar herunterladen kann, der zur Verwendung des Lookup-Dienstes benötigt wird.
- Ausserdem muss ein HTTP-Dienst eingerichtet werden, um Code zwischen Clients und Services auszutauschen.

1.2. Das Erstellen von Verteilten Anwendungen

In realen Applikationen werden die Jini-Dienste verteilt auf mehreren Rechnern laufen, schon aus Zuverlässigkeits-Gründen.

Damit ist auch klar, dass der Code von unterschiedlichsten Rechnern stammen kann.

In der Regel werden Sie eine Jini Applikation auf einem einzelnen Rechner entwickeln. Das führt aber zu einigen typischen Problemen:

- Sie können nicht überprüfen, ob die „Verteilung“ problemlos funktioniert
  Beispiel: Die Class-Dateien befinden sich im CLASSPATH oder im aktuellen Verzeichnis
  Konsequenz: ein eigentlich gewollter Download des Codes wird durch RMI umgangen.
- Ihr PC bekommt Atemprobleme:
  Auf Ihrem PC müssen mindestens drei Dienste gleichzeitig laufen:
  - der Lookup-Dienst
  - der eigentliche Dienst
  - Clients des Dienstes
  zudem vermutlich noch einige, wenn nicht alle weiteren Dienste (http, RMI).

Viele potentielle Probleme einer verteilten Anwendung werden dadurch verschleiert.

Weitere Probleme schaffen Sicherheitsrichtlinien. Sicher haben Sie in Ihrem Windows Profile Verzeichnis ein viel zu offenes Policy-File, ein Policy-File, in dem Sie Ihren Applikationen viel zu viele Zugriffsrechte und Ausführungsrechte geben. Damit reduzieren Sie die Fehlersuche beträchtlich, verhindern aber auch ein gründliches Testen.

Hier einige Tips aus unterschiedlichen Quellen und einiger Praxis:

- starten Sie mehrere http Server: für jedes Programm, welches Programmcode herunterladen soll, sollte ein eigener http Server gestartet werden
- beachten Sie die CODEBASE (ist die Datei lokal, über den CLASSPATH oder echt über einen http-Server zur richtigen Stelle gelangt?).
- benutzen Sie immer einen Security-Manager, also von Anfang an.
- Vermeiden Sie eine globale CLASSPATH Angabe. Meine Systemvariable CLASSPATH existiert überhaupt nicht. Ich definiere den CLASSPATH möglichst nur für genau ein Programm, also gleich bei dessen Ausführung (oder wie im Beispiel in einer Environment-Datei).
- Fassen Sie, wann immer mehrere Dateien heruntergeladen werden müssen, diese zu einem Archiv zusammen.

Vermutlich sind einige Kommentare zu diesen Guidelines angebracht.

1.2.1. Einsatz von HTTP- Servern

Wenn Sie, wie in unserem Beispiel, eine saubere Trennung des Codes für Client, Server und Jini erreichen wollen, ist der Einsatz mehrerer http-Server sinnvoll. Jeder dieser http Server besitzt sein eigenes Stammverzeichnis, sein eigenes Root. Ab diesem werden innerhalb der Anwendung die Class-Dateien geladen.

Der Einsatz mehrerer http-„Transporter“ reduziert auch das Risiko, dass beim Verteilen der Applikation kaum Überraschungen passieren.

Natürlich sollten Sie nicht übertreiben und für jede mobile Klasse einen eigenen Server (und Port) verbrauchen.

1.2.2. CODEBASE

Das Grundschema funktioniert folgendermassen:

- die CODEBASE wird auf dem Server gesetzt und
- vom Server dem Client mitgeteilt.

Die CODEBASE sollte nicht **`file://...`** enthalten. Sie werden sonst beim Deployen, beim Verteilen oder installieren auf mehreren Rechnern, viele Probleme haben.

Auch etwas, was Sie in meinen Beispielen finden, der Einsatz von **`localhost`**, sollten Sie nach Möglichkeit vermeiden. Der Grund ist derselbe wie beim Einsatz des Dateisystems: unter Umständen werden Dateien nicht über einen „verteilten Mechanismus“, sondern einfach lokal geladen, also unter Umgehung der Objekt-Serialisierung, Lookup und all diesen Mechanismen.

1.2.3. Security Manager

Obschon ich in meinen Programmbeispielen überwiegend auf Security verzichte, wird in der Regel ein Security-Manager gesetzt. Damit wird es möglich später vor dem praktischen Einsatz eine andere Policy zu definieren und auszutesten.

Beim Entwickeln der Grundfunktionalität macht es kaum Sinn, verschäfte Sicherheitsanforderungen einzubauen, ausser es handelt sich um ein Security Thema.

1.2.4. CLASSPATH

Falls Sie Ihre Client und Server und Service-Klassen in den CLASSPATH aufnehmen, gehen Sie das Risiko ein, dass die JVM, die Java Virtual Machine, die Dateien lokal laden kann, also einmal mehr nicht über einen verteilten Mechanismus.

Sie sehen in den meisten meiner Beispiele, dass lediglich jene Klassen aufgenommen werden, welche auch expliziz benötigt werden. Bei Jini sind diese Basisklassen in der Umgebungsdatei in einen generellen CLASSPATH aufgenommen. Die eigentlichen Anwendungsklassen werden jedoch nur lokal, also beim Aufruf der JVM aufgenommen, mit der –cp Option. Auf jeden Fall nicht aufnehmen sollten Sie alle ...–dl.jar Dateien.

1.2.5. Einsatz von Archiven

Eine gute Methode dafür zu sorgen, dass die benötigten Klassen und nur diese heruntergeladen werden können, ist die Bildung von Archiven.

Jini verwendet dieses Schema:

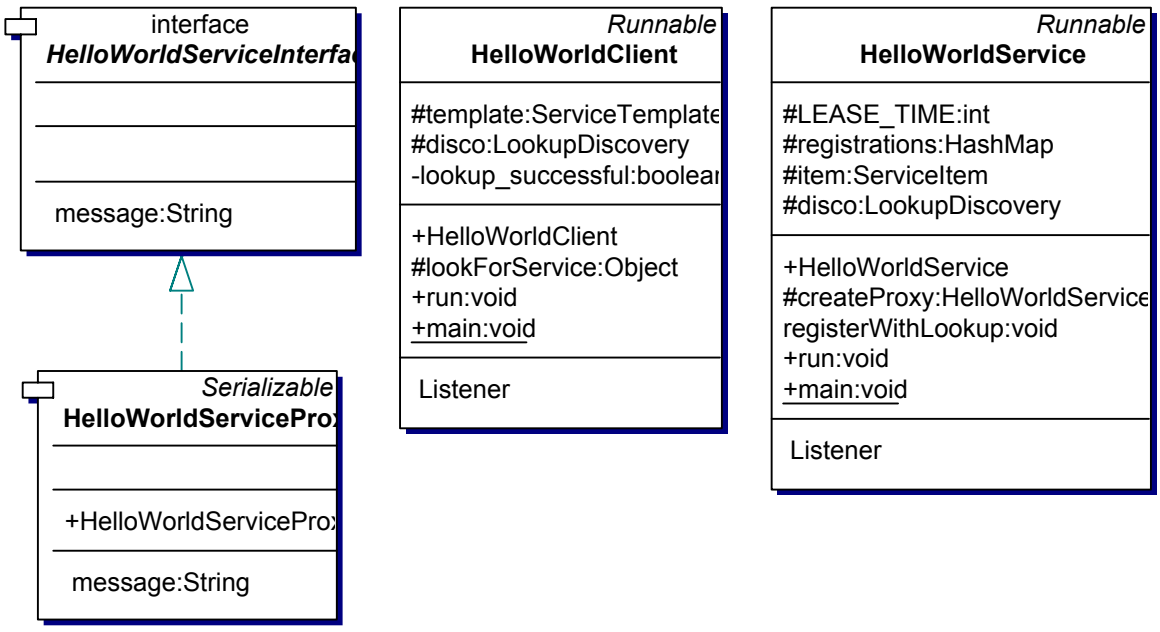
- reggie.jar enthält die Klassen des Lookup-Dienstes
- reggie-dl.jar wird vom Client bei Bedarf heruntergeladen.

1.3. Kommentare zum HelloWorld Programm

Dieses Beispiel besteht aus einem Interface, in dem die Funktionalität des Services beschrieben wird:

- Dieses Interface wird im Client benutzt, um das Ergebnis des Lookups zu Casten.
- Es wird auch von einem Proxy-Objekt implementiert, also dem eigentlichen Service.
- Im Server wird das Interface benutzt, um das Proxy-Objekt zu beschreiben, Der Server meldet diesen Dienst beim Lookup-Service an.

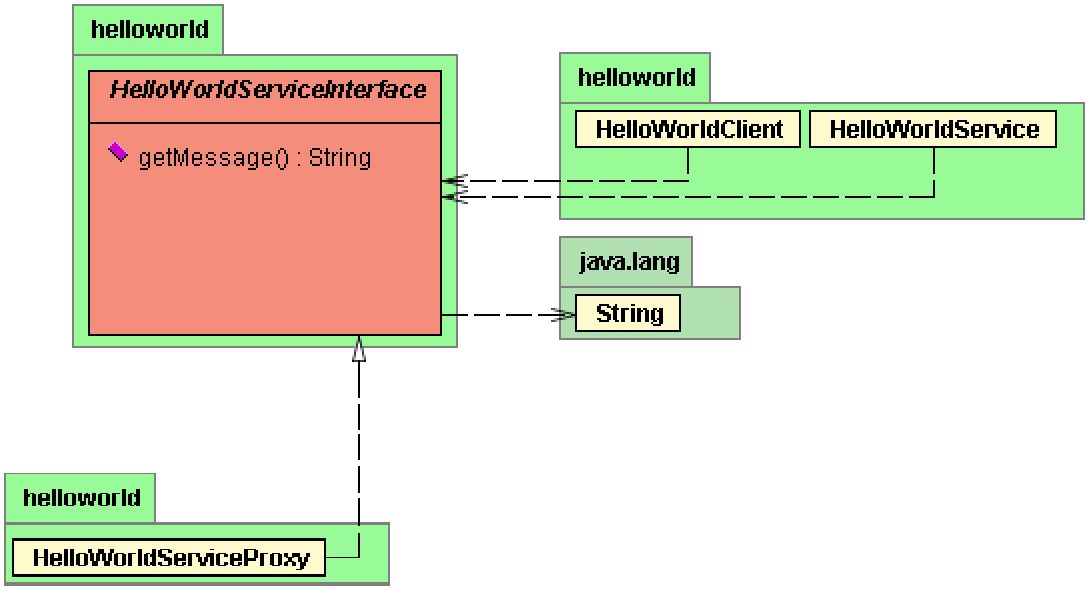
Das folgende Diagramm zeigt also nur einen Auszug der eigentlichen Verflechtung :



HelloWorldService ist eine „Mantelanwendung“, ein „Wrapper“, der den Service beim Lookup-Dienst anmeldet und dafür sorgt, dass der Dienst nicht als nicht mehr nötig an den verteilten Garbage Collector übergeben wird. Wir kommen darauf noch zurück. Diese Klasse wird für den eigentlichen Service nicht benötigt.

1.3.1. Das HelloWorldServiceInterface

Das Interface beschreibt eine einzige einfache Methode: das Ergebnis eines Methodenaufrufs. Das Proxy Objekt implementiert dieses Interface (der Client und der Server nutzen das Interface für die abstrakte Beschreibung des Dienstes, und nutzen das Proxy-Objekt, zur Definition des Services bzw. dessen Nutzung).



Das **`String`**-Objekt ist die Message, welche die Methode **`getMessage()`** an den Aufrufer sendet.

```
// Interface, welches durch den Service proxy implementiert
// wird.

package jini.helloworld; //können Sie erst mal ignorieren

public interface HelloWorldServiceInterface {
    public String getMessage();
}
```

Dieses Beispiel zeigt nur einen winzigen Teil der Komplexität verteilter Anwendungen:

- Unser Dienst, das Proxy-Objekt, kommt ohne weitere Dienste, Geräte oder Backend-Prozessen aus.
- Unser Dienst sendet selbständig eine Meldung an den Client.

Trotzdem benötigen wir eine „Mantelanwendung“, einen „Wrapper“, der den Service beim Lookup-Dienst anmeldet und dafür sorgt, dass der Dienst nicht als nicht mehr nötig an den verteilten Garbage Collector übergeben wird. Wir kommen darauf noch zurück.

1.3.2. Jini-Packages

Sun hat folgende Konventionen beachtet beim Einpacken der Klassen in Packages:

- Kern des Jini Systems steckt in den net.jini.core-Klassen.
  Diese befinden sich im Archiv jini-core.jar.
- Klassen, welche ausschliesslich mithilfe der Core-Klassen erstellt wurden, sind vom Typus net.jini.*
  Diese befinden sich in jini-ext.jar.
- Schliesslich stellt Sun einige Hilfsklassen zur Verfügung.
  Diese befinden sich im Archiv sun-util.jar.

Im HelloWorld Beispiel verwenden wir zudem RMI und seine Klassen und Archive. Diese besprechen wir hier aber nicht.

Unser Beispiel umfasst jede Menge dieser Basisklassen. Sie sehen dies an folgendem Beispiel

- Jini Core-Klassen:

```
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.discovery.LookupDiscovery;
import net.jini.core.lookup.ServiceItem;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceRegistration;
```

- Java Basisklassen (Eingabe/Ausgabe, Objektserialisierung)

```
import java.io.IOException;
import java.io.Serializable;
```

- Java RMI (Remote Methode Invocation) Klassen

```
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;
```

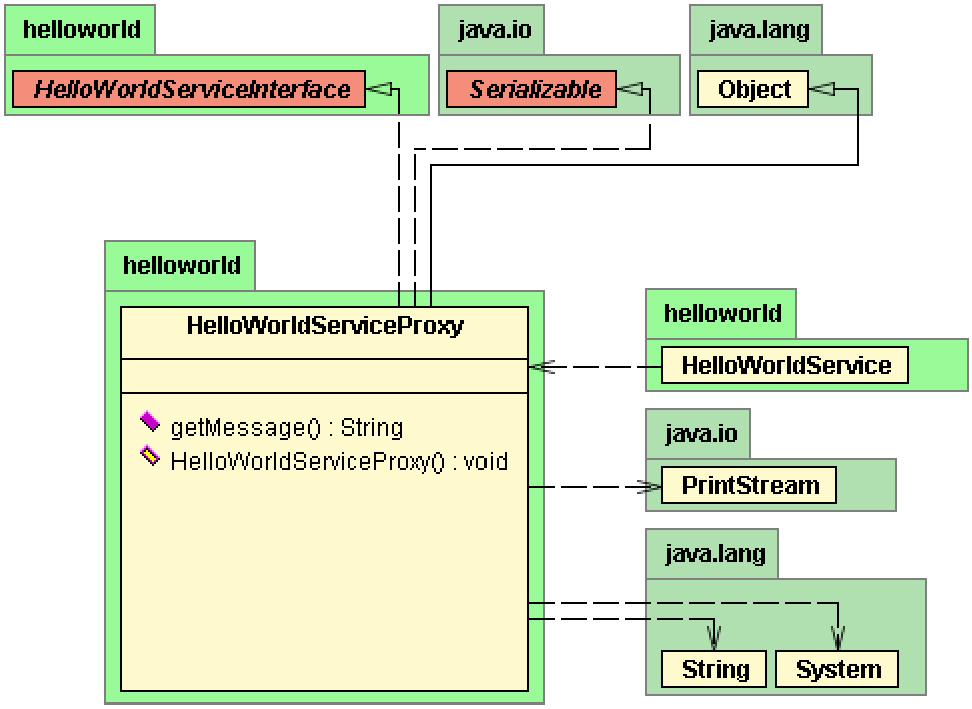
- Java Hilfsklassen (Java Collection: Maps, Hashtable, u.v.a.m.)

```
import java.util.*;
```

1.3.3. Die HelloWorldServiceProxy-Klasse

Unser eigentliches Service-Objekt stammt ist eine Instanz der **`HelloWorldServiceProxy`** Klasse.

Diese Klasse besitzt folgende Struktur:



Wie wir oben gesehen haben, implementiert diese Klasse vor allem unser **`HelloWorldServiceInterface`**, neben all den anderen Klassen, die Sie im obigen Diagramm auch noch erkennen können.

```
// Proxy Objekt
// Der Client lädt diese Klasse über das Netz,
// als serialisiertes Objekt,
// welches das Interface HelloWorldServiceInterface

class HelloWorldServiceProxy implements Serializable,
    HelloWorldServiceInterface {
    public HelloWorldServiceProxy() {

System.out.println("[HelloWorldServiceProxy]HelloWorldProxy()"
);
    }
    public String getMessage() {

System.out.println("[HelloWorldServiceProxy.HelloWorldProxy()]
getMessage()");
        return "Hello, world!";
    }
}
```

Die Struktur der Klasse erkennen wir aus dem obigen UML Diagramm (JBuilder).

- Die Klasse implementiert das Interface **`java.io.Serializable`**.
  Dies ist ein Muss, weil die Proxy-Klasse über das Netz an den Lookup-Dient gesendet werden muss. Der Lookup-Dienst sendet das Objekt dann zu jedem Client. Das serialisierte Objekt wird als Bytefolge mithilfe von RMI bzw. Sockets an das Remote-System gesendet und dort rekonstruiert.
- Der Dienst-Proxy implementiert die Schnittstelle HelloWorldServiceInterface, die dem Client bekannt ist (wie wir gleich sehen werden, oder oben bereits gesehen haben). Diese Schnittstelle sollte möglichst einfach aussehen!
- Da der Dienst serialisierbar sein soll, muss er einen argumentlosen Konstruktor besitzen, sonst könnte die Klasse nicht serialisiert bzw. deserialisiert werden!
- Die Klasse selbst ist nicht public. Damit hat der Client keine Chance einen Shortcut zu nehmen. Der Client muss die Klasse über den Lookup-Dienst anfordern und erhält die Klasse serialisiert.
- Die Klasse könnte auch als innere Klasse definiert werden. Das hat einen enormen Nachteil:
  in diesem Fall würde bei der Objektserialisierung auch das umschliessende Objekt serialisiert, also die Kommunikation unnötig aufgebläht, sowie unnötige Abhängigkeiten aufgebaut.
  Wenn Sie die Service-Klasse als static definieren, geschieht die Serialisierung wieder wie gehabt, also ohne das umschliessende Objekt.
  Aber eine separate Klasse ist einfacher zu kontrollieren.

Falls Sie die Serialisierung vergessen:

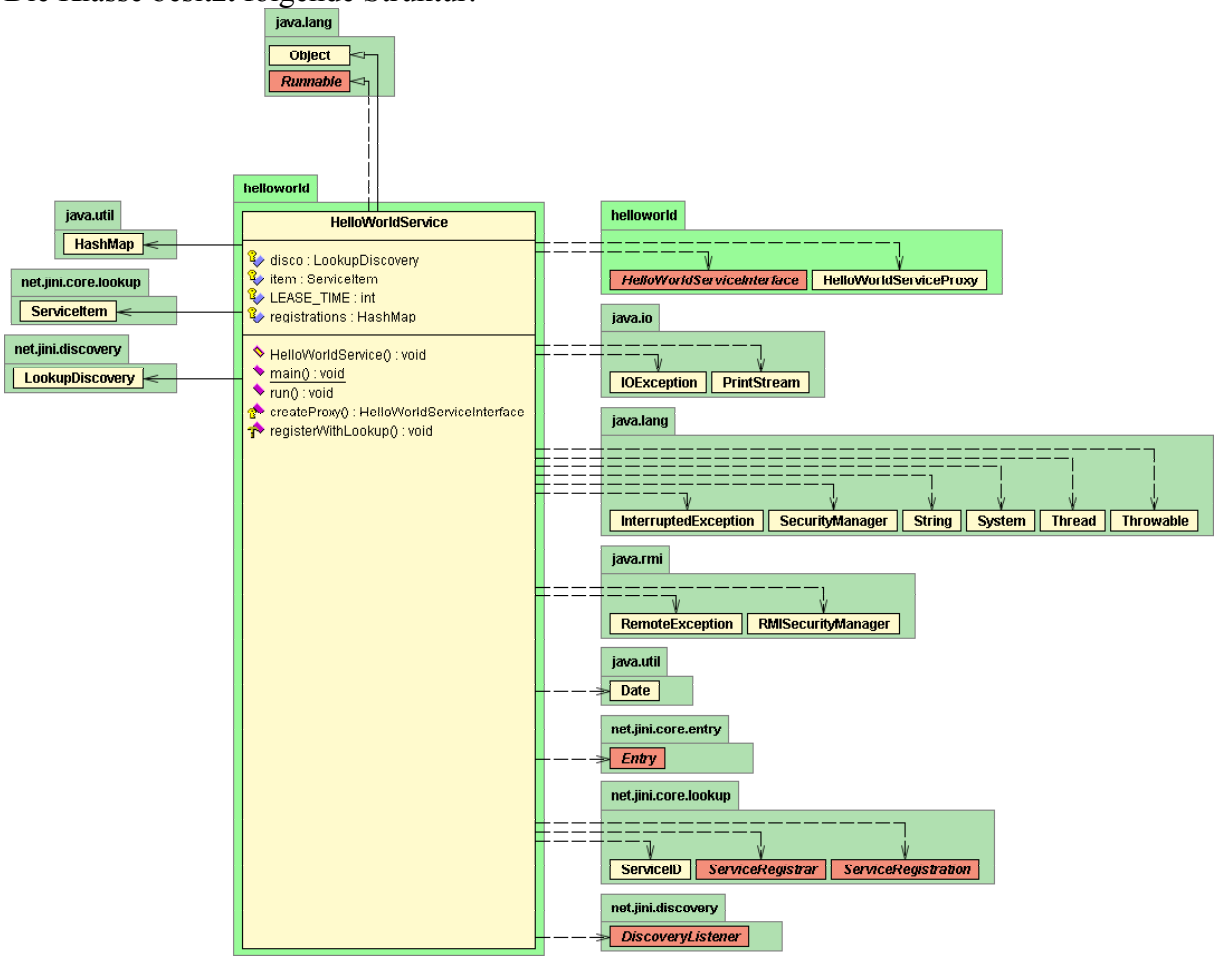
- Dann wird bei der Ausführung eine **`java.io.NotSerializableException`** geworfen, also der Fehler erst zur Ausführungszeit entdeckt.

- Wie wir noch sehen werden, wird das Proxy-Objekt, das Service-Objekt, an die Klasse ServiceItem übergeben. Diese Klasse übergibt eine Instanz (als **`Object`**) des Proxies an den Lookup-Dienst. Weil ein beliebiges Objekt übergeben werden kann, wird auch nicht überprüft, ob dieses Objekt serialisierbar ist.

1.3.4. Die HelloWorldService Klasse

Der Name ist nicht optimal gewählt, denn diese Klasse meldet den Service lediglich an und sorgt dafür das das Leasing erhalten bleibt (siehe unten).

Die Klasse besitzt folgende Struktur:



Die Aufgabe dieser Klasse ist das Suchen (Discovery) von Lookup-Diensten und die Veröffentlichung des Proxies. Das Zusammenspiel zwischen dem Lookup-Dienst und unserer Klasse geschieht mithilfe der inneren Klasse **`Listener`**, welche die Schnittstelle **`DiscoveryListener`** implementiert:

```
    // Innerere Klasse : Listener für Discovery Events
    class Listener implements DiscoveryListener {
        // Diese wird aufgerufen,
        // falls ein neuer Lookup Service gefunden wird.
        public void discovered(DiscoveryEvent ev) {
System.out.println("[HelloWorldService$Listener.discovered()]Es
wurde ein neuer Lookup Service entdeckt!");
            ServiceRegistrar[] newregs = ev.getRegistrars();
            for (int i=0 ; i<newregs.length ; i++) {
                if (!registrations.containsKey(newregs[i])) {
                    registerWithLookup(newregs[i]);
                }
            }
        }
        public void discarded(DiscoveryEvent ev) {
```

```
System.out.println("[HelloWorldService$Listener.discarded()]
Entfernen von Lookup Services!");
            ServiceRegistrar[] deadregs = ev.getRegistrars();
            for (int i=0 ; i<deadregs.length ; i++) {
                registrations.remove(deadregs[i]);
            }
        }
    }
```

Die Funktionsweise dieser Klasse werden wir noch kennen lernen. Für den Moment nur soviel:

- es geht auch ohne diese innere Klasse
- die Einführung dieser Klasse erlaubt eine bessere Aufgabentrennung.
  Die innere Klasse ist für den Discovery-orientierten Teil der Service-Wrapper-Klasse zuständig.

Nach der Deklaration dieser inneren Hilfsklasse wird der Konstruktor der Service-Wrapper-Klasse definiert:

```
    public HelloWorldService() throws IOException {
 System.out.println("[HelloWorldService]HelloWorldService()");
        item = new ServiceItem(null, createProxy(), null);
        // Security Manager setzen (RMI)
System.out.println("[HelloWorldService.HelloWorldService()]set
SecurityManager()");
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new
                            RMISecurityManager());
        }
        // Suchen der"public" Group
        // Dieser entspricht ein leerer String
System.out.println("[HelloWorldService.HelloWorldService()]Loo
kupDiscovery()");
        disco = new LookupDiscovery(new String[] { "" });
        // Installiere einen Listener.
System.out.println("[HelloWorldService.HelloWorldService()]add
Listener()");
        disco.addDiscoveryListener(new Listener());
    }
```

Hier der Ablauf im Konstruktor:

1) zuerst wird ein **`ServiceItem`** kreiert. Instanzen dieser Klasse werden den Lookup-Diensten während des Registrierungsprozesses übergeben.
```
item = new ServiceItem(null, createProxy(), null);
```

2) Der Konstruktor dieser **`ServiceItem`**-Klasse besitzt drei Argumente:
   1. Dienst-ID – eine global eindeutige Identifikation des Dienstes: hier **`null`**
      Falls das Argument **`null`** ist, stellt der Lookup-Dienst eine eindeutige ID zur Verfügung.

   2. Instanz des Proxy-Dienstes – serialisierbar, welche nach dem Endecken des Lookup-Dienstes an diesen gesendet wird. Dort wartet das Objekt auf Clients.

      In unserem Beispiel wird mit der Methode **`createProxy`**() ein Proxy-Objekt kreiert.
      Auch in diesem Fall könnten wir auf diese Methode verzichten. Die Definition der Methode hilft das Programm besser zu strukturieren.

   3. Liste von Attributen, die mit dem Dienst-Proxy verknüpft werden.
      Beispiel: Attribute bezüglich eines Peripheriegerätes (Ort, Modell, ...)

      Das Argument ist ein Array von Objekten, welche das Interface **`Entry`** implementieren.

      In unserem brauchen wir kein Detaillierung des Dienst-Proxies, daher **`null`**.

3) Prüfen, ob ein **`SecurityManager`** definiert und installiert ist.
```
if (System.getSecurityManager() == null) {
     System.setSecurityManager(new RMISecurityManager());
}
```
Sollte dies nicht der Fall sein, installieren wir den RMI Security-Manager.

Der **`SecurityManager`** ist in Java immer dann möglich, wenn wir Code von irgendwo herunterladen, also nicht über den CLASSPATH geladen werden.

In unserem Beispiel müssen wir einen Security-Manager implementieren, weil wir die Proxy-Klasse zu unterschiedlichsten Clients herunterladen möchten.

4) Kreieren einer Instanz der **LookupDiscovery** Klasse:
```
disco = new LookupDiscovery(new String[] { "" });
```
- Diese Klasse hilft uns das Jini-Discovery-Protokoll sinnvoll zu nutzen.
- Das Objekt **disco** kann einen **DiscoveryListener** aufrufen, sobald ein neuer Lookup-Dienst gefunden wird. In unserem Fall suchen wir die **public** Gruppe. Dieser entspricht ein leerer String. Gruppen sind Namen für Jini-Gemeinschaften.

5) Hinzufügen des Listeners zum **LookupDiscovery** Objekt:
```
disco.addDiscoveryListener(new Listener());
```
Hier treffen wir auf die oben erwähnte innere Klasse **Listener**, welche das **DiscoveryListener** Interface implementiert. Der Listener informiert uns also über allfällig neu entdeckte Lookup-Services.

1.3.4.1. Asynchrone Kommunikation des Discovery-API

Jini verwendet ausschliesslich für die Discovery einen asynchronen Listener-Mechanismus. Der Grund ist einzig und allein der, dass Jini damit umgehen muss, dass eventuell ein Lookup-Service erst nach einer Anfrage gestartet wurde. Wèrde dieser Mechanismus synchron aufgebaut, könnte er zu unnötigen Verzögerungen und Blockierungen führen.

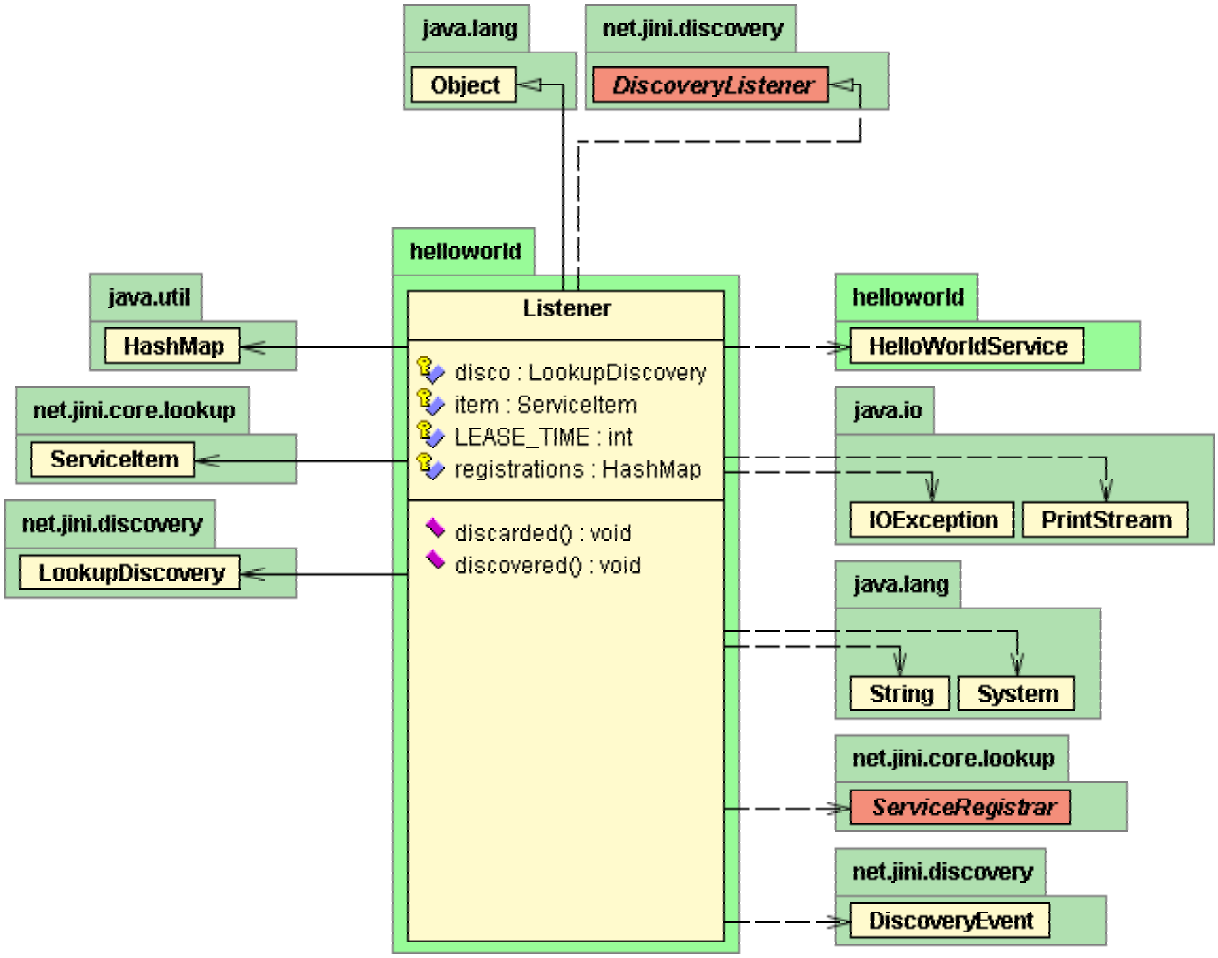
1.3.5. Die (innere) Klasse Listener – Discovery und Lookup

Diese Hilfsklasse unterstützt die Discovery. Sie implementiert die Schnittstelle **`DiscoveryListener`**, mit den zwei Methoden **`discovered`**() und **`discarded`**():

- **`discovery`**() wird aufgerufen, sobald ein Lookup-Dienst gefunden wurde.
- **`discarded`**() wird aufgerufen, sobald ein bekannter Dienst nicht mehr zur Verfügung steht oder wenn man die Gruppe ändern möchte.

```
class Listener implements DiscoveryListener {
    public void discovered(DiscoveryEvent ev) {
    public void discarded(DiscoveryEvent ev) {
}
```

Beiden Methoden wird als Argument ein **`DiscoveryEvent`** übergeben. Dieses beschreibt den Lookup-Dienst, der entdeckt oder verworfen wurde.



1.3.5.1. Wann müssen Lookup-Dienste gelöscht werden

Die Methode **`LookupDiscovery.discard(DiscoveryEvent)`** wird aufgerufen, falls:

1) Falls ein Lookup-Dienst für unsere Gruppe nicht mehr benötigt wird, beispielsweise weil Sie die Gruppe wechseln.

2) Falls Sie diesen Dienst expliziz nicht mehr benötigen, der Standardfall für diesen Methodenaufruf.
   Dieser Fall kann beispielsweise durch eine Exception ausgelöst werden, eine Exception, die anzeigt, dass der Lookup-Dienst abgestürzt oder fehlerhaft ist.

   Der Lookup-Dienst hilft Ihnen beim Wiederentdecken des Dienstes, falls er wieder aktiv und brauchbar wird.

1.3.5.2. Die `LookupDiscovery.discovery()` Methode

In unserem Fall sieht diese Methode recht einfach aus:

```
      protected HashMap registrations = new HashMap();
......

public void discovered(DiscoveryEvent ev) {
System.out.println("[HelloWorldService$Listener.discovered()]
Es wurde ein neuer Lookup Service entdeckt!");
            ServiceRegistrar[] newregs = ev.getRegistrars();
            for (int i=0 ; i<newregs.length ; i++) {
                if (!registrations.containsKey(newregs[i])) {
                  try {
                    HelloWorldService hws = new
                                    HelloWorldService();
                    hws.registerWithLookup(newregs[i]);
                  } catch(IOException ioe) {}
                }
            }
}
```

Immer wenn uns das Discovery-Protokoll mitteilt, dass ein neuer Lookup-Dienst gefunden wurde, also die Methode **`discovered`**() aufgerufen wird, wird überprüft, ob in der Collection (**`HashMap`**) dieser Dienst bereits vorhanden ist.
(`if (!registrations.containsKey(newregs[i]))`)

- ist dies der Fall, geschieht nichts.
- ist dies *nicht* der Fall, speichern wir ihn in der Collection und lassen unseren Dienst bei ihm registrieren, mit `hws.registerWithLookup(newregs[i]);`

1.3.5.3. Die `HelloWorldService.registerWithLookup()` Methode

Die Registrierung selber ist unspektakulär:

- In der obigen Methode wird ein neues **`HelloWorldService`** Objekt kreiert und dessen Methode **`registerWithLookup`**() aufgerufen.

- In der Methode **`registerWithLookup`**() wird mithilfe der **`ServiceRegistrar`** Objekte, welches zum neu entdeckten Lookup-Service gehört, registriert:

```
registration = registrar.register(item, LEASE_TIME);
```

```
    synchronized void registerWithLookup(ServiceRegistrar
                                         registrar) {
System.out.println("[HelloWorldService]registerWithLookup()");
        ServiceRegistration registration = null;
        try {
            registration = registrar.register(item,
                                              LEASE_TIME);
        } catch (RemoteException ex) {
System.out.println("\t\t\t[RemoteException]Registrierung
schlug fehl: " + ex.getMessage());
            return;
        }
        // Ist dies die erste Registrierung?
        // Falls ja: Service-ID abspeichern
System.out.println("[HelloWorldService.registerWithLookup()]
getServiceID()");
        if (item.serviceID == null) {
            item.serviceID = registration.getServiceID();
System.out.println("\t\t\tServiceID : " + item.serviceID);
        }
        registrations.put(registrar, registration);
    }
```

Der **`ServiceRegistrar`** ist definiert als ein Interface auf den Lookup-Service.

```
public ServiceRegistration register(ServiceItem item,
                                    long leaseDuration)
                             throws java.rmi.RemoteException
```

Die Methode **`ServiceRegistrar.register()`** besitzt zwei Argumente:

- Das erste Argument ist das **`ServiceItem(null, createProxy(), null);`**

- Das zweite Argument bestimmt die Leasingdauer, in unserem Beispiel 10 Minuten.

1.3.5.4. Redundante Lookup-Dienste

Mit der Collection, dem HashMap, vermeiden wir eine mehrfache Registrierung eines Service-Objekts bei einem Lookup-Dienst.

Dies ist nicht unbedingt nötig:

- wird ein Service mehr als einmal registriert, so wird jeweils einfach der bereits vorhandene Eintrag überschrieben.

1.3.5.5. Registrierungs-Thread

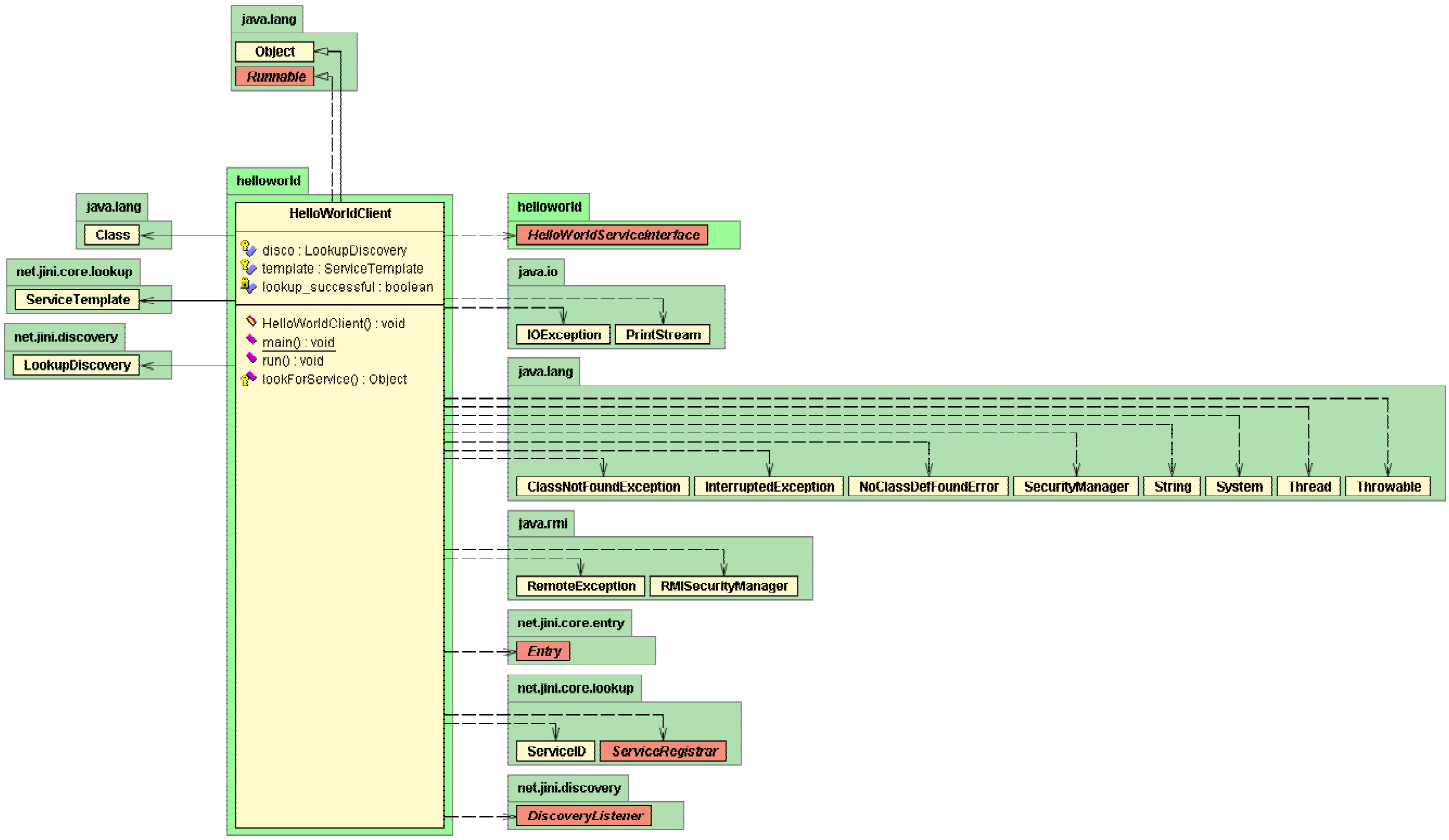
In unserem Beispiel haben wir lediglich einen einzigen Thread definiert, der sowohl für Lookup als auch für die Registrierung zuständig ist.

Der Registrierungsprozess kann einige Zeit in Anspruch nehmen. Daher ist es besser, ihn in einem separaten Thread laufen zu lassen.

In unserem Fall können während der Registrierung keine Lookups durchgeführt werden.

1.3.6. Der Client

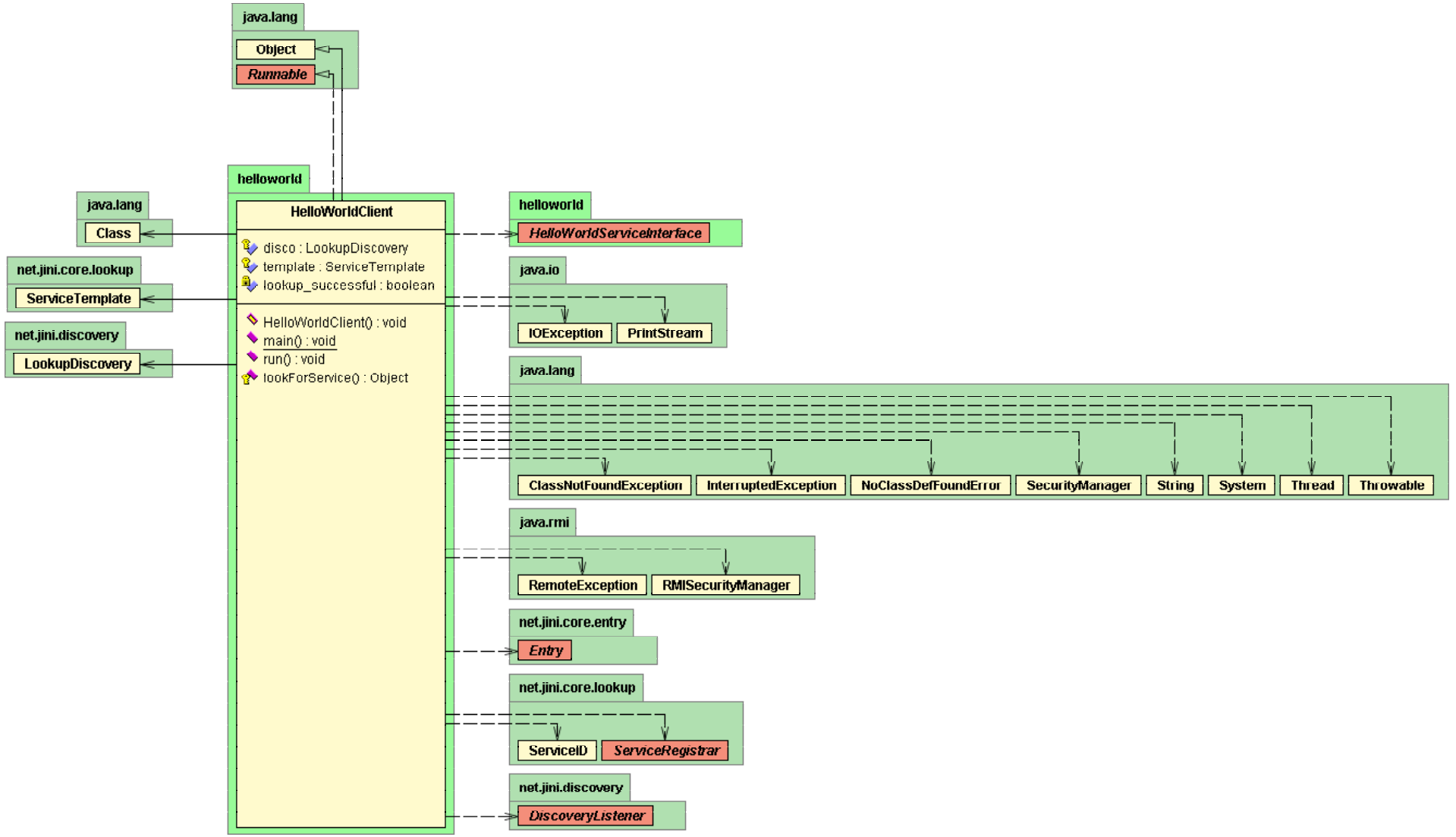
Der Client sieht ähnlich aus wie der Server.



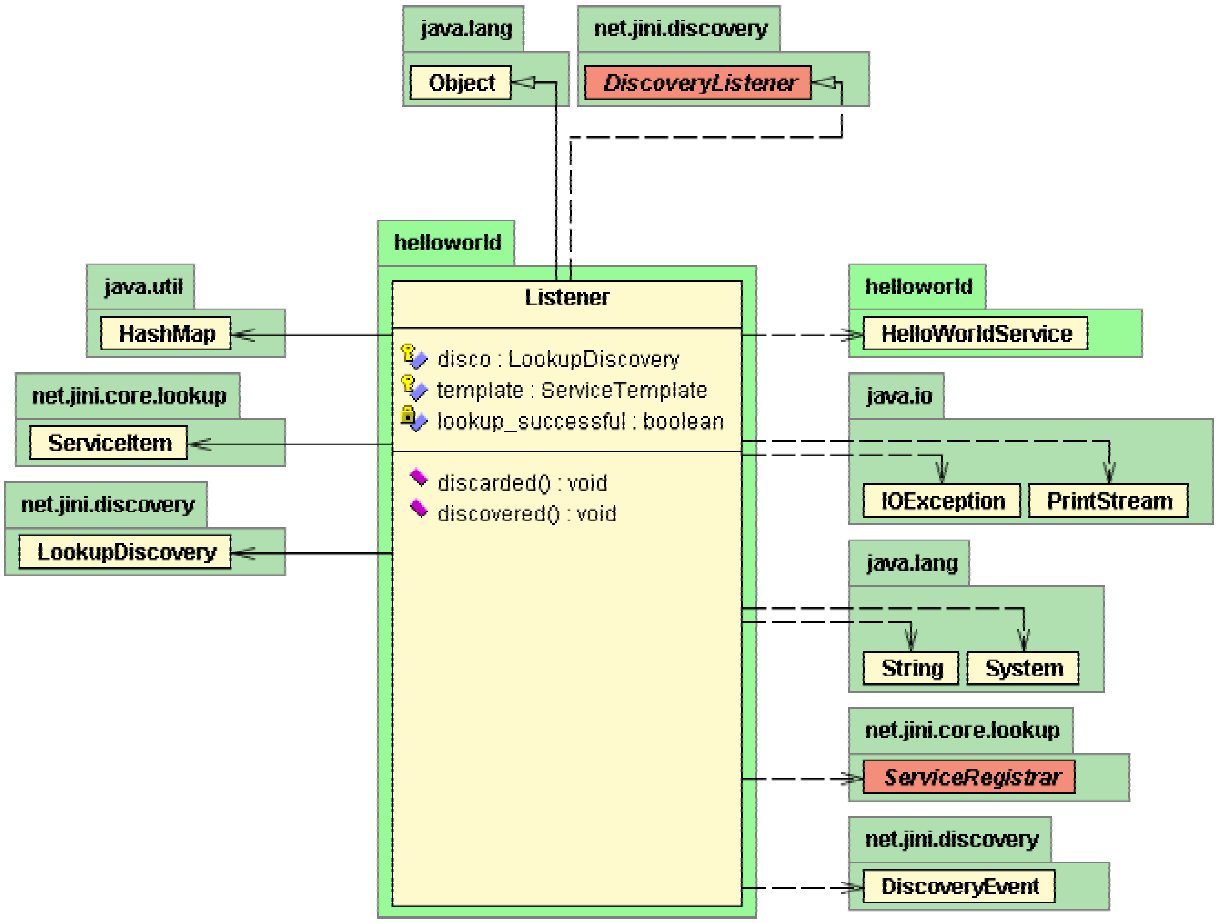
Auch er besitzt eine innere Klasse DiscoveryListener, welche für die Discovery-Protokolle zuständig ist.

Wichtig ist auch im Fall des Clients der Konstruktor. Dieser beschreibt die eigentliche Dynamik des Clients:

```
public HelloWorldClient() throws IOException {
    Class[] types = { HelloWorldServiceInterface.class };
    if (System.getSecurityManager() == null) {
        System.setSecurityManager(
                new RMISecurityManager());
    }
    disco = new LookupDiscovery(new String[] { "" ," "});
    pDiscovery.addDiscoveryListener()");
    disco.addDiscoveryListener(new Listener());
}
```


java.lang
Object
Runnable
helloworld
HelloWorldClient
disco : LookupDiscovery
template : ServiceTemplate
lookup_successful : boolean
HelloWorldClient() : void
main() : void
run() : void
lookForService() : Object
java.lang
Class
net.jini.core.lookup
ServiceTemplate
net.jini.discovery
LookupDiscovery
helloworld
HelloWorldServiceInterface
java.io
IOException
PrintStream
java.lang
ClassNotFoundException
InterruptedException
NoClassDefFoundError
SecurityManager
String
System
Thread
Throwable
java.rmi
RemoteException
RMISecurityManager
net.jini.core.entry
Entry
net.jini.core.lookup
ServiceID
ServiceRegistrar
net.jini.discovery
DiscoveryListener

1.3.6.1. Die innere Listener Klasse des Clients



Der Aufbau ist analog zur Listener (inneren) Klasse in der Server / Service Applikation, daher auch der selbe Namen.

```
class Listener implements DiscoveryListener {
  //protected ServiceTemplate template;
  //protected LookupDiscovery disco;
  //private boolean lookup_successful=false;

    public void discovered(DiscoveryEvent ev) {
        ServiceRegistrar[] newregs = ev.getRegistrars();
        HelloWorldClient hwc=null;
        try {
            hwc = new HelloWorldClient();
        } catch(IOException ioe) {}
        for (int i=0 ; i<newregs.length ; i++) {
            hwc.lookForService(newregs[i]);
        }
    }
    public void discarded(DiscoveryEvent ev) {
    }
}
```

1.3.6.2. Der Konstruktor des Clients

Der Konstruktor sieht auch ähnlich aus wie beim Service/Server:

```
public HelloWorldClient() throws IOException {
    Class[] types = { HelloWorldServiceInterface.class };
    template = new ServiceTemplate(null, types, null);

    // Security Manager (RMI) setzen
    if (System.getSecurityManager() == null) {
        System.setSecurityManager(
                new RMISecurityManager());
    }

    // Public Group durchsuchen
      disco = new LookupDiscovery(new String[] { "" ," "});

    // Listener installieren
    disco.addDiscoveryListener(new Listener());
}
```

- Der Grundaufbau besteht im Kreieren eines **`template`** Objekts. Diesem widmen wir uns gleich noch.
- Auch im Client müssen wir den Security-Manager von RMI einbauen, da der Client das Proxy-Objekt herunterladen soll.
- Der Client sucht die public Gruppe (leerer String, das Blank kann weggelassen werden).
- Dann hängen wir dem Lookup-Service einen Listener an, damit wir über Aktivitäten informiert werden.

1.3.6.3. Templates für das Finden von Jini Diensten

Das ServiceTemplate wie wir es im Client verwenden, ist folgendermassen definiert:

```
public class ServiceTemplate extends java.lang.Object
                        implements java.io.Serializable
```

Auf den Konstruktor gehen wir genauer ein:

ServiceTemplate(
ServiceID serviceID,
java.lang.Class[] serviceTypes,
Entry[] attrSetTemplates
)

Der ServiceTemplate-Code verhält sich wie die Spezifikation einer Abfrage. Wenn Sie ein ServiceTemplate an einen Lookup-Server übergeben, durchsucht dieser alle bei ihm registrierten Dienste.

Ein Template / Vorlage stimmt mit einem Dienst überein, falls:

- die Dienst-ID in der Vorlage mit der Dienst-ID eines registrierten Dienstes übereinstimmt oder im entsprechenden Feld der Vorlage **`null`** steht
  und
- der registrierte Dienst eine Instanz oder ein Subtyp *jedes* Typs im Feld **`serviceType`** der Vorlage ist oder in diesem Feld null steht
  und
- die Attributliste des Dienstes mindestens ein Attribut enthält, das mit jedem Eintrag in der Vorlage übereinstimmt oder im Attributfeld der Vorlage **`null`** steht.

ServiceTemplate sind also Hilfsobjekte, mit denen Dienste gesucht werden können, wobei auch polymorph gesucht wird (Unterklassen können Stellvertreter der Oberklasse sein).

In unserem Beispiel lassen wir den ersten und dritten Parameter offen. Der mittlere Parameter muss gleich dem Dienst gesetzt werden, den wir suchen, als Klasse: **`Class[] types = { HelloWorldServiceInterface.class }`**

Das Interface muss natürlich dem Client bekannt sein. In unserem Beispiel haben wir eine Kopie der Interface-Beschreibung ins Client Verzeichnis kopiert.

1.3.6.4. Suchen eines Dienstes

Die Lookup-Dienste werden durch **`ServiceRegistrar`**'s repräsentiert.

Diese Objekte besitzen eine Methode zum Suchen der Dienste: **`lookup()`**.

Falls eine Übereinstimmung festgestellt wird, wird dieses Objekt an den Client übermittelt, **`null`** falls nichts/keine Übereinstimmung gefunden wurde. Eventuell werden viele passende Dienste gefunden!

```
protected Object lookForService(ServiceRegistrar lusvc) {
    Object o = null;
    try {
        o = lusvc.lookup(template);
    } catch (RemoteException ex) {
        return null;
    }
    if (o == null) {
        System.err.println("Kein passender Service");
        return null;
    }
     ((HelloWorldServiceInterface)o).getMessage());
    return o;
}
```

Wir ignorieren alles bis auf das erste Objekt!

1.4. Das war's!

Damit haben wir unser erstes Beispiel einigermassen ausführlich besprochen!
Weitere Details können Sie erfahren, indem Sie sich mit dem Jini API vetraut machen.

1.5. Anhang – Die Jini-Dienste im Überblick

Das ist eigenlich die nullste Version eines Textes, den ich noch schreiben will:

- Wie funktionieren die Jini-Dienste und Protokolle im Detail?
- Wie lassen sie sich auf andere Plattformen transportieren, auch ohne RMI und diesen ganzen Overhead?
- Wie sehen typische Anwendungen aus? (ich möchte schliesslich auch meine Lego-Roboter – Jini Lösung vorstellen!)

1.5.1. Discovery und Lookup Protokolle

Kern des Jini Systems sind drei Protokolle:

- Discovery
- Join
- Lookup

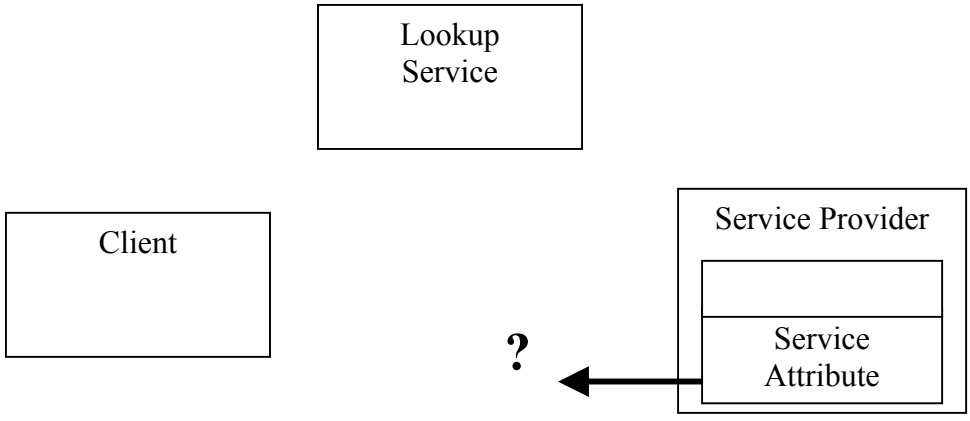
Discovery + Join arbeiten zusammen, wenn ein neues Jini-fähiges Gerät ins System eingefügt wird.

- Discovery geschieht beim Suchen eines Lookup-Services durch einen Service. Der Service möchte sich beim Lookup-Service registrieren.
- Join geschieht dann, wenn ein Service einen Lookup-Service gefunden hat und er diesem beitreten möchte.

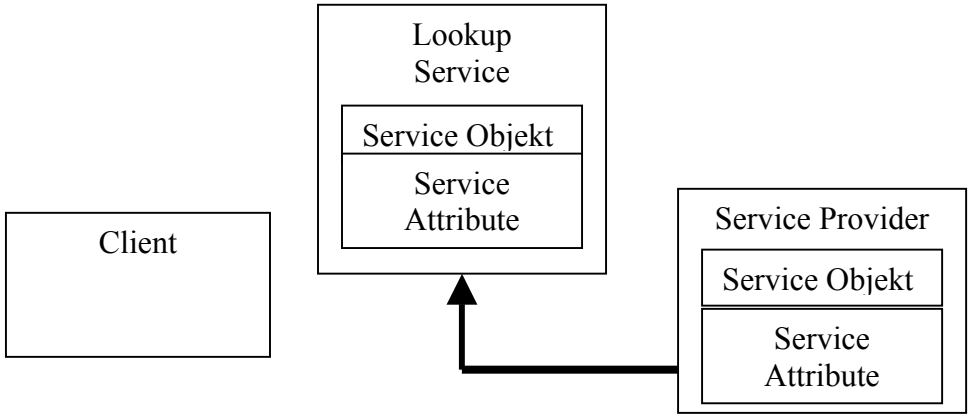
Lookup besteht darin, dass ein Client oder Benutzer einen verteilten Service lokalisieren muss und den Service gemäss dem Interface nutzen möchte.

1.5.1.1. Discovery

Ein Service sucht einen Lookup-Service:

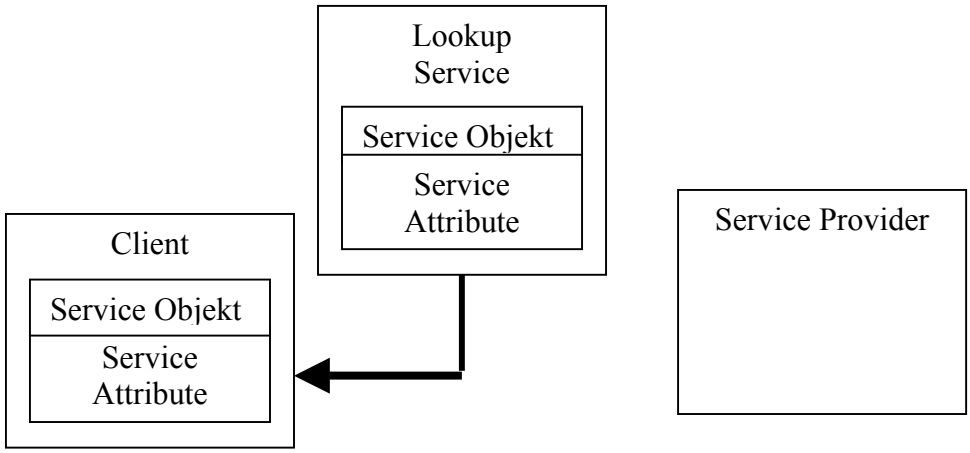


1.5.1.2. Join



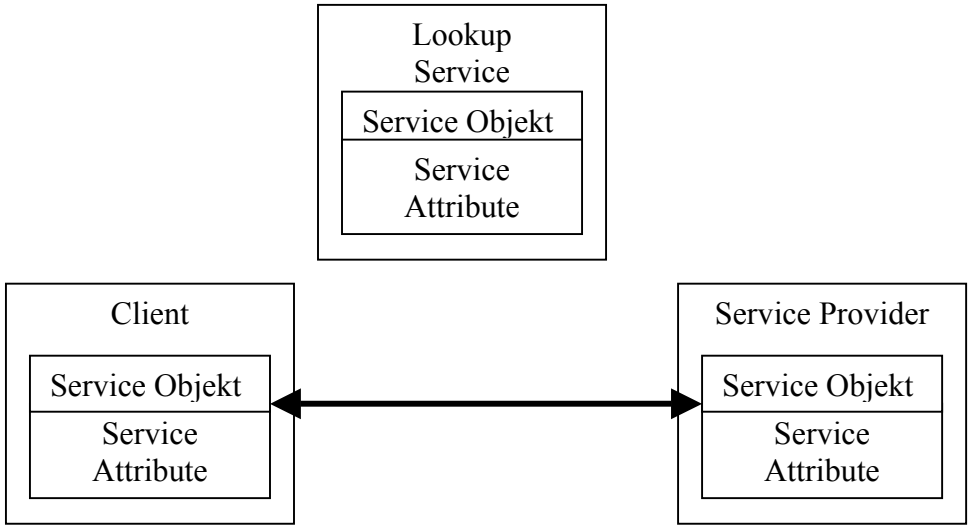
1.5.1.3. Lookup

Der Client möchte den Service nutzen. Er spezifiziert diesen beispielsweise mithilfe der Java Klasse (`Class[] types={HelloWorldServiceInterface.class}`) .
Eine kopie des Service Objekts wird zum Client verschoben und eingesetzt, um mit dem Service zu kommunizieren.



1.5.1.4. Service Invocation

Der Client interagiert mit dem Service Provider via Service-Objekt:



1.5.2. Hinweise

Ich habe die einzelnen Jini Klassen mit ToGether analysiert und zum Teil mit dem JBuilder getestet. Ich werde aber nun vermutlich auf Eclipse (Eclipse.org) wechseln.

In der Regel gehe ich immer gleich vor beim Einarbeiten in ein neues Thema.

1) ein Beispiel zum Laufen bringen
2) das Beispiel verstehen
3) das Beispiel abändern und parallel dazu die Theorie vertiefen.

Die Analyse-Tools, wie ToGether helfen einem sehr beim Versuch eine Übersicht zu gewinnen. Partiell nutze ich reverse Enginering (Jad, FrontEnd, SourceForge).

Viel Spass mit Jini, ich habs!