7 Entwurf einer Beispielapplikation

Um die in den vorhergehenden Kapiteln vorgestellten Themen im Zusammenspiel zu erleben, entwickeln wir in diesem Kapitel ein durchgängiges Beispiel. Darin werden XML und JSON zur Datenspeicherung und zum Austausch genutzt und schließlich Aufrufe von REST-Services in einer MongoDB protokolliert.

Als Beispiel wird eine Highscore-Verwaltung erstellt. Beginnend mit einem einfachen Programm, das eine Highscore-Liste aus einer CSV-Datei einlesen kann, wird die Funktionalität schrittweise ausgebaut. In mehreren Iterationen entsteht dann eine Applikation, die über das Netzwerk durch andere Applikationen angesprochen werden kann und dazu über eine REST-Schnittstelle verfügt. Außerdem wird ein simples Administrations-GUI in HTML angeboten. Kleine JavaScript-Elemente rufen den REST-Service auf. Eine kurze Einführung in HTML und JavaScript, die das Verständnis des hier entwickelten Web-GUI erleichtern, bieten jeweils die Anhänge C und D.

7.1 Iteration 0: Ausgangsbasis

Die Ausgangsbasis bildet ein Programm, das Highscore-Daten aus einer CSV-Datei einliest und diese als Liste aufbereitet. Diese CSV-Daten haben in etwa das nachfolgend gezeigte Format – damit das Ganze praxisnäher wird, soll unsere gleich realisierte Einlesekomponente auch mit unvollständigen Angaben zurechtkommen und diese bei der Weiterverarbeitung ignorieren. Außerdem sind Zeilen, die mit dem Zeichen # starten, Kommentare und sollen nicht ausgewertet werden:

```
# Name, Punkte, Level,   Datum
Matze,    1000,     7,   12.12.2012
ÄÖÜßöäü,   777,     5,   10.10.2010
Peter,     985,     6,   11.11.2011


# Fehlender Datumswert
Peter,     985,     6

# Falsches Format des Levels
Peter,     985,    K6,   11.11.2009

# Fehlerhaftes Datumsformat
Micha,     100,     1,   1/1/2001
```

Michael Inden, Der Java-Profi: Persistenzlösungen und REST-Services, dpunkt.verlag, ISBN 978-3-86490-374-8

Business-Objekt

Nachdem wir das Datenformat der Eingabe kennen, schauen wir uns die Implementierung der Klasse `Highscore` an. Sie bildet das einfache Domänen- oder Business-Modell. Diese Klasse verwaltet dazu einen Namen, einen Punktestand, ein Level sowie ein Datum vom Typ `LocalDate`:

```
public final class Highscore
{
    private final String name;
    private final int points;
    private final int level;
    private final LocalDate day;

    public Highscore(final String name, final int points,
                     final int level, final LocalDate day)
    {
        this.name = name;
        this.points = points;
        this.level = level;
        this.day = day;
    }

    // ...
}
```

CSV-Import von Highscores

Die Implementierung zum Einlesen von Highscore-Daten aus einer CSV-Datei wird nur kurz dargestellt, da der Fokus auf den nachfolgenden Erweiterungen liegt. Erwähnenswert ist, dass im Sourcecode verschiedene mit JDK 8 eingeführte Neuerungen, etwa `Files.readAllLines()`, `Iterable<E>.forEach()`, `Optional<T>` und Lambdas, zum Einsatz kommen. Bei `readAllLines()` ist es möglich, das Encoding der Eingabedaten anzugeben, hier durch die Konstante `ISO_8859_1` repräsentiert:

```
public final class HighscoresCsvImporter
{
    private static final Logger log =
                        Logger.getLogger("HighscoresCsvImporter");

    public static List<Highscore> readHighscoresFromCsv(final String fileName)
    {
        final List<Highscore> highscores = new LinkedList<Highscore>();

        try
        {
            // JDK 8: readAllLines(), forEach(), Optional<T> & Lambdas
            final List<String> lines = Files.readAllLines(
                            Paths.get(fileName), StandardCharsets.ISO_8859_1);
            lines.forEach(line ->
            {
                final Optional<Highscore> optHighscore =
                                        extractHighscoreFrom(line);
                optHighscore.ifPresent(highscore -> highscores.add(highscore));
            });
        }
```

```
        catch (final IOException e)
        {
            log.warning("processing of file '" + fileName + "' failed: " + e);
        }

        return highscores;
    }

    // ...
}
```

Die eigentliche Extraktionsarbeit findet in der im nachfolgenden Listing gezeigten Methode `extractHighscoreFrom(String)` statt. Beim Umwandeln der textuellen Informationen nutzen wir die Methoden `Integer.parseInt()` für Zahlen und `LocalDate.parse()` für Datumsangaben. Weil Letztere im deutschen Format vorliegen, verwenden wir hier einen speziellen `DateTimeFormatter`:

```
private static Optional<Highscore> extractHighscoreFrom(final String line)
{
    final int VALUE_COUNT = 4;

    // Spalte die Eingabe mit ';' oder ',' auf
    final String[] values = line.split(";|,");

    // Behandlung von Leerzeilen und Kommentaren sowie unvollständigen Einträgen
    if (isEmptyLineOrComment(values) || values.length != VALUE_COUNT)
    {
        return Optional.empty();
    }

    try
    {
        // Auslesen der Werte als String + Typprüfung + Konvertierung
        final String name = values[0].trim();
        final int points = Integer.parseInt(values[1].trim());
        final int level = Integer.parseInt(values[2].trim());
        final String dateAsString = values[3].trim();

        final DateTimeFormatter dateTimeFormatter =
                              DateTimeFormatter.ofPattern("dd.MM.yyyy");
        final LocalDate day = LocalDate.parse(dateAsString, dateTimeFormatter);

        return Optional.of(new Highscore(name, points, level, day));
    }
    catch (final NumberFormatException e)
    {
        log.warning("Skipping invalid point or level value '" + line + "'");
    }
    catch (final DateTimeParseException e)
    {
        log.warning("Skipping invalid date value '" + line + "'");
    }
    return Optional.empty();
}
```

Schließlich verbleibt noch die Hilfsmethode `isEmptyLineOrComment(String)`, die aber für das Beispiel nicht von Relevanz ist und daher hier nicht gezeigt wird. Diese finden Sie selbstverständlich im online verfügbaren Sourcecode zum Buch.

Michael Inden, Der Java-Profi: Persistenzlösungen und REST-Services, dpunkt.verlag, ISBN 978-3-86490-374-8

Beispielapplikation

Nachdem wir die Basiskomponenten betrachtet haben, wollen wir uns deren Kombination zum Einlesen der CSV-Datei und der Extraktion von `Highscore`-Objekten in Form einer `main()`-Methode anschauen.

```
public static final void main(final String[] args)
{
    final String filePath = "resources/Highscores.csv";

    final List<Highscore> highscores = readHighscoresFromCsv(filePath);

    highscores.forEach(System.out::println);
}
```

Listing 7.1 *Ausführbar als* '**READHIGHSCOREFROMCSVEXAMPLE**'

Starten wir das Programm READHIGHSCORESFROMCSVEXAMPLE, so kommt es zu folgender Ausgabe:

```
Highscore [name=Matze, points=1000, level=7, date=2012-12-12]
Highscore [name=ÄÖÜßöäü, points=777, level=5, date=2010-10-10]
Highscore [name=Peter, points=985, level=6, date=2011-11-11]
```

Erwartungskonform zur Implementierung werden die im CSV vorliegenden Kommentare übersprungen und nur die drei Zeilen der validen Nutzdaten in `Highscore`-Objekte konvertiert.

Design

Die Implementierung ist bislang überschaubar. Für die geplanten Erweiterungen empfiehlt es sich trotzdem, einen Überblick z. B. in Form eines UML-Klassendiagramms zu erstellen. Werfen wir also einen kurzen Blick auf das Design für eine Bestandsaufnahme und schauen dann in den nachfolgenden Iterationen, wie sich das Ganze entwickelt.

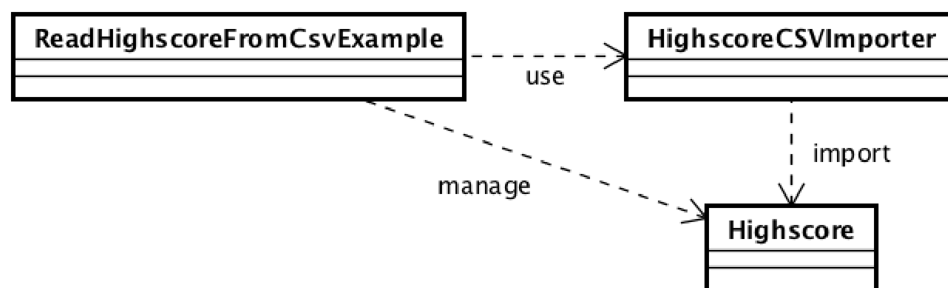

Abbildung 7-1 *Iteration 0 der Beispielapplikation*

Fazit

Wir haben nun die Ausgangsbasis kennengelernt. Diese einfache Applikation erlaubt jedoch nur das Lesen eines vorgegebenen Datenbestands. Um als Highscore-Verwaltung brauchbar zu werden, müssen wir auch eine Änderung und die Persistierung des Datenbestands vorsehen. Das ist Thema der kommenden Iteration 1.

7.2 Iteration 1: Zentrale Verwaltung von Highscores

In dieser Iteration erstellen wir eine Klasse `HighscoreManager`, die sich um die Verwaltung von `Highscore`-Einträgen kümmern soll. Zudem ist es das Ziel, eine Persistierung mithilfe von Serialisierung zu realisieren – für Spieler unseres Spiels wäre es wohl kaum akzeptabel, wenn deren errungene Highscores nicht gespeichert würden. Schauen wir uns die dazu notwendigen Schritte an.

Verwaltung von `Highscore`-Einträgen

Bisher wurde die Liste der Highscores nur eingelesen, aber nicht zentral verwaltet. Als erste Abhilfe implementieren wir die Klasse `HighscoreManager`, die eine Liste von `Highscore`-Objekten als Attribut besitzt und zunächst nur das Hinzufügen von Highscores sowie den Import aus einer CSV-Datei anbietet. Diese Funktionalitäten realisieren wir in einer Variante 1 folgendermaßen:

```
public class HighscoreManagerV1
{
    private final List<Highscore> highscores = new ArrayList<>();

    public List<Highscore> getHighscores()
    {
        // Kopie erzeugen: keine internen Daten herausgeben
        return new ArrayList<>(highscores);
    }

    public void add(final Highscore highscore)
    {
        highscores.add(highscore);
    }

    public void populateFromCsv(final String filePath)
    {
        final List<Highscore> readHighscores = readHighscoresFromCsv(filePath);

        highscores.clear();
        highscores.addAll(readHighscores);
    }
}
```

Michael Inden, Der Java-Profi: Persistenzlösungen und REST-Services, dpunkt.verlag, ISBN 978-3-86490-374-8

Beispielapplikation

Unser Hauptprogramm zum Starten ändert sich nur minimal, wird aber durch den Einsatz und die Abstraktion des `HighscoreManagers` klarer. Zudem lassen sich nun neue Highscores hinzufügen. Das probieren wir gleich aus, nachdem wir einen initialen Datenbestand aus der CSV-Datei importiert haben:

```
public static final void main(final String[] args)
{
    final HighscoreManagerV1 manager = new HighscoreManagerV1();

    manager.populateFromCsv("resources/Highscores.csv");
    manager.add(new Highscore("Michael", 7271, 10, LocalDate.of(2016, 2, 7)));
    manager.add(new Highscore("Werner", 1000,  8, LocalDate.of(2016, 1, 31)));

    final List<Highscore> highscores = manager.getHighscores();
    highscores.forEach(System.out::println);
}
```

Listing 7.2 *Ausführbar als* **'HIGHSCORELISTEXAMPLE'**

Startet man das obige Programm HIGHSCORELISTEXAMPLE, so wird die CSV-Datei eingelesen und zwei `Highscore`-Objekte erzeugt. Zusätzlich zu den drei aus der CSV-Datei importierten Einträgen werden noch die beiden per `add()` hinzugefügten Highscores dargestellt:

```
Highscore [name=Matze, points=1000, level=7, date=2012-12-12]
Highscore [name=ÄÖÜßöäü, points=777, level=5, date=2010-10-10]
Highscore [name=Peter, points=985, level=6, date=2011-11-11]
Highscore [name=Michael, points=7271, level=10, date=2016-02-07]
Highscore [name=Werner, points=1000, level=8, date=2016-01-31]
```

Wenn man die Ausgabe genauer anschaut, bemerkt man, dass eine Sortierung, wie man sie für eine Liste von Highscores intuitiv erwarten würde, noch fehlt: Die Sortierung soll nach Punktestand und Datum erfolgen – auch soll das Level eine Rolle spielen. Höhere Punktestände sollen zuoberst eingeordnet werden. Beim Level wird ein höheres weiter oben angesiedelt. Beim Datum werden ältere bevorzugt, weil der Highscore dann schon länger besteht. Das wollen wir nun implementieren. Dabei hilft uns, dass die obigen Testdaten (abgesehen vom Datum) so gewählt sind, dass wir die Sortierung leicht prüfen können.

Verbesserungen im `HighscoreManager`

Es gibt verschiedene Varianten, wo wir die Sortierfunktionalität ansiedeln können: Zunächst einmal wäre das die Klasse `Highscore` selbst. Man könnte diese das Interface `Comparable<Highscore>` erfüllen lassen, womit eine natürliche Ordnung beschrieben würde. Doch wie soll diese aussehen? Eine Variante besteht in der Definition von `Comparator<Highscore>`-Instanzen in der Klasse `HighscoreManager`. Das scheint hier näher am zu lösenden Problem zu sein, weil hier eine Menge von Komparatoren definiert und kombiniert werden sollen.

Die Teilschritte einer Sortierung nach Punkten, Datum und Level sowie deren Kombination kann man sehr einfach mithilfe von `Comparator<Highscore>` modellieren. Wir greifen hier auf die vielfältigen Neuerungen aus Java 8 zurück, insbesondere die Methode `Comparator.comparing()`, die einen einsatzfähigen Komparator liefert. Auch nutzen wir die beiden Methoden `thenComparing()` zur Hintereinanderschaltung und `reversed()` zur Umkehrung der Reihenfolge. Mit diesem Wissen ergänzen wir die Sortierfunktionalität im `HighscoreManager` folgendermaßen:

```
public class HighscoreManager
{
    final Comparator<Highscore> byPoints = comparing(Highscore::getPoints);
    final Comparator<Highscore> byDay = comparing(Highscore::getDay);
    final Comparator<Highscore> byLevel = comparing(Highscore::getLevel);

    final Comparator<Highscore> byPointsLevelAndDay =
                                         byPoints.reversed().
                                         thenComparing(byLevel.reversed()).
                                         thenComparing(byDay);

    private final List<Highscore> highscores = new ArrayList<>();

    public List<Highscore> getHighscores()
    {
        // Kopie erstellen: keine internen Daten herausgeben
        final List<Highscore> sortedResults = new ArrayList<>(highscores);

        sortedResults.sort(byPointsLevelAndDay);
        return sortedResults;
    }

    // ...
}
```

Mit diesen im Listing fett markierten Erweiterungen werden die Einträge wunschgemäß sortiert. Das zeigt ein Start des Programms HIGHSCORELISTIMPROVEDEXAMPLE und vor allem dessen Konsolenausgabe:

```
Highscore [name=Michael, points=7271, level=10, date=2016-02-07]
Highscore [name=Werner, points=1000, level=8, date=2016-01-31]
Highscore [name=Matze, points=1000, level=7, date=2012-12-12]
Highscore [name=Peter, points=985, level=6, date=2011-11-11]
Highscore [name=ÄÖÜßöäü, points=777, level=5, date=2010-10-10]
```

Persistenz-Komponente

Die Highscore-Verwaltung besitzt nun mit dem `HighscoreManager` einen zentralen Zugriffspunkt. Eine nutzende Applikation könnte bereits Highscores hinzufügen, jedoch fehlt noch eine Speicherung und das Einlesen zuvor gespeicherter Highscores.

Im Rahmen dieser ersten Iteration wollen wir uns zur Persistierung auf den in Java integrierten Serialisierungsmechanismus stützen. Dazu muss die Klasse `Highscore` das Interface `Serializable` implementieren:

```
public final class Highscore implements Serializable
```

Michael Inden, Der Java-Profi: Persistenzlösungen und REST-Services, dpunkt.verlag, ISBN 978-3-86490-374-8

Dann kann die eigentliche Interaktion mit dem Dateisystem basierend auf Standards aus dem JDK geschehen. Wir erstellen folgende Klasse `HighscorePersister`, um die Funktionalität in einer Klasse sauber zu kapseln:

```
public class HighscorePersister
{
    public void persist(final OutputStream os,
                        final List<Highscore> highscores) throws IOException
    {
        final ObjectOutputStream outStream = new ObjectOutputStream(os);
        outStream.writeObject(highscores);
    }

    public List<Highscore> readFrom(final InputStream is) throws IOException
    {
        final ObjectInputStream inStream = new ObjectInputStream(is);
        try
        {
            return (List<Highscore>) inStream.readObject();
        }
        catch (ClassNotFoundException e)
        {
            // hier nicht möglich, weil alle Klassen unter unserer Kontrolle
        }
        return Collections.emptyList();
    }
}
```

Die Klasse `HighscoreManager` ergänzen wir dann um die Methoden `saveTo()` und `loadFrom()`, die wir folgendermaßen realisieren:

```
public class HighscoreManager
{
    // ...

    public void saveTo(final OutputStream os) throws IOException
    {
        final HighscorePersister persister = new HighscorePersister();
        persister.persist(os, highscores);
    }

    public void loadFrom(final InputStream is) throws IOException
    {
        final HighscorePersister persister = new HighscorePersister();
        final List<Highscore> readHighscores = persister.readFrom(is);

        highscores.clear();
        highscores.addAll(readHighscores);
    }
}
```

Beispielapplikation

Die gerade realisierte Funktionalität zur Persistenz wollen wir prüfen und schreiben dazu eine `main()`-Methode. Dort erzeugen wir zwei `Highscore`-Einträge und speichern diese in der Datei `Highscores.ser`. Danach werden die Daten aus der bereits bekannten CSV-Datei gelesen und zum Schluss die `Highscore`-Objekte aus der zuvor gespeicherten Datei `Highscores.ser` rekonstruiert:

```
public static final void main(final String[] args) throws IOException
{
    final HighscoreManager manager = new HighscoreManager();

    manager.add(new Highscore("Michael", 7271, 10, LocalDate.of(2016, 2, 7)));
    manager.add(new Highscore("Werner",  1000,  8, LocalDate.of(2016, 1, 31)));

    performActionOnFile(manager, "resources/Highscores.ser");
}

private static void performActionOnFile(final HighscoreManager manager,
                                        final String fileName)
                                        throws IOException
{
    try (final OutputStream os = new FileOutputStream(fileName);
         final InputStream is = new FileInputStream(fileName))
    {
        System.out.println("After save to file: " + fileName);
        manager.saveTo(os);
        manager.getHighscores().forEach(System.out::println);

        System.out.println("\nAfter load from csv: ");
        manager.populateFromCsv( "resources/Highscores.csv");
        manager.getHighscores().forEach(System.out::println);

        System.out.println("\nAfter load from file: " + fileName);
        manager.loadFrom(is);
        manager.getHighscores().forEach(System.out::println);
    }
}
```

Listing 7.3 *Ausführbar als* '**HIGHSCOREPERSISTEXAMPLE**'

Starten wir das Programm HIGHSCOREPERSISTEXAMPLE, so sehen wir, dass die beschriebenen Aktionen ausgeführt werden. Die Auswirkungen sind anhand der folgenden Konsolenausgaben gut nachvollziehbar:

```
After save to file: Highscores.ser
Highscore [name=Michael, points=7271, level=10, date=2016-02-07]
Highscore [name=Werner, points=1000, level=8, date=2016-01-31]

After load from csv:
Highscore [name=Matze, points=1000, level=7, date=2012-12-12]
Highscore [name=Peter, points=985, level=6, date=2011-11-11]
Highscore [name=ÄÖÜßöäü, points=777, level=5, date=2010-10-10]

After load from file: Highscores.ser
Highscore [name=Michael, points=7271, level=10, date=2016-02-07]
Highscore [name=Werner, points=1000, level=8, date=2016-01-31]
```

Michael Inden, Der Java-Profi: Persistenzlösungen und REST-Services, dpunkt.verlag, ISBN 978-3-86490-374-8

Design

Das Programm ist ein klein wenig umfangreicher und komplexer geworden. Für die nutzende Applikationsklasse werden Details aber sehr gut durch die Klasse `HighscoreManager` versteckt. Insgesamt ergibt sich folgendes UML-Klassendiagramm, in dem die neuen Klassen grau hinterlegt sind – diese Kennzeichnung nutze ich auch in den folgenden Diagrammen.

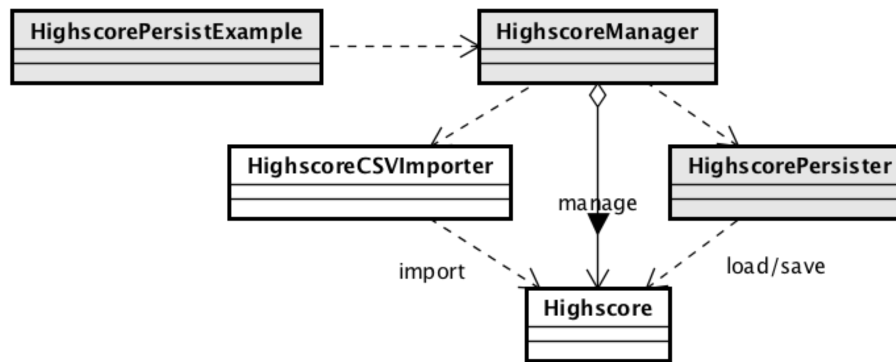

Abbildung 7-2 *Iteration 1 der Beispielapplikation*

Fazit

Die wichtigsten Funktionalitäten zur lokalen Verwaltung einer Highscore-Liste haben wir mittlerweile umgesetzt. Möglicherweise kommt aber der Wunsch auf, die Daten in einem portablen Format wie XML exportieren zu können. Ideal wäre, wenn sich dies ohne größere Änderungen in den `HighscoreManager` integrieren ließe.

7.3 Iteration 2: Verwaltung von XML

Wie gerade motiviert, soll nun mit XML ein portables Format zum Datenaustausch bereitgestellt werden. Die Funktionalität des Ex- und Imports von XML wollen wir in das bisherige Design integrieren. Die Persistierung erfolgt bislang durch die Klasse `HighscorePersister`. Somit scheint diese ein geeigneter Kandidat zu sein, dort die Erweiterung vorzunehmen. Natürlich könnte man die unterschiedlichen Ausprägungen durch Fallunterscheidungen mit `if`-Anweisungen realisieren. Das wird aber recht schnell unübersichtlich und schlecht erweiterbar. Funktionalität in Objekten bereitzustellen ist unter anderem Bestandteil des Entwurfsmusters STRATEGY. Dieses beschreibe ich ausführlich in meinem Buch »Der Weg zum Java-Profi« [8].

Strukturelle Vorarbeiten

Um die Varianten des Ex- und Imports im restlichen Programm nicht unterscheiden zu müssen, extrahieren wir ein gemeinsames Interface `IPersistStrategy` aus der ursprünglich zur Persistierung genutzten Klasse `HighscorePersister`:

```
public interface IPersistStrategy
{
   void persist(OutputStream os, List<Highscore> highscores) throws IOException;
   List<Highscore> readFrom(InputStream is) throws IOException;
}
```

Zudem implementieren wir eine Strategie zur Persistierung auf Basis von Serialisierung sowie in Form von XML. Dazu definieren wir folgende Aufzählung `PersistMode`:

```
public enum PersistMode
{
    SERIALIZATION
    {
        IPersistStrategy getStrategy()
        {
          return new PersistWithSerializationStrategy();
        }
    },
    XML
    {
        IPersistStrategy getStrategy()
        {
          return new PersistWithXmlStrategy();
        }
    };


    abstract IPersistStrategy getStrategy();
}
```

Außerdem benennen wir die in Iteration 1 entwickelte Klasse `HighscorePersister` in `PersistWithSerializationStrategy` um und lassen diese das zuvor gezeigte Interface wie folgt implementieren – die beiden Methoden sind unverändert:

```
public class PersistWithSerializationStrategy implements IPersistStrategy
{
    @Override
    public void persist(final OutputStream os,
                        final List<Highscore> highscores)
                        throws IOException
    {
        final ObjectOutputStream outStream = new ObjectOutputStream(os);
        outStream.writeObject(highscores);
    }

    @Override
    public List<Highscore> readFrom(final InputStream is) throws IOException
    {
        final ObjectInputStream inStream = new ObjectInputStream(is);
        try
        {
            return (List<Highscore>) inStream.readObject();
        }
        catch (ClassNotFoundException e)
        {
            // kann hier nicht auftreten, weil nur eigene Klassen
        }
        return Collections.emptyList();
    }
}
```

Michael Inden, Der Java-Profi: Persistenzlösungen und REST-Services, dpunkt.verlag, ISBN 978-3-86490-374-8

Schließlich benötigen wir wieder eine Klasse zur zentralen Steuerung der Persistierung. Dabei orientieren wir uns an der Klasse `HighscorePersister`, integrieren jedoch die beiden zu unterstützenden Strategien: In den Methoden zum Laden und Speichern steuert nun ein Parameter, welche Strategie zur Persistierung gewählt wird:

```
public class HighscorePersister
{
    public void persist(final PersistMode mode,
                        final OutputStream os,
                        final List<Highscore> highscores)
                        throws IOException
    {
        final IPersistStrategy strategy = mode.getStrategy();
        strategy.persist(os, highscores);
    }

    public List<Highscore> readFrom(final PersistMode mode,
                                    final InputStream is)
                                    throws IOException
    {
        final IPersistStrategy strategy = mode.getStrategy();
        return strategy.readFrom(is);
    }
}
```

XML-Persistierung

Wir haben mit der Klasse `HighscorePersister` die zentrale, steuernde Komponente zum Ex- und Import kennengelernt. Dort wurde beim Befüllen der Map bereits die `PersistWithXmlStrategy` erzeugt, die der Verarbeitung von XML dient. Wie schon bei der Verarbeitung von XML in Abschnitt 1.3.2 angedeutet, muss man zum Speichern von Listen eine zusätzliche Klasse bereitstellen, damit die JAXB-Automatiken funktionieren. Wir implementieren dazu die Hilfsklasse `HighscoreList` folgendermaßen:

```
@XmlRootElement (name="Highscores")
@XmlAccessorType (XmlAccessType.FIELD)
public class HighscoreList
{
    @XmlElement(name = "Highscore")
    private List<Highscore> highscores;

    public List<Highscore> getHighscores()
    {
        return highscores;
    }

    public void setHighscores(final List<Highscore> highscores)
    {
        this.highscores = highscores;
    }
}
```

Mithilfe dieser Klasse können wir uns dann auf die in JAXB integrierten Automatiken von `Marshaller` und `Unmarshaller` abstützen, um die XML-Verarbeitung wie folgt zu implementieren:

```
public class PersistWithXmlStrategy implements IPersistStrategy
{
    @Override
    public void persist(final OutputStream os,
                        final List<Highscore> highscores)
                        throws IOException
    {
        final HighscoreList highscoreList = new HighscoreList();
        highscoreList.setHighscores(highscores);

        try
        {
            final Marshaller marshaller = createMarshaller();
            marshaller.marshal(highscoreList, os);
        }
        catch (final JAXBException ex)
        {
            throw new IOException(ex);
        }
    }

    @Override
    public List<Highscore> readFrom(final InputStream is) throws IOException
    {
        try
        {
            final Unmarshaller unmarshaller = createUnmarshaller();
            final HighscoreList highscoreList = (HighscoreList)
                                    unmarshaller.unmarshal(is);
            return highscoreList.getHighscores();
        }
        catch (final JAXBException ex)
        {
            throw new IOException(ex);
        }
    }

    private Marshaller createMarshaller() throws JAXBException,
          PropertyException
    {
       final JAXBContext context = JAXBContext.newInstance(HighscoreList.class);
       final Marshaller marshaller = context.createMarshaller();
       marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
       return marshaller;
    }

    private Unmarshaller createUnmarshaller() throws JAXBException
    {
       final JAXBContext context = JAXBContext.newInstance(HighscoreList.class);
       return context.createUnmarshaller();
    }
}
```

Michael Inden, Der Java-Profi: Persistenzlösungen und REST-Services, dpunkt.verlag, ISBN 978-3-86490-374-8

Beispielapplikation

Um die gerade erweiterte Funktionalität zur Persistenz auszuprobieren, sind lediglich minimale Anpassungen bzw. Erweiterungen um Parameter vom Typ `PersistMode` notwendig. Das zieht sich recht weit durch das Programm und betrifft auch die Klasse `HighscoreManager` und deren beide Methoden `saveTo()` und `loadFrom()`:

```
public static final void main(final String[] args) throws IOException
{
    final HighscoreManager manager = new HighscoreManager();
    manager.add(new Highscore("Michael", 7271, 10, LocalDate.of(2016, 2, 7)));
    manager.add(new Highscore("Werner",  1000,  8, LocalDate.of(2016, 1, 31)));

    performActionOnFile(manager, PersistMode.SERIALIZATION,
                                 "resources/Highscores.ser");
    performActionOnFile(manager, PersistMode.XML, "resources/Highscores.xml");
}

private static void performActionOnFile(final HighscoreManager manager,
                                        final PersistMode mode,
                                        final String fileName)
                                        throws IOException
{
    try (final OutputStream os = new FileOutputStream(fileName);
         final InputStream is = new FileInputStream(fileName))
    {
        System.out.println("After save to file: " + fileName);
        manager.saveTo(mode, os);
        manager.getHighscores().forEach(System.out::println);

        System.out.println("\nAfter load from csv: ");
        manager.populateFromCsv( "resources/Highscores.csv");
        manager.getHighscores().forEach(System.out::println);

        System.out.println("\nAfter load from file: " + fileName);
        manager.loadFrom(mode, is);
        manager.getHighscores().forEach(System.out::println);
    }
}
```

Listing 7.4 *Ausführbar als* **'HIGHSCOREPERSISTIMPROVEDEXAMPLE'**

Starten wir das Programm HIGHSCOREPERSISTIMPROVEDEXAMPLE, so wird nun auch eine Datei `Highscores.xml` mit folgendem Inhalt geschrieben:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Highscores>
    <Highscore>
        <day>2016-02-07</day>
        <level>10</level>
        <name>Michael</name>
        <points>7271</points>
    </Highscore>
    <Highscore>
        <day>2016-01-31</day>
        <level>8</level>
        <name>Werner</name>
        <points>1000</points>
    </Highscore>
</Highscores>
```

Design

Unter der Motorhaube hat sich einiges getan. Aber aus Sicht einer nutzenden Applikation bleibt das Design recht stabil – natürlich ändern sich Kleinigkeiten, um die neuen Funktionalitäten ansprechen zu können. Die größten Änderungen erfährt die Klasse `HighscorePersister`, die nun abhängig vom gewünschten Format eine korrespondierende Ex-/Import-Strategie wählt.

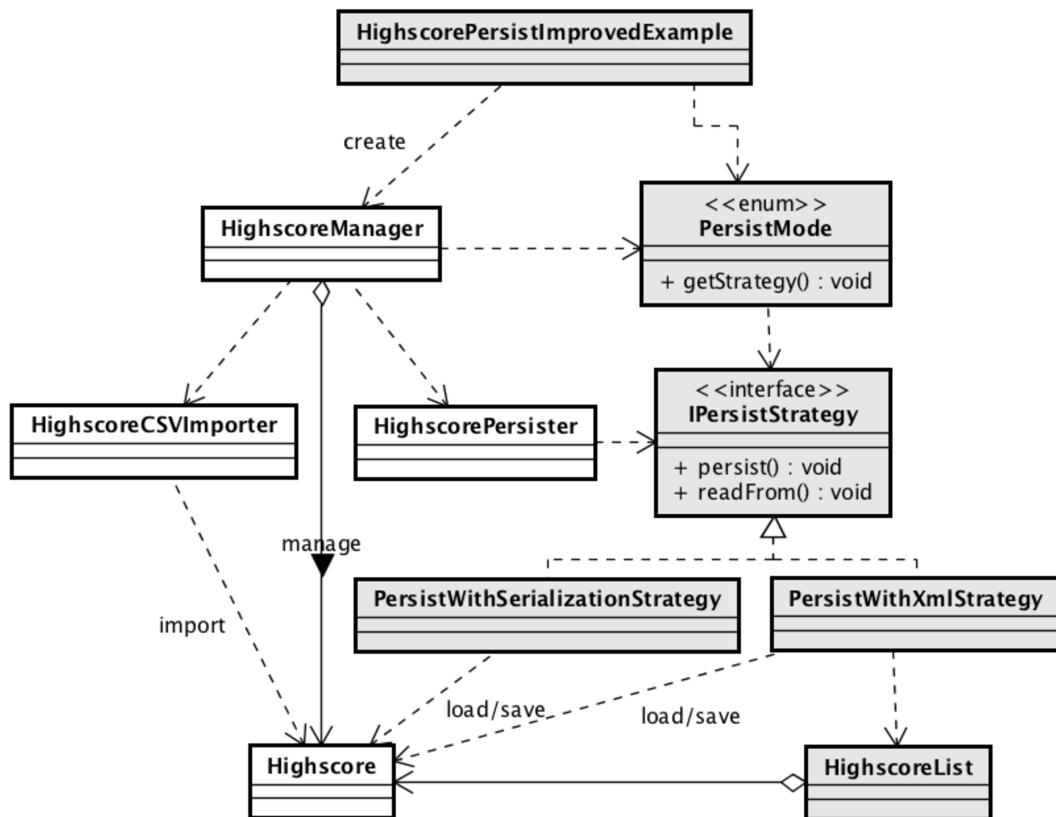

Abbildung 7-3 *Iteration 2 der Beispielapplikation*

Fazit

In nur wenigen Iterationen haben wir aus einer ziemlich einfachen Applikation eine vollwertige Highscore-Verwaltung entwickelt, die für ein einzelnes Spiel ausreichend wäre. Sollen aber Highscores unterschiedlicher Spieler auf verschiedenen Rechnern miteinander abgeglichen werden, so benötigt man eine über das Netzwerk aufrufbare Funktionalität, wozu sich ein REST-Service anbietet.

Michael Inden, Der Java-Profi: Persistenzlösungen und REST-Services, dpunkt.verlag, ISBN 978-3-86490-374-8