Technische Hochschule Ingolstadt

Fakultät für Elektrotechnik und Informatik

Zukunft in Bewegung

Die Prozessicht

Architektur- und Entwurfsmuster der Softwaretechnik

Prof. Dr. Bernd Hafenrichter 02.05.2022

Asynchrone Programmierung

Motivation

- Programme arbeiten i.d.R. sequentiell
- Mögliche Parallelität wird nicht ausgenutzt
- Wie können einzelne Berechnungsschritte parallelisiert werden?

```
String result1 = computationFunction1();
String result2 = computationFunction2();

System.out.println( "Result1: " + result1 );
System.out.println( "Result2: " + result2 );
```

Lösung

- Asynchrone Programmierung

Asynchrone Programmierung

Asynchrone Programmierung

- Programmteile werden asynchron mit Hilfe eines ThreadPool‘s ausgeführt.

```
public interface ThreadPool {

/**
 * hand over a request to the thread pool.
 */
public void submitRequest( WorkItem item );
}
```

```
public interface WorkItem {

/**
 * This method is executed by a worker thread
 */
public void execute();
}
```

Asynchrone Programmierung

Asynchrone Programmierung

- Programmteile werden asynchron mit Hilfe eines ThreadPool‘s ausgeführt.
- Die gesamte Verarbeitung erfolgt asynchron. Auf Ergebnisse der Berechnung kann in einer Folgeverarbeitung nicht zugegriffen werden

```
ThreadPool threadPool = new ThreadPoolImpl(10);

threadPool.submitRequest( ParallelClient::baseFunction );
```

Asynchrone Programmierung

Futures

- Ein Future oder ein Promise bezeichnet in der Programmierung einen Platzhalter für ein Ergebnis, das noch nicht bekannt ist, meist weil seine Berechnung noch nicht abgeschlossen ist.
- Ein Future ist meist das Ergebnis eines asynchronen Aufrufs einer Funktion und kann verwendet werden, um auf das Ergebnis zuzugreifen, sobald es verfügbar ist.
- Diese Art der Programmierung erlaubt eine weitgehend transparente Parallelisierung nebenläufiger Prozesse. Das Konzept der Futures wurde 1977 in einem Artikel von Henry G. Baker und Carl Hewitt vorgestellt.

- https://de.wikipedia.org/wiki/Future_(Programmierung)

Asynchrone Programmierung mit Futures

Beispiel für ein Future-Interface

```
public interface FutureResult<T> {

 /**
  * Returns the result of the computation or
  * waits until the result is available
  */
 public T get() throws InterruptedException;

 /**
  * Returns current state of the computation
  */
 public FutureState getState();
}
```

Asynchrone Programmierung mit Futures

Verwendung eines Futures

```
// Create a first asynchronous computation
FutureResult<String> result1 = FutureFactory.execute( PullClientWithFutureGet::computationFunction1 );

// Create a second asynchronous computation
FutureResult<String> result2 = FutureFactory.execute( PullClientWithFutureGet::computationFunction2 );

// Do some  other tasks in parallel

// Obtain the result from the computation
String theResult1 = result1.get()
String theResult2 = result2.get()
```

Asynchrone Programmierung mit Futures

Futures

- Futures realisieren das „Pull“-Prinzip. D.h. ein Client wartet auf das Ergebnis
- Der Client wird trotzdem blockiert bis das Ergebnis der Berechnung zur Verfügung steht um die Folgeverarbeitung fortzuführen

```
FutureResult<String> futur1 = FutureFactory.execute( PullClientWithFutureGet::computationFunction1 );
FutureResult<String> futur2 = FutureFactory.execute( PullClientWithFutureGet::computationFunction2 );
// Obtain the result from the computation.  Wait for the result if necessary
String theResult1 = futur1.get()
String theResult2 = futur1.get()
```

- Lösung: Push-Prinzip. Wenn das Ergebnis vorliegt stoße die folgende Verarbeitung

Asynchrone Programmierung mit Futures

Definition einer Consumers-Funktion

- Ein Consumer ist eine Funktion welche das Ergebnis einer Future-Berechnung entgegen nimmt und eine Folgeverarbeitung durchführt.
- Die Consumer-Funktion wird asynchron ausgeführt sobald das Ergebnis bereitsteht.
- Das Ergebnis wird in die „Consumer“-Funktion gepusht

```
public interface Consumer<T> {
        /**
         * Execute the consumer function on the value
         * of a preceeding future reslut
         */
        public void execute( T value );
}
```

Asynchrone Programmierung mit Futures

Definition einer Consumers-Funktion

- Ein Consumer ist eine Funktion welche das Ergebnis einer Future-Berechnung entgegen nimmt und eine Folgeverarbeitung durchführt.
- Die Consumer-Funktion wird asynchron ausgeführt sobald das Ergebnis bereitsteht.
- Das Ergebnis wird in die „Consumer“-Funktion gepusht

```
FutureResult<String> future1 = FutureFactory.execute(
    PullClientWithFutureGetAndConsumer::computationFunction1 );

// Führe die consumer Funktion asynchron aus nachem das result zur Verfügung steht
FutureFactory.applyAfter(future1,
                                resultValue -> System.out.println( "Result1: " + resultValue ) );
```

Asynchrone Programmierung mit Futures

Ergebnisse mehrere Futures kombinieren

- Wird das Ergebnis mehrerer asynchroner Berechnung für die Folgeverarbeitung benötigt müssen alle Future-Results bereitgestellt werden bevor die Berechnung beginnen kann.
- Alle Ergebnisse werden in den Consumer „gepusht“.
- Die Verarbeitung erfolgt asynchron wenn alle Ergebnisse vorliegen

```
public interface BiConsumer<T,R> {
  /**
   * Consume the result of the generated elements from different futures
   */
  public void execute( T valuea, R valueb );
  }
}
```

Asynchrone Programmierung mit Futures

Ergebnisse mehrere Futures kombinieren

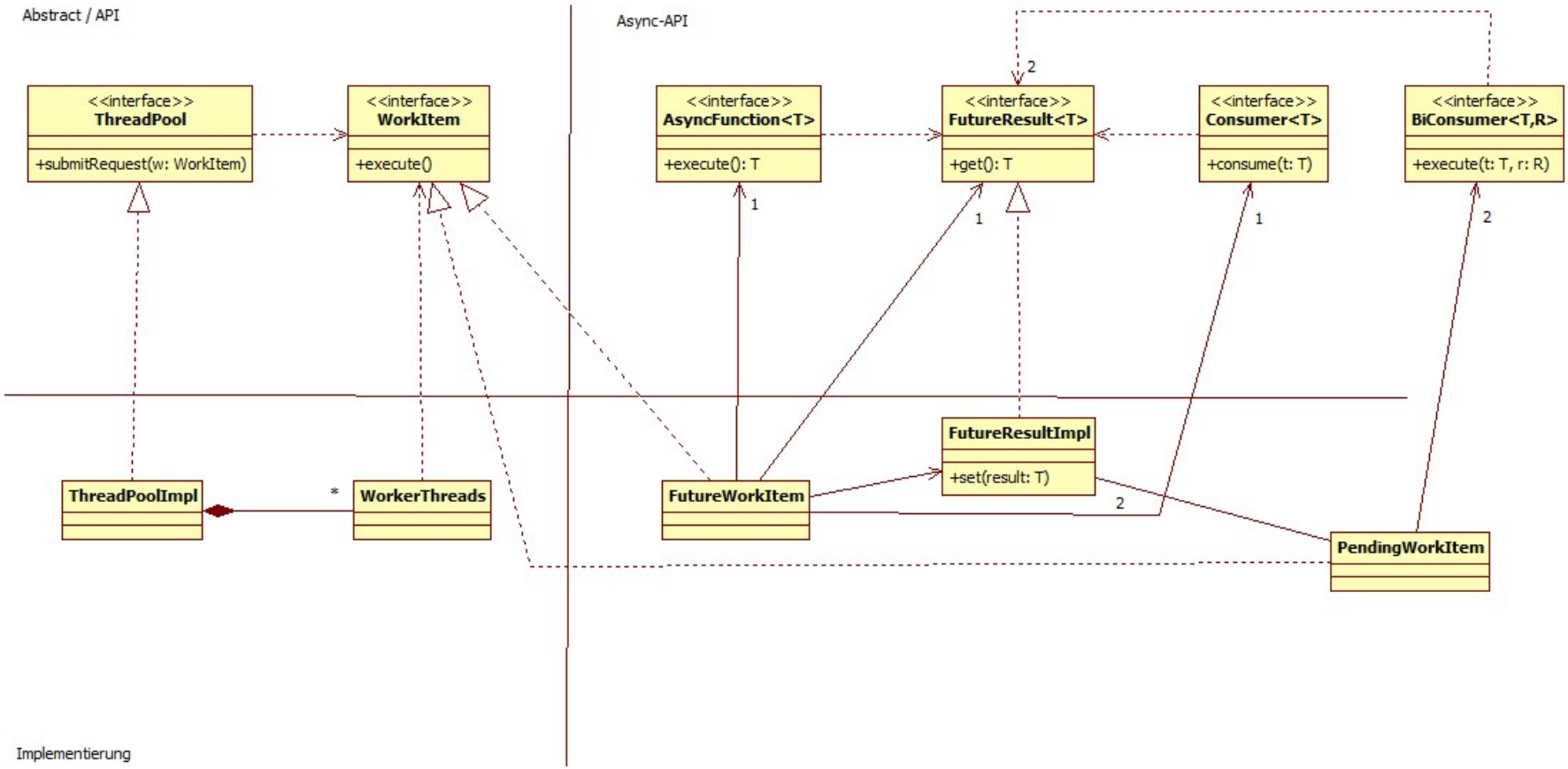
- Wird das Ergebnis mehrerer asynchroner Berechnung für die Folgeverarbeitung benötigt müssen alle Future-Results bereitgestellt werden bevor die Berechnung beginnen kann.

```
FutureResult<String> result1 = FutureFactory.execute(PushClientWithCombine::computationFunction1 );
FutureFactory.applyAfter( result1, e -> System.out.println( "Result1: " + e ) );

FutureResult<String> result2 = FutureFactory.execute( PushClientWithCombine::computationFunction2 );
FutureFactory.applyAfter( result2, e -> System.out.println( "Result2: " + e ) );

FutureFactory.combine( result1, result2, PushClientWithCombine::combinerFunction );
```

Asynchrone Programmierung mit Futures



Asynchrone Programmierung mit Futures

Ausblick – Verwendung von Fluent-Interfaces

- Die asynchrone Programmierung wirkt unnatürlich
- Wie kann man diese Art der Programmierung möglichst elegant in eine Programmiersprache einbetten?
- Lösung: FluentApi-Desgn (Die API liest sich wie natürliche Sprache)

```
FutureResult<String> result1 = FutureFactory.execute( FluentApiPushClientWithCombine::computationFunction1 )
                                        .thenApply( e -> System.out.println( "Result1: " + e ) );

FutureFactory.execute(FluentApiPushClientWithCombine::computationFunction2 )
                                        .thenApply( e -> System.out.println( "Result2: " + e ) )
                                            .combine( result1, (a,b) -> combinerFunction(a,b,startTime) );
```

Asynchrone Programmierung mit Futures

Asynchrone Programmierung moderenen Programmiersprachen

- Java:
  - https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html
  - https://github.com/electronicarts/ea-async
  - https://dzone.com/articles/async-await-in-java
- C# :
  - https://docs.microsoft.com/de-de/dotnet/csharp/programming-guide/concepts/async/