ESPRITSYSTEMS

Programmierung von asynchronen Tasks

Ausführung, Kontrolle und Verwaltung asynchroner Tasks mit Hilfe des *EsprIT* AsyncTask-Frameworks

November 2011
Rainer Büsch

Inhaltsverzeichnis

1 Einführung....3
1.1 Java Concurrency Framework....3
1.2 Erweiterungen in EsprIT....3
2 Einfaches AsyncTask Beispiel....4
2.1 Task Definition....4
2.2 Task erzeugen und starten....5
2.3 Task Monitoring....5
2.4 Task Monitoring im GUI....6
3 Task Synchronisation....6
3.1 Zustands-Gates....6
3.2 Das RunStateFlag....7
3.3 Task Zustandswechsel....8
4 Abhängige Tasks....8
4.1 Child-Tasks (LocalTask)....8
4.2 Sub-Tasks (AsyncExcecutable)....8
4.3 Progress von Child- und Sub-Tasks....9
4.4 Task-Collections....9
4.5 Ausführung von AsyncExecutables....9
4.6 Child-Tasks innerhalb von AsyncExcecutables....10
5 Weitere Eigenschaften von Tasks....11
5.1 Serialisierbarkeit....11
5.2 Wiederverwendbarkeit....11
5.3 Benamung und Typisierung....11
5.4 Abbrechbarkeit....12
5.5 Vorzeitiges Beenden eines Task....12
5.6 Threadpool-Zuordnung....12
5.7 Task Priorität....12
6 Verwaltung von Tasks....12
6.1 Task-Registrierung....12
6.2 Abbrechen von Tasks....13
6.3 Besondere Task Typen....13
    6.3.1 AbstractSwingTask....13
    6.3.2 AbstractOsProcessTask....13
    6.3.3 JavaCommandTask....14
7 Workflows....14
7.1 Workflow Task....14
7.2 Decider Task....15
7.3 Workflow Manager....16
8 Remote Tasks....17
8.1 Remote Tasks auf Server....17
8.2 Remote Task auf CoServer....19

1 Einführung

In den meisten Applikationen gibt es Aufgaben (z.B. Einlesen von Daten), die längere Zeit dauern und daher typischerweise als eigenständig parallel laufende Threads implementiert werden. Multithreading in Java ist im Prinzip einfach. Deutlich schwieriger aber ist es, nebenläufige Aufgaben (Tasks) zu verwalten, sie zu beobachten (Monitoring), aufeinander zu synchronisieren oder jederzeit kontrolliert abzubrechen.

Das *EsprIT-AsyncTask-Framework* wurde entwickelt, um diese Funktionalitäten auf möglichst einfache Art bereitzustellen. Das vorliegende Dokument beschreibt das Konzept dieses Frameworks. Es behandelt die wesentlichen Elemente seiner Implementierung und demonstriert die Anwendung an einfachen Beispielen. Die folgenden Ausführungen erheben keinen Anspruch auf Vollständigkeit. Vielmehr geht es darum, dem Leser ein Grundverständnis für die Funktionsweise zu vermitteln und eine Übersicht über die Möglichkeiten zu geben.

➔ Zum Verständnis des Folgenden sind Grundkenntnisse über das Java Concurreny-Framework sowie das EsprIT Client/Server-Framework erforderlich.

1.1 Java Concurrency Framework

Java beinhaltet seit der Version 1.5 standardmäßig eine Reihe von Supportklassen zur Ausführung asynchroner Tasks. Die Grundidee dieses Frameworks ist es, asynchronen Code - in *Runnable* oder *Callable* Objekte gehüllt - an einen *Executor* zur Ausführung zu übergeben (als „*Task*“). Der *Executor* ist dabei in der Regel ein Threadpool, in dem je nach Konfiguration die einmal instantiierten Threads wiederverwendet werden. Beim Einstellen eines Tasks in den Threadpool erhält der Anwender ein *Future* Objekt zurück und hat damit weiterhin eine gewisse Kontrolle über den asynchronen Ablauf. Insbesondere ist ein Task durch den *cancel()* Aufruf jederzeit abbrechbar.

Das Executor Framework sowie die Java Concurrency Klassen bilden eine gute Basis, um asynchrone Abläufe robust zu programmieren. Allerdings ist es trotzdem eine große Herausforderung, diese Technik in einem Swing GUI zu verwenden, denn es fehlt an Möglichkeiten...

➔ laufende Tasks bei der Ausführung zu beobachten und auf deren Zustandswechsel gezielt und korrekt synchronisiert zu reagieren.

➔ auf das Abbrechen oder auf Ablauffehler eines Tasks zu reagieren, z.B. ein “Aufräumen” im Abbruch- oder Fehlerfalle einzubauen.

➔ viele gleichzeitig asynchron laufende Tasks zu verwalten und ggf. Einzelne gezielt abzubrechen.

➔ mehrere Tasks in einer Serie (als sog. “Workflow”) ablaufen zu lassen.

➔ mehrere Tasks gekoppelt (z.B. Tasks als Unter-Tasks von anderen) ablaufen zu lassen.

➔ asynchrone und GUI-synchrone Vorgänge zu koppeln[1]. Die Standard-Java Klasse *SwingWorker* leistet hier nur sehr rudimentäre Dienste.

➔ auf anderen Rechnern laufende Tasks zu beobachten, sich (über Netz) auf diese zu synchronisieren und sie zu kontrollieren.

1.2 Erweiterungen in EsprIT

Das *EsprIT AsyncTask-Framework* bietet Lösungen für die genannte Thematik. Es baut auf dem

1 Ein Task darf keine Methoden in Swing-GUI-Komponenten aufrufen! Dies darf gemäß dem Single-Thread Modell von Swing nur der *EventDispatchingThread*, der das GUI steuert.

Java Concurrency Framework auf und erweitert es im Wesentlichen um folgende Funktionalitäten:

- ➔ EsprIT stellt im *ApplicationContext* einen *TaskManager* zur Verfügung. Dieser beinhaltet einen Standard-Threadpool sowie einen Mechanismus zur Verwaltung und Kontrolle aller parallel laufenden Tasks.
- ➔ Laufende Tasks können anhand einer *TaskId* identifiziert und jederzeit abgebrochen werden, unabhängig davon, in welchem Threadpool sie laufen.
- ➔ Tasks können als eine *Collection* vieler einzelner asynchroner Aufgaben gruppiert werden.
- ➔ Tasks können mit *TaskMonitoren* versehen werden, die ihre asynchrone Ausführung beobachtbar machen. Ein Monitor wird über die wechselnden Task-Zustände informiert und kann darauf spezifisch reagieren.
- ➔ Über einen speziellen *SwingTaskMonitor* können GUI-synchrone Aktionen vom asynchron laufenden Task gesteuert werden. Hierauf basieren die sog. *Workflows,* in der eine Reihe von asynchronen Tasks – im GUI beobachtbar und mit GUI-Interaktion – abgearbeitet werden.
- ➔ Tasks können andere Tasks als Child-Tasks ausführen, in beliebiger Schachtelungstiefe. Dadurch sind voneinander abhängige asynchrone Vorgänge zentral steuerbar.
- ➔ Ein Client kann remote[2] Tasks auf dem Server starten. Der Client als Eigentümer ist an den remoten Task „gebunden“ und kann synchron auf seine Zustandwechsel reagieren. Dies funktioniert auch für remote Tasks, die „über den Server hinweg“ auf einem CoServer laufen.

2 Einfaches AsyncTask Beispiel

2.1 Task Definition

Das folgende Beispiel zeigt den grundsätzlichen Umgang mit Tasks im EsprIT-Framework. Ein Task ist eine Klasse, die von *AbstractLocalTask* ableitet und dadurch das *LocalTask* Interface bereits implementiert hat. Lediglich die noch fehlende *executeAsync()* Methode ist zu überschreiben. Sie beinhaltet den Code, der asynchron ausgeführt werden soll:

```
public class ExCalculationTask extends AbstractLocalTask {

     private int loops = 50;

     public ExCalculationTask(ApplicationContext ctx) {
            super(ctx);
     }

     @Override @Async
     public void executeAsync() throws Exception {
            int sum = 0;
            for (int i = 0; i < loops; i++) {
                   sum += i;
                   notifyProceeded(sum, i, loops, "Task step: "+i);
                   sleepChecked(50);
            }

            setResult(sum);
     }
}
```

Dieser Task bildet in einer Schleife eine einfache Summe. Bei jedem Schleifendurchlauf wird *notifyProceeded(...)* aufgerufen wodurch ein ggf. eingebundener Monitor über den Fortschritt benachrichtigt würde. Der *sleepChecked(long)* Aufruf verlangsamt den Task künstlich zugunsten

2 Diese Funktion steht nur in der *EsprIT-ServerSuite* zur Verfügung.

eventuell parallel laufender anderer Tasks („freundliches“ Verhalten). Ohne diesen Aufruf würde der Task die CPU stark belasten und damit auch das Hauptprogramm in Mitleidenschaft ziehen. Darüberhinaus prüft dieser Aufruf ob der Anwender mittlerweile versucht hat, den Task zu canceln. Falls ja, würde an dieser Stelle eine *CancelledException* geworfen und die Ausführung abgebrochen. Das Endergebnis wird mit *setResult(Object)* im Task selbst gespeichert.

2.2 Task erzeugen und starten

Wir wollen diesen Task erzeugen und in einem Threadpool laufen lassen:

```
ApplicationContext ctx = Util.getDefaultContext();
ExCalculationTask task = new ExCalculationTask(ctx);
TaskId taskId = task.submit();  // starts running asynchroneously
Util.sleep(100);                // let it run for a 100 millis
ctx.cancelTask(taskId);         // and now cancel it
```

Mit dem *submit()* Aufruf wird der Task in den Standard-Threadpool des *ApplicationContexts* eingestellt. Alternativ könnte er mit *submit(ExecutorService)* auch in einem beliebigen anderen Pool laufen. Als Ergebnis des Einstellens in den Pool erhält man eine *TaskId*, die den Task eindeutig identifiziert und anhand derer man den asynchronen Ablauf, wie gezeigt, jederzeit abbrechen kann. Ein *submit()* Auftruf kann fehlschlagen, wenn die Resourcen des Threadpools erschöpft sind.

2.3 Task Monitoring

Als Nächstes wollen wir den Task bei seinem Ablauf beobachten. Dazu wird im Constructor ein *TaskMonitor* eingebunden, der während des Ablaufs über die sich ändernden Task-Zustände informiert wird:

```
    public ExCalculationTask(ApplicationContext ctx) {
           super(ctx);
           addMonitor(new DebugTaskMontior(ctx));
    }
```

Lassen wir diesen Task laufen, dann sehen wir folgenden Output auf der Systemkonsole[3]:

```
INF: STARTED, tid=1, name=ExCalculationTask
INF: PROGRESS, step=0, maxSteps=50, msg="Task step: 0", progRes=1 // Zwischenergebnis 1
INF: PROGRESS, step=1, maxSteps=50, msg="Task step: 1", progRes=3 // Zwischenergebnis 2
...
INF: SUCCEEDED, result=1225
INF: FINISHED, tid=1, name=ExCalculationTask
```

Ein laufender Task benachrichtigt den Task-Monitor indem er bei jedem Zustandswechsel in ihm die entsprechende Methode aufruft (z.B. *taskStarted()* für den STARTED Zustand, etc.). Der Hauptrahmen (STARTED, SUCCEEDED, FINISHED) wird automatisch generiert. Die PROGRESS Meldungen haben wir durch den Aufruf *notifyProceeded(...)* selbst hineinprogrammiert. Die Zustände STARTED und FINISHED erscheinen garantiert immer. Anstelle von SUCCEEDED könnte allerdings auch CANCELLED stehen, wenn der Ablauf abgebrochen wurde bzw. FAILED, wenn von der *executeAsync()* Methode eine Exception geworfen wurde.

Es können beliebig viele Monitore an einen Task eingehängt werden. Sie werden in der Reihenfolge, wie sie hinzugefügt wurden, informiert.

3 Die Ausgabe wurde auf das Wesentliche reduziert

➔ Man beachte, dass die Methoden des Task-Monitors von dem Poolthread aufgerufen werden, der den Task ausführt. Innerhalb dieser Methoden dürfen also keine Aufrufe an GUI-Komponenten programmiert sein, das würde dem Single-Thread Model von Swing-GUIs zuwiderlaufen und könnte zu ernsten Problemen führen.

2.4 Task Monitoring im GUI

Um in einem GUI die Task-Fortschritte zu visualisieren, müssen die asynchronen Zustands-Benachrichtigungen in GUI-synchrone Aufrufe umgewandelt werden. Genau dies leistet der *SwingTaskMonitor*, der entsprechende *TaskEvents* erzeugt, für die man sich als *Listener* registieren kann. Noch einfacher ist es, von der Klasse ***AbstractSwingTask*** abzuleiten. Diese besitzt einen solchen Monitor und ist bereits selbst Listener der eigenen *TaskEvents*. Man braucht dann nur noch die Methoden *doneStarted()*, *doneProceeded(...)*, etc. zu überschreiben, um GUI-synchrone Aktionen einzubinden.

```
public class ExCalculationTask extends AbstractSwingTask {
     ...
     @Override @Sync
     public void doneProceeded(TaskEvent e) {
          progressBar.setValue(e.getProgressPercentage()); // drive ProgressBar
     }
     @Override @Sync
     public void doneSucceeded(TaskEvent e) {
          showInfo("Task succeeded with result: "+e.getResult()); // popup dialog
     }
}
```

Eine GUI-Fehlerbehandlung ist im *SwingTaskMonitor* bereits standardmäßig eingebaut.

3 Task Synchronisation

Oft ist es vom Hauptprogramm aus notwendig, auf das Ergebnis eines nebenläufigen Tasks zu warten. Dazu dient der Aufruf von *task.awaitResult()*, der so lange blockiert, bis das Ergebnis vorliegt. Das zu erwartende Ergebnis kann der Task selbst sich mit *setResult(Object)* setzen. Wurde kein Ergebnis gesetzt, wird *null* zurückgegeben. Das folgende Beispiel zeigt, wie auf das Ergebnis eines Tasks gewartet wird:

```
ExCalculationTask task = new ExCalculationTask(ctx);
TaskId taskId = task.submit();  // starts running asynchroneously
int resultValue = task.awaitResult(5, TimeUnit.SECONDS); // limit the wait time
```

Der Rückgabewert von *awaitResult()* ist generisch und wird automatisch auf den links definierten Typ gecastet. Wurde das angegebene Timeout überschitten, dann wird der Task automatisch abgebrochen.

3.1 Zustands-Gates

Ein Task beinhaltet drei besondere *Gates* zur Zustandssynchronisation:

- **Release-Gate**
  Steuert das Loslaufen des Tasks nach dem Einstellen in den Threadpool. Defaultmäßig ist dieses Gate offen und der Task beginnt sofort zu laufen. Mit *setAutoReleaseOnStart(false)* ist das Gate zu Beginn geschlossen und der Task blockiert bis *releaseStart()* explizit aufgerufen wird. Damit hat man es auch nach dem Einstellen in den Pool noch unter Kontrolle, wann genau der Task wirklich losläuft.
- **Started-Gate**
  Dieses Gate ist immer zu Beginn geschlossen und wird erst geöffnet, nachdem der Task wirklich losgelaufen ist. Auf diesen Zeitpunkt kann man mit *awaitStarted()* gezielt warten.
- **Finished-Gate**
  Dieses Gate ist geschlossen bis der Task vollständig zu Ende gelaufen ist. Mit *awaitFinished()* kann man sich gezielt auf diesen Zeitpunkt synchronisieren. Man beachte dass *awaitFinished()* im Unterschied zu *awaitResult()* nicht unterbrechbar ist. Wird ein Task abgebrochen, dann läßt *awaitFinished()* erst dann los, wenn der Task selbst seine Reaktion auf den Abbruch (*doOnCancel()*) abgearbeitet hat; *awaitResult()* hingegen läßt sofort beim Abbruch los und gibt *null* zurück.

3.2 Das RunStateFlag

Das *RunStateFlag* bildet die verschiedenen Task-Zustände ab und stellt die zentrale Synchronisierungskomponente eines Tasks dar. Die möglichen Zustände sind:

INITIALIZED	Der Task wurde frisch erzeugt und ist noch nie gelaufen
STARTED	Der Task ist losgelaufen
CHILD_STARTED	Ein Child-Task wurde gestartet
PROGRESS	Ein Task (oder Child-Task) hat einen Fortschritt erzielt
CHILD_SUCCEEDED[4]	Ein Child-Task ist erfolgreich durchgelaufen
CHILD_FINISHED	Ein Child-Task wurde beendet
CANCELLED	Der Task (oder Child-Task) wurde abgebrochen
FAILED	Der Task (oder Child-Task) ist fehlgeschlagen
SUCCEEDED	Der Task ist erfolgreich durchgelaufen
FINISHED	Der Task ist zu Ende gelaufen

Mit *getRunStateFlag()* erhält man vom Task eine *RunStateFlag* Referenz und kann darin mit Hilfe der betreffenden Getter-Methoden den Zustand abfragen. Dieses Flag enthält zudem die beim *submit()* vergebene *TaskId*, die den laufenden Task identifiziert. Die *TaskId* ist nur gültig zwischen den Zuständen STARTED und FINISHED, ansonsten ist sie *null*. Sie wird für jeden Lauf neu vergeben.

- Die vom *SwingTaskMonitor* erzeugten *TaskEvents* beinhalten eine Kopie des *RunStateFlags*, sowie die für jeden Zustand spezielle Zusatzinformation.

4 Die Zustände CHILD_CANCELLED und CHILD_FAILED existieren nicht, da sie identisch wären mit CANCELLED und FAILED des Main-Tasks.

3.3 Task Zustandswechsel

Ein typischer Ablauf eines Tasks sieht z.B. so aus:

STARTED (immer)

PROGRESS (falls programmiert, *n* mal)

SUCCEEDED (alternativ CANCELLED oder FAILED)

FINISHED (immer)

Die wichtigsten Zustandswechsel sind verbunden mit speziellen Methodenaufrufen. Z.B. geht ein Task, falls er vom Benutzer abgebrochen wurde, in den CANCELLED Zustand über und es wird die Methode *doOnCancel()* aufgerufen. Diese kann überschrieben werden, um mit dem Abbruch eine spezielle Reaktion zu verbinden, wie beispielsweise das Löschen einer Ausgabedatei. Jeder Zustandswechsel wird zudem an evtl. eingehängte Task-Monitore weitergegeben.

4 Abhängige Tasks

4.1 Child-Tasks (*LocalTask*)

Ein Task kann mit *executeChildTask(**LocalTask**)* einen anderen Task als „Child-Task" ausführen. Der „Main-Task" wartet dann bis der Child-Task abgelaufen ist. Ein Abbruch des Child-Tasks führt zum Abbruch des Main-Tasks. Ebenso führt eine vom Child-Task geworfene Exception zum Fehlschlagen des Main-Tasks. Der Child-Task kann seinerseits wiederum Child-Tasks ausführen u.s.w.. Auf diese Weise können beliebig viele geschachtelte Tasks in Abhängigkeit des Main-Tasks ablaufen. Alle Child-Tasks benutzen das *RunStateFlag* des Main-Tasks und unterliegen dadurch einer gemeinsamen Synchronisation. Jeder Child-Task läuft mit einer eigenen *TaskId*, die in der *TaskId*-Liste des *RunStateFlags* eingetragen ist. Der Abbruch des Main-Tasks mit *ctx.cancelTask(TaskId)* führt zum successiven Abbruch aller Child-Tasks. Dabei führen alle Child-Tasks ihre individuelle Reaktion *doOnCancel()* aus.

Ein Child-Task muss im Übrigen nicht im gleichen Threadpool laufen, wie der Main-Task. Mit *executeChildTask(LocalTask, ExecutorService)* kann er in einen beliebigen anderen Pool eingestellt werden.

➔ Man beachte, dass nur Instanzen vom Typ *LocalTask* als Child-Task laufen können. Dies sind i.d.R. Klassen, die von *AbstractLocalTask* abgeleitet sind. Diese Klassen sind also in ihrer Vererbungshierarchie festgelegt.

4.2 Sub-Tasks (AsyncExcecutable)

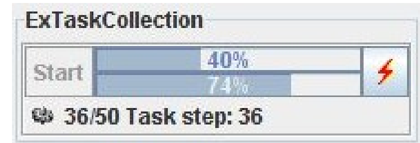
Ein Sub-Task ist eine Klasse, die das Interface *AsyncExecutable* implementiert. Er kann innerhalb eines Tasks mit *executeSubTask(**AsyncExecutable**)* asynchron ausgeführt werden. Dies ist wichtig bei Klassen, die nicht von *AbstractLocalTask* abgeleitet werden können, da sie bereis von einer anderen Klasse ableiten[5].

Sub-Tasks werden direkt im Thread des Main-Tasks ausgeführt, sie erhalten deshalb auch keine echte *TaskId*, sondern werden mit einer *PseudoTaskId* identifiziert. Zur Laufzeit erhalten sie das *RunStateFlag* des Main-Tasks gesetzt. Jeder „echte" *LocalTask* implementiert auch *AsyncExecutable* und kann daher wahlweise entweder als Child-Task oder als Sub-Task laufen.

➔ Man beachte, dass durch die Implementierung von *AsyncExecutable* jede beliebige Klasse asynchron ausführbar gemacht werden kann. Dadurch ist man in der Vererbungshierarchie also nicht festgelegt.

5 Vielfach-Vererbung ist in Java nicht möglich

4.3 Progress von Child- und Sub-Tasks



Beim Start/Ende eines Child/Sub-Tasks erhält ein Monitor die Zustands-Benachrichtigung CHILD_STARTED bzw. CHILD_FINISHED. Alle dazwischenliegenden PROGRESS Meldungen sind dann dem gerade laufenden Child- bzw. Sub-Task zuzuordnen. Der Main-Task zählt in seinem *RunStateFlag,* wieviele Child/Sub-Tasks bereits gelaufen sind. Mit dieser Information, die auch in den betreffenden *TaskEvents* enthalten ist, läßt sich ein doppelter *ProgressBar* steuern. Dabei zeigt der „Main-ProgressBar" die Zahl der bereits gelaufenen Child/Sub-Tasks und der „Child-ProgressBar" visualisiert den Fortschritt des gerade laufenden Child/Sub-Tasks. Damit der „Main-ProgressBar" richtig angesteuert werden kann, sollte dem Main-Task mit *setMaxChildTasks(int)* mitgeteilt werden, wieviele Child/Sub-Tasks voraussichtlich laufen werden.

4.4 Task-Collections

Die Klasse *TaskCollection* ist selbst ein Task, der eine Liste von *AsyncExecutables* beinhaltet, die nacheinander mit *executeSubTask(AsyncExecutable)* aufgerufen werden. So lassen sich auf einfache Weise viele asynchrone Vorgänge in einem einzigen Task zusammenbinden. Jedes der eingehängten *AsyncExecutables* könnte wieder ein *LocalTask* sein, der seinerseits wieder Child- oder Sub-Tasks ausführt. Alle diese Tasks (oder Task-Bäume) laufen synchronisiert unter der Kontrolle des Main-Tasks (in diesem Falle *TaskCollection*) ab. Das folgende Beispiel zeigt eine Zusammenstellung von mehreren Tasks und wie man sie startet:

```
public class ExTaskCollection extends TaskCollection {
     public ExTaskCollection(ApplicationContext ctx) {
            super(ctx);
            addTask(new ExCalculationTask(ctx));
            addTask(new ExDataParser(ctx));
            addTask(new ExXmlParser(ctx));
     }
}
...
ExTaskCollection tasks = new  ExTaskCollection(ctx);
tasks.submit(); // starten
```

Da eine *TaskCollection* vor ihrem Start mit Child/Sub-Tasks befüllt wird, weiss sie natürlich immer, wieviele Tasks laufen werden. Daher ist sie stets in der Lage einen doppelten Progress-Bar richtig anzusteuern.

4.5 Ausführung von *AsyncExecutables*

Klassen, die *AsyncExecutable* implementieren, aber nicht von *AbstractAsyncTask* ableiten, können nicht unmittelbar in einen Threadpool eingestellt werden. Um sie **asynchron** auszuführen, müssen sie vielmehr in einen „echten" Task eingebettet werden. Dazu gibt es einen speziellen *ExecutorTask*, der die *AsyncExecutable*-Instanz im Constructor übernimmt und sie als Sub-Task ausführt, wie das folgende Beispiel zeigt:

```
AsyncExecutable executable = new XMLWriter(ctx);

ExecutorTask task = new ExecutorTask(executable);

TaskId taskId = task.submit(); // starten
```

Darüber hinaus ist es aber auch möglich, ein *AsyncExecutable* **synchron** auszuführen, einfach

durch Aufruf seiner *executeSync()* Methode - der Code läuft dann direkt im aufrufenden Thread. Allerdings läuft es dann zustandslos (ohne *TaskId*) und ist daher nicht cancelbar[6]. Diese Technik empfiehlt sich dann, wenn die Ausführung nur kurze Zeit dauert. Das oben gezeigte Beispiel reduziert sich somit auf eine einzige Zeile:

```
new MyXMLWriter(ctx).executeSync();
```

Eventuelle von der *executeSync()* Methode geworfene Exceptions muss der Anwender dann allerdings selbst behandeln. Bei asynchroner Ausführung würde dies automatisch vom ausführenden Task übernommen.

4.6 Child-Tasks innerhalb von *AsyncExcecutables*

Angenommen ein *AsyncExcecutable* möchte innerhalb seiner *executeAsync()* Methode Daten verarbeiten, die nur von einem anderen Task geliefert werden können. Da dieser Task asynchron läuft, müssen wir auf sein Ergebnis warten. Das folgende Beispiel zeigt, wie dies möglich ist. Der hier gezeigte *GeometryProcessor* leitet von der Basisklasse *AsyncExcecutableAdapter* ab, welcher *AsyncExcecutable* bereits implementiert hat - bis auf die noch fehlende *executeAsync()* Methode:

```
public class GeometryProcessor extends AsyncExcecutableAdapter
    public void executeAsync() throws Exception {
            ...
            GeometryParserTask parser = new GeometryParserTask(ctx);
            parser.submit();
            Geometry geom = parser.awaitResult();
            ...
    }
}
new ExecutorTask(new GeometryProcessor(ctx)).submit();
```

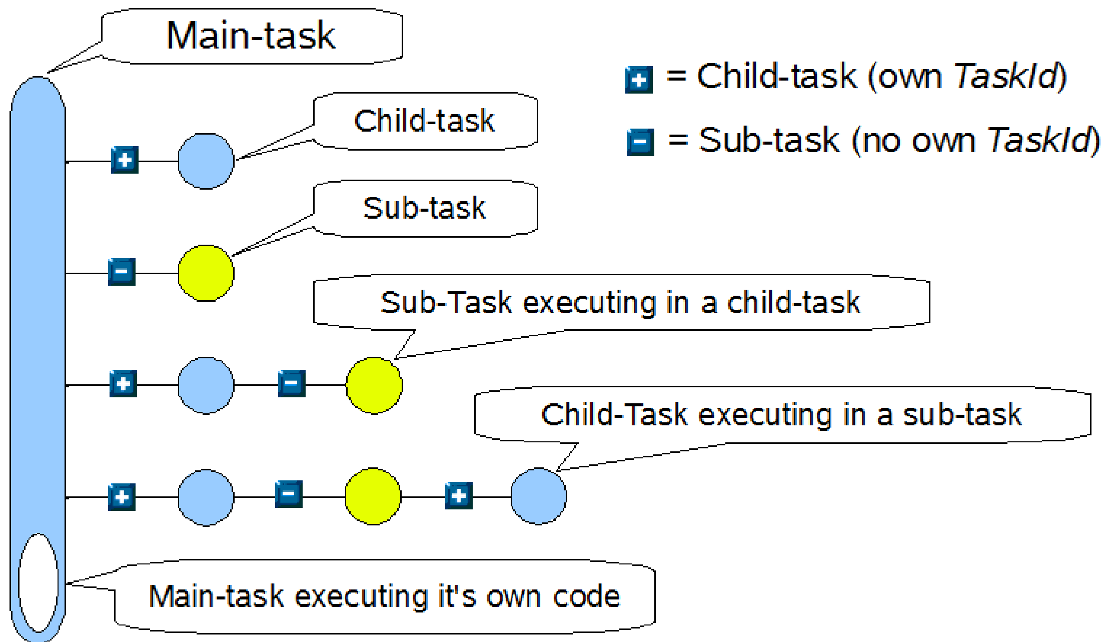
Hierbei blockiert *awaitResult()* so lange, bis der *GeometryParserTask* das Ergebnis fertiggestellt hat. Der *GeometryProcessor* wird seinerseits von einem *ExecutorTask* ausgeführt. Damit haben wir folgende Task-Abhängigkeitsstruktur:

ExecutorTask – *GeometryProcessor* + *GeometryParserTask*
(– = Sub-Task, + = Child-Task)

Wird ein Canceln des ausführenden *ExecutorTask* auf den *GeometryParserTask* durchschlagen? In diesem Fall ist dies nicht selbstverständlich, da der Letzterer mit einer anderen *TaskId* als der *ExecutorTask* läuft. Der Abbruch wird in diesem Falle trotzdem erfolgreich sein, da der *awaitResult()* Aufruf **unterbrechbar** ist und im Falle des Falles eine *InterruptedException* wirft. Als Folge davon cancelt der *GeometryParserTask* sich selbst. Bei der Task-Abhängigkeit *AsyncExecutable*+*AsyncLocalTask* muss vom Anwender auf Unterbrechbarkeit geachtet werden. Hätte man *awaitFinished()* anstelle von *awaitResult()* verwendet, dann würde ein Cancel nicht funktionieren, da diese Methode nicht unterbrechbar ist.

Wie das obige Beispiel zeigt, ist es also möglich, dass ein *AsyncExcecutable* selbst wieder einen Child-Task - mit eigener TaskId und ggf. in einem beliebigen Threadpool - laufen lassen kann. Der *AsyncExcecutable* läuft seinerseits in einem äußeren Task (z.B. dem MainTask). Auf diese Weise lassen sich ganze Task-Bäume aufbauen. Ein Canceln des MainTasks schlägt auf alle im Baum eingebundenen Child-Tasks durch.

6 Man beachte, dass zum Canceln eine *TaskId* benötigt wird, die man aber nur durch das *submit()* in einen Threadpool erhalten kann. Über einen direkten Zugriff auf das *RunStateFlag* des *AsyncExcecutable* ist es immer noch cancelbar.



Dieses Bild zeigt einen Main-Task, der mehrere Tasks hintereinander jeweils als Child- oder Sub-Task laufen lässt und zuletzt auch noch eigenen Code ausführt. Jeder Child- oder Sub-Task kann seinerseits wieder Child- oder Sub-Tasks ausführen. Die aktuelle Schachtelungstiefe spiegelt sich im Render-Namen des Tasks wider *(siehe 5.3)*.

5 Weitere Eigenschaften von Tasks

5.1 Serialisierbarkeit

Tasks sind nicht serialisierbar. Ein Task läuft in der Umgebung eines *ApplicationContexts* und hat typischerweise Referenzen auf Komponenten, mit denen er arbeiten soll. Da nicht erwartet werden kann, dass alle involvierten Komponenten serialisierbar sind, ist ein Task auch nicht serialisierbar. Man beachte aber, dass *AsyncExecutable* Instanzen sehr wohl serialisierbar sein können. Letztere benötigen lediglich einen *ExecutorTask* zur Ausführung.

5.2 Wiederverwendbarkeit

Task Instanzen sind – anders als Threads – wiederverwendbar. Sie können beliebig oft erneut in einen Threadpool eingestellt werden. Bei Tasks, die häufig laufen müssen, bedeutet dies ggf. eine erhebliche Resourcenersparnis, da keine neuen Instanzen erzeugt werden müssen. Allerdings darf derselbe Task nicht zweimal **gleichzeitig** laufen.

5.3 Benamung und Typisierung

Ein Task kann mit *setName(String)* benamt und mit *setType(TaskType)* typisiert werden. Diese Informationen werden z.B. beim Rendern eines Tasks bzw. seines *RunStateFlags* verwendet. Falls nicht anders definiert, wird der einfache Klassenname des Tasks als Render-Name und das in der Konstanten *LocalTaskType.LOCAL_TASK* definierte Icon als Render-Icon verwendet. Der Render-Name verlängert sich, je nach Verschachtelungstiefe von Child/Sub-Tasks, wie z. B:

MyMainTask + *MyChildTask* – *MySubTask*

In diesem Beispiel läuft im *MyMainTask* gerade der ChildTask *MyChildTask*, der seinerseits den SubTask *MySubTask* ausführt (+ = ChildTask, – = SubTask).

5.4 Abbrechbarkeit

Ein Task kann kritische Phasen durchlaufen, in denen er nicht abbrechbar sein darf. Mit *setCancellable(boolean)* kann der Task sich jederzeit selbst entsprechend einstellen. Diese Information wird mit jeder PROGRESS Meldung an Monitore weitergegeben. Beim Versuch, einen nicht cancelbaren Task abzubrechen, erhält der Aufrufer eine *TaskNotCancellableException.*

5.5 Vorzeitiges Beenden eines Task

Hat ein Task sein Ziel erreicht, dann ist normalerweise seine *executeAsync()* Methode zu Ende gelaufen. Es gibt aber auch Fälle, wo ein vorzeitiges Beenden vorteilhaft ist. Z.B. könnte man beim Parsen einer Datei feststellen, dass man die gesuchte Information gefunden hat und möchte dann möglichst unmittelbar terminieren. In einem solchen Falle kann einfach eine ***StopException*** (eine *RuntimeException*) geworfen werden. Sie wird nicht als Fehler oder Abbruch gewertet, sondern beendet den Task im SUCCESS Zustand.

5.6 Threadpool-Zuordnung

Ein Task kann mit *submit(ExecutorService)* in einen bestimmten Threadpool eingestellt werden. Typischerweise wird diese Methode aber so überschrieben, dass der Task selbst weiß, in welchen Threadpool er gehört. Daher genügt zum Starten des Tasks in der Regel der Aufruf von *submit()* ohne Argumente.

5.7 Task Priorität

Ein Task kann mit *setPriority(int)* mit einer Priorität versehen werden, die dann vom ausführenden Thread eines Threadpools übernommen wird. Als Priority-Wert sind nur die in der Thread-Klasse definierten Priority-Konstanten zulässig. Der Standard-Wert ist *Thread.NORM_PRIORITY.* Man beachte, dass die Auswirkung der Priorität plattformabhängig sein kann. In der Regel verhält es sich so, dass ein höher priorisierter Task den Lauf von niedriger priorisierten Tasks blockiert, bis er selbst zu Ende gelaufen ist oder ebenfalls blockiert. Prioritäten sollten daher vorsichtig und sparsam angewendet werden.

6 Verwaltung von Tasks

Der *ApplicationContext* beinhaltet den *TaskManager*, dessen Aufgabe es ist, die laufenden Tasks zentral zu verwalten. Über ihn sind alle Tasks anhand ihrer *TaskId* zugreifbar, unabhängig davon, in welchem Threadpool sie laufen.

6.1 Task-Registrierung

Jeder Task, der mit *submit()* in einen Pool eingestellt wurde, registriert sich beim *TaskManager*, wenn er losgelaufen ist und deregistriert sich wieder, wenn er zu Ende gelaufen ist. In der Zwischenzeit kann er anhand seiner *TaskId* aufgefunden und ggf. abgebrochen werden. Letzteres ist nicht unbedingt bei allen Tasks wünschenswert. Bestimmte wichtige Service-Tasks beispielsweise dürfen nicht abbrechbar sein. Solche Tasks wurden mit set*RegisterForCancel(false)* so eingestellt, dass sie als nicht-cancelbar registriert werden. Der Aufruf *cancelAll()* im *TaskManager* bricht alle cancelbaren Tasks ab. Wenn der *ApplicationContext* geschlossen wird, z.B. beim Terminieren des Hauptprogramms, dann werden alle laufenden Tasks automatisch abgebrochen und der Threadpool wird heruntergefahren.

6.2 Abbrechen von Tasks

Der Vorgang des Abbruchs ist komplexer, als man vermuten mag. Ein Abbruch führt stets dazu, dass im laufenden Task die *cancel()* Methode aufgerufen wird. Diese markiert das *RunStateFlag* des Tasks unwiderruflich als gecancelled. Der Task überprüft regelmäßig, spätestens bei jeder Zustandsänderung (insbesondere PROGRESS), ob das Cancel-Flag gesetzt wurde. Wenn ja wirft er eine (unchecked) *CancelledException*.

Was aber passiert, wenn der Task sich gerade in einem geblockten Zustands befindet, wie z.B. *Thread.sleep(long)*? Er würde dann nicht reagieren! Im Java Concurrency Framework bietet das *Future* Objekt die Möglichkeit, den Task auch aus einem solchen Zustand „herauszureißen" - er wirft dann eine (checked) *InterruptedException*. Dieser Mechanismus wird intern auch im hier beschriebenen Framework genutzt.

Beide Exceptions führen zum gleichen Ergebnis, nämlich dass der Task in den CANCELLED Zustand wechselt, als Reaktion *doOnCancel()* abarbeitet, evtl. eingehängte Monitore informiert und schließlich terminiert[7].

➔ Es ist guter Programmierstil innerhalb der *executeAsync()* Methode eines Tasks regelmäßig *checkCancelled()* aufzurufen. Dadurch sind die möglichen Abbrechpunkte definiert. Alternativ kann auch *sleepChecked(millis)* verwendet werden. Die damit verbundene Verzögerung gibt anderen parallel laufenden Tasks - insbesondere auch dem Hauptprogramm - die Möglichkeit zu laufen und führt zu einer gleichmäßigeren CPU-Resourcenverteilung.

6.3 Besondere Task Typen

Jede Aufgabe, die asynchron ausgeführt werden soll, kann als *AsyncTask* bzw. als Ableitung der *AbstractAsyncTask*-Klasse formuliert werden. Eine ganze Reihe nützliche Tasks sind im Esprit-Framework bereits enthalten und stehen zur Verwendung zur Verfügung. Hier nur die Wichtigsten:

6.3.1 AbstractSwingTask

Dieser Task beinhaltet bereits einen *SwingTaskMonitor*, der die asynchronen Task-Zustandswechsel in GUI-synchrone *TaskEvents* umwandelt. Durch Überschreiben der diversen *done** Methoden Können Sie auf diese Events reagieren. Diese Methoden werden synchron im *EventDispatcherThread* aufgerufen und dürfen deshalb Aktionen im GUI ausführen.

6.3.2 AbstractOsProcessTask

Dieser Task steuert den Ablauf eines externen Betriebssystem-Prozesses, der mit *setCommand(String[])* definert wird und in dem mit *setWorkingDir(File)* eingestellten Arbeitsverzeichnis arbeitet. Mit *putEnv(String,String)* können - falls erforderlich - dem Prozess Umgebungsvariablen mitgegeben werden.

Mit *setNotifyConsoleOutput(true)* aktivieren Sie das Horchen auf Consolen-Ausgaben des Prozesses. Durch Überschreiben von *doOnConsoleOutput(String)* kann auf jede Ausgabezeile einzeln reagiert werden. Mit *setOutputFile(File)* wird die Consolen-Ausgabe in eine Datei umgeleitet.

Betriebssystem-Prozesse schreiben ihre Ergebnisse typischerweise in neue Dateien im Filesystem. Die Methode *setNotifyFileSystemChange(true)* aktiviert einen *WatchService*, der das Arbeitsverzeichnis des Prozesses beobachtet. Durch Überschreiben von *doOnFileSystemChange(ChangeType, File)* kann auf jede Filesystem-Änderung reagiert werden.

Ein erfolgreich gelaufener Betriebssystem-Prozess terminiert normalerweise mit dem Exit-Status 0.

---

7 Noch komplexer wird der Abbruch von Sub-Tasks und Child-Tasks, die ja in beliebiger Schachtelungstiefe laufen können. Eine detaillierte Betrachtung würde den Rahmen dieser Abhandlung sprengen.

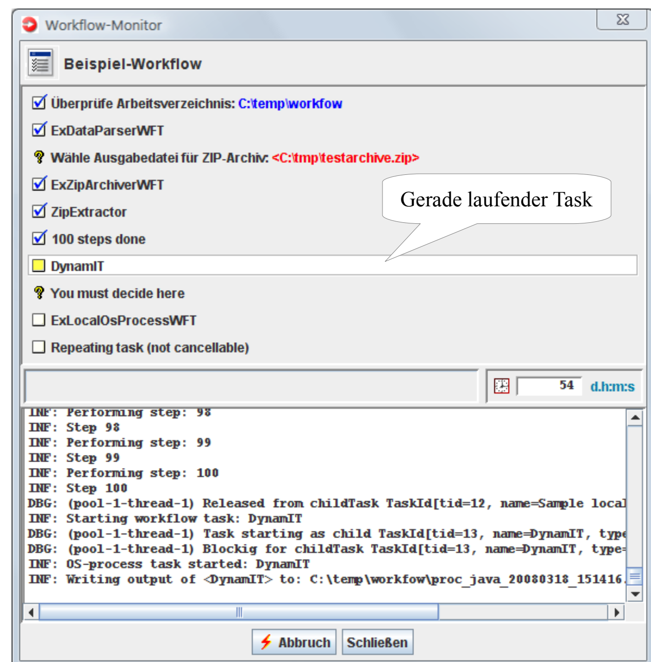
Jeder andere Wert wird als Fehler interpretiert. Durch Überschreiben von *checkExitStatus(int)* ist dieses Verhalten anpassbar.

6.3.3 JavaCommandTask

Dies ist ein besonderer *OsProcessTask*, der eine externe Java-VM startet und zwar in jeder beliebigen auf dem System installierten Java-Version (siehe auch *JavaVersionManager*). Die vielen möglichen Eingaben zur Definition sind in einem *JavaCommand* Objekt gekapselt: die zu startende Java-Klasse, alle zum Ablauf notwendigen Jar-Dateien und native Bibliotheken sowie die Runtime- und VM-Optionen.

7 Workflows

Unter einem *Workflow* verstehen wir einen besonderen Task (abgeleitet von *AbstractWorkflow*), der eine Ansammlung von *WorkflowTasks* als Child-Tasks nacheinander ablaufen lässt. Im Unterschied zu einer *TaskCollection* kann seine Abarbeitung direkt im GUI mit Hilfe eines *Workflow-MonitorDialogs* beobachtet werden, wie im unten stehenden Bild gezeigt. Zudem hat jeder *WorkflowTask* Zugriff auf seinen Vorgänger und kann dessen Ergebnis überprüfen, bzw. darauf aufbauen.



Jeder abzuarbeitende *WorkflowTask* wird dargestellt durch ein Check-Label. Der gerade aktive Task erscheint hervorgehoben. Ein bereits gelaufener Task ist abgehakt. Ein mit Fragezeichen markierter Task fordert eine interaktive Eingabe oder eine Entscheidung vom Benutzer. Je nach Entscheidung können dynamisch neue Tasks eingefügt, vorhandene gelöscht oder übersprungen werden.

Ein *ProgressBar* zeigt die Fortschritte des gerade laufenden Tasks, das Zeitfenster daneben misst die Gesamtdauer. Zusätzlich hat der Monitor-Dialog einen eigenen Log-Kanal, in dem die Consolen-Ausgaben der Tasks erscheinen.

7.1 Workflow Task

Alle Tasks eines Workflows implementieren das Interface *WorkflowTask*, welches drei wichtige Methoden vorschreibt:

@Sync enterTask(WorkflowTask previousTask)	Läuft GUI-synchron zu Beginn eines Tasks
@Async executeAsync()	Hauptteil des Tasks – läuft asynchron
@Sync exitTask()	Läuft GUI-synchron am Ende eines Tasks

Die mit *@Sync* annotierten Methoden laufen GUI-synchron im *EventDispatcherThread*. In diesen Methoden ist der Zugriff auf GUI-Komponenten problemlos. Die mit *@Async* markierte Methode *executeAsync()* läuft in dem Thread, der den *Workflow* abarbeitet; von hier aus ist ein Zugriff auf das GUI verboten. Aufgrund dieser Methoden gibt es in einem *WorkflowTask* also einen definierten Wechsel zwischen GUI-synchronem und -asynchronem Code.

Schauen wir uns ein einfaches Workflow-Beispiel an:

```
public class ExWorkflow extends AbstractWorkflow {
     public ExWorkflow(ApplicationContext ctx) throws Exception {
            super(ctx);
            addTask(new ExDataParserWFT(this));
            addTask(new ExZipFileChooserWFT(this)); // speichert Benutzer-Eingabe
            addTask(new ExZipArchiverWFT(this));    // benutzt Eingabe von Vorgänger
     }
}
```

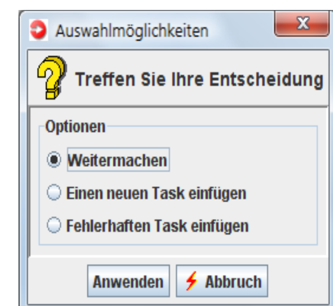
In diesem *Workflow* sind drei *WorkflowTasks* eingebunden. Der Erste ist ein „gewöhnlicher“ Task, der einfach asynchron abläuft. Der Zweite ist insofern ein Besonderer, als er GUI-interaktiv wird: er fragt den Benutzer per Dialog nach einer Ausgabedatei und speichert diese. Der Dritte übernimmt innerhalb von *enterTask(...)* das Ergebnis seines Vorgängers und benutzt es in seiner anschließenden asynchronen Ausführung. Auf diese Weise erfolgt eine Datenübergabe zwischen aufeinander folgenden Tasks.

7.2 Decider Task

Decider-Tasks sind solche, die auf die Frage *isDeciderTask()* mit *true* antworten. Sie werden mit einem Fragezeichen-Icon dargestellt und beinhalten typischerweise eine GUI-Interaktion, die vom Benutzer eine Entscheidung fordert.

Ein Beispiel für einen Decider-Task könnte so aussehen:

```
public class ExTaskOptionDialogWFT extends AbstractTaskOptionDialogWFT {
     public ExTaskOptionDialogWFT(AbstractWorkflow workflow) {
            super(workflow);
            setName("Treffen Sie Ihre Entscheidung");
     }
     @Override
     public void fillAvailableOptions(List<TaskOption> list, WorkflowTask prevTask) {
            list.add(StandardTaskOption.CONTINUE);
            list.add(ExTaskOption.INSERT_ANOTHER_TASK);
            list.add(ExTaskOption.INSERT_ERRONEOUS_TASK);
     }
     @Override
     public TaskOption getDefaultOption() {
            return StandardTaskOption.CONTINUE;
     }
}
```



Dieser *Decider-Task* generiert einen Auswahl-Dialog wie im nebenstehenden Bild gezeigt. Alle in die Optionsliste eingefüllten *TaskOptions* werden dem Benutzer zur Auswahl angeboten, wobei die Default-Option bereits vorselektiert ist.

Ein einfaches Schließen dieses Dialogs ist nicht möglich, vielmehr ist der Benutzer zu einer eindeutigen Entscheidung gezwungen. Ein Abbruch würde die Fortführung des gesamten Workflows beenden.

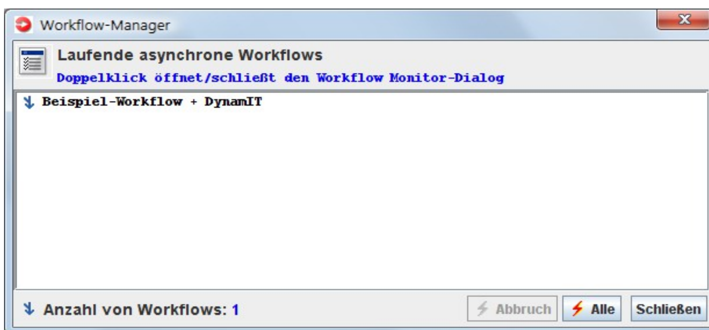
Die Aktion, die auf die gewählte Option erfolgt, ist in der Methode *handleDecision(TaskOption)* des Workflows programmiert, wie im folgenden Beispiel gezeigt.

➔ Man beachte, dass bei Auswahl einer *StandardTaskOption* (*CONTINUE, SKIP_NEXT, SKIP_ALL*) diese automatisch ausgewertet wird – *handleDecision(TaskOption)* wird in diesem Falle nicht mehr aufgerufen.

```
public class ExWorkflow extends AbstractWorkflow {
      ...
      @Override
      public void handleDecision(TaskOption option) {
             if (option == ExTaskOption.INSERT_OTHER_TASK) {
                    insertTask(new ExLocalAsyncWFT(this));
             }
             if (option == ExTaskOption.INSERT_ERRONEOUS_TASK) {
                    insertTask(new ExErroneousWFT(this));
             }
      }
}
```

7.3 Workflow Manager

Da Workflows eventuell aus vielen *WorkflowTasks* bestehen, kann es durchaus vorkommen, dass sie Laufzeiten im Bereich von Minuten oder Stunden haben. Es ist möglich, beliebig viele verschiedene Workflows gleichzeitig laufen zu lassen - jedoch keinen doppelt. Jeder läuft mit seinem eigenen *WorkflowMonitorDialog*. Viele Dialoge aber würden das GUI überladen. Daher kann man die Dialoge einfach schließen ohne die Workflows anzuhalten. Der *WorkflowManager* verwaltet alle diese Dialoge und erlaubt den steten erneuten Zugriff. Ist ein Workflow zu Ende gelaufen, öffnet sich sein Monitor-Dialog automatisch für eine abschließende Kontrolle.



Ein Workflow kann alle Arten von Tasks beinhalten, insbesondere auch remote Tasks, die auf einem Server oder CoServer laufen (werden weiter unten beschrieben). Dazu gehören auch transaktionale File-Transfers sowie Stream-Downloads. Darüberhinaus können auch komplexe GUI-Interaktionen eingebunden werden, wie z.B. der Aufruf eines CAD-Systems, dessen Ausgabe anschließend asynchron im Workflow weiterverarbeitet wird.

8 Remote Tasks

Bisher haben wir nur solche Tasks behandelt, die in einer lokalen Java-VM laufen (auf dem Client). Im Folgenden betrachten wir sog. *remote* Tasks, die in einer entfernten Java-VM laufen (auf dem Server). Als „*remote*“ betrachten wir einen Task dann, wenn er - von einem Client initiiert - auf dessen Server in der Runtime-Umgebung eines *ServerContext* läuft[8].

8.1 Remote Tasks auf Server

Wir wollen unseren Beispiel-Task so umschreiben, dass er als *RemoteTask* auf dem Server läuft:

```
public class ExRemoteCounterTask extends AbstractRemoteTask {

     public ExRemoteCounterTask(ServerContext serverCtx) {
            super(serverCtx);
     }

     @Override
     public void executeAsync() throws Exception {
            logInfo("Remote counter-task started by: "+getOwnerId().getUserInfo());
            for (int i = 0; i < 100; i++) {
                   sleepChecked(50); // bee friendly and check for cancellation
                   notifyProceeded(null, i, 100, "Task step: "+i);
            }
     }
}
```

Dieser Task implementiert das Interface *RemoteTask*, das u.a. vorschreibt, wie er in einen Threadpool des Servers einzustellen ist, nämlich mit dem Aufruf *submit(SessionId)*, der eine *RemoteTaskId* zurück gibt. Durch Angabe der *SessionId* weiß der Server stets, wer der Eigentümer (*getOwnerId()*) des Tasks ist. Der beim *RemoteTask* standardmäßig eingehängte *RemoteTaskMonitor* erzeugt für jeden Zustandswechsel das entsprechende *RemoteTaskEvent* und weiß anhand der *SessionId* an wen die Fortschrittsmeldungen zu senden sind.

Zum Start des remoten Tasks wird ein zusätzlicher lokaler *ClientEDT* (**EDT** = **E**vent **D**riven **T**ask) benötigt, der die *RemoteTaskEvents* des serverseitig laufenden Tasks empfängt und verarbeitet. Hier ist ein einfaches Beispiel für einen lokalen Task *ExSimpleEDT*, der den den remoten Task *ExRemoteCounterTask* startet und sich an ihn bindet:

```
public class ExSimpleEDT extends AbstractClientEDT {
     public ExSimpleEDT(ClientContext ctx) {
            super(ctx);
     }
     @Override
     public void receivedTaskEvent(RemoteTaskEvent event) {
            logInfo("Received event: "+event); // react on events
     }
     @Override
     protected RemoteTaskCreator getRemoteTaskCreator() {
            return new MyTaskCreator();
     }
```

8 Diese Funktionalität steht nur im *EspritNetSuite* Framework zur Verfügung

```
    private static class MyTaskCreator implements RemoteTaskCreator {
        RemoteTask createRemoteTask(ServerContext serverCtx) {
            return new ExRemoteCounterTask(serverCtx);
        }
    }
}
```

Man beachte die statische Hilfsklasse *MyTaskCreator* und die Methode *getRemoteTaskCreator()*, die eine Instanz davon bekannt gibt. Diese Instanz ist serialisierbar und dient lediglich dazu, den gewünschten Task *ExRemoteCounterTask* auf Serverseite zu erzeugen (zur Erinnerung: Task-Instanzen selbst sind NICHT serialisierbar).

Wenn der *ExSimpleEDT* Task gestartet wird, sendet er einen *RemoteTaskStartAgent* zum Server (in der Superklasse *AbstractClientEDT* implementiert), der den *RemoteTaskCreator* verwendet, um in der Server-Umgebung den remoten Task zu erzeugen. Der Agent stellt den erzeugten Task in einen serverseitigen Threadpool ein und erhält dabei dessen *RemoteTaskId*, die er in seiner Response dem Client übermittelt.

Die Methode *receivedTaskEvent(RemoteTaskEvent)* wird jedes mal aufgerufen, wenn ein Event des serverseitig laufenden Tasks empfangen wurde. Durch Überschreibung der jeweiligen *do** Methoden kann hier gezielt auf die verschiedenen Zustandsänderungen reagiert werden.

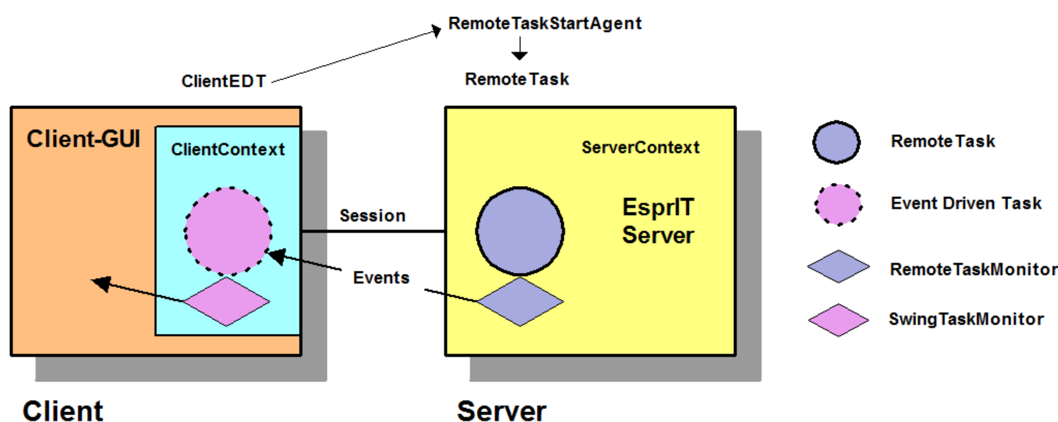
Der folgende Code zeigt, wie der *ExSimpleEDT* Task – und damit auch sein remoter Gegenpart *ExRemoteCounterTask* – letztendlich gestartet wird und ggf. abgebrochen werden kann:

```
ExSimpleEDT edt = new ExSimpleEDT(clientCtx);
TaskId taskId = edt.submit(); // TaskId des lokalen EDT (i.d.R. nicht benötigt)
...
RemoteTaskId remoteTaskId = edt.getRemoteTaskId();
clientCtx.cancelTask(remoteTaskId); // RemoteTaskId des remoten Tasks
```

In einem *ClientContext* (auch *CoClientContext*) ist die Methode *cancelTask(TaskId)* so implementiert, dass für eine *RemoteTaskId* in Wirklichkeit ein *RemoteTaskCancelAgent* gestartet wird, der den Task auf Serverseite abbricht. Der korrespondierende lokale *ClientEDT* terminiert automatisch als Konsequenz davon.

➔ Der *ClientEDT* erwartet zwingend das FINISHED Event seines remote laufenden Gegenparts. Im Falle eines Netzwerkfehlers wird er selbst das fehlende Event erzeugen und sauber terminieren.

Das folgende Bild verdeutlicht den *RemoteTask* Mechanismus:

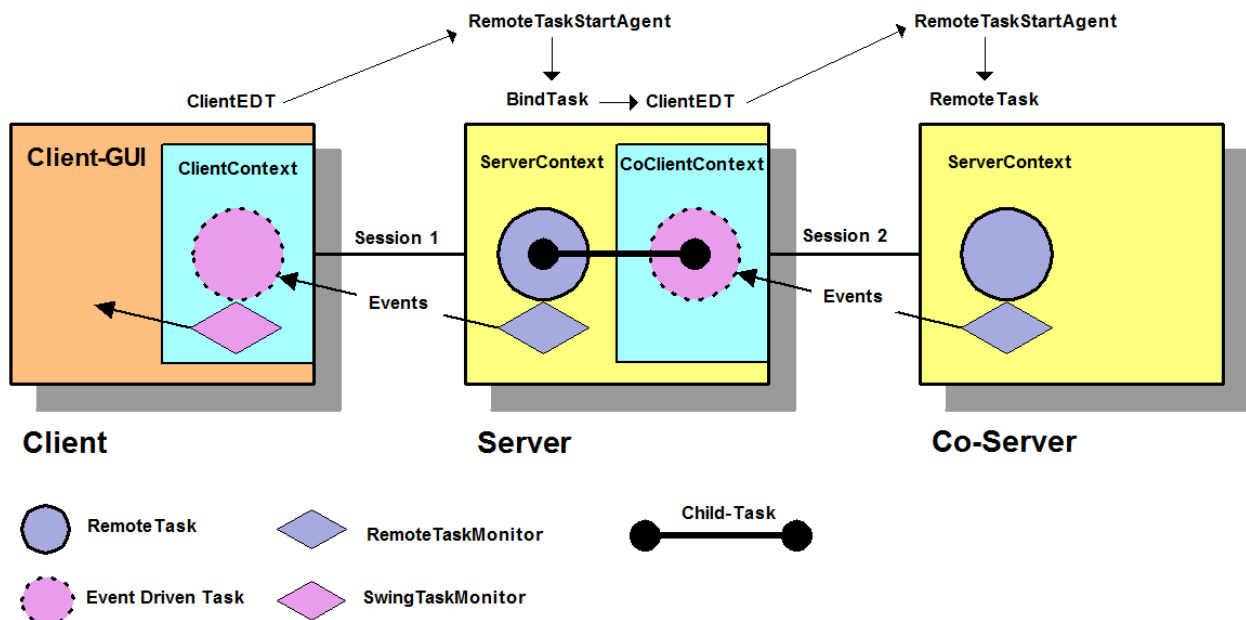


Der dargestellte *ClientEDT* besitzt seinerseits einen *SwingTaskMonitor*, der die empfangenen *RemoteTaskEvents* an das Client-GUI weitergibt, womit sich beispielsweise ein *ProgressBar*

antreiben lässt, der dann die Fortschritte des remoten Tasks visualisiert. Wie dieses Bild verdeutlicht, sind bei der remoten Ausführung zwei Tasks involviert, die gekoppelt laufen: ein clientseitiger *ClientEDT* und ein serverseitiger *RemoteTask*.

8.2 Remote Task auf CoServer

Als nächsten Schritt wollen wir einen remoten Task auf einem CoServer laufen lassen. Die Verbindung zwischen einem Server und seinem CoServer unterscheidet sich nur unwesentlich von einer normalen Client-Server Verbindung - sie ist durch eine *CoClientContext*-Instanz implementiert. Auch in einem *CoClientContext* kann ein *ClientEDT* laufen, gekoppelt mit einem *RemoteTask* auf dem CoServer. Um die vom CoServer empfangenen *RemoteTaskEvents* zum Client weiterzuleiten, ist ein zusätzlicher *BindTask* auf dem Server notwendig, der den *ClientEDT* als Child-Task laufen lässt. Bei der remoten Ausführung auf einem CoServer sind also mindestens vier Tasks involviert, von denen jeder in einem anderen Context läuft - das folgende Bild soll dies veranschaulichen:



Zum Start eines Tasks auf einem CoServer sind also folgende Schritte notwendig:

- Auf dem Client:
  Start des *ClientEDT*. Dieser benutzt einen *RemoteTaskStartAgent* um den serverseitigen *BindTask* zu starten (der dem Client wie ein gewöhnlicher *RemoteTask* erscheint). Sodann wartet der *ClientEDT* auf die vom *BindTask* gesendeten *RemoteTaskEvents*.
- Auf dem Server:
  Start des *BindTask*. Dieser sucht den *CoClientContext* des Ziel-CoServers und startet in diesem einen weiteren *ClientEDT* als Child-Task. Letzterer benutzt wiederum einen *RemoteTaskStartAgent*, der auf dem CoServer schließlich den gewünschten *RemoteTask* startet.
- Auf dem CoServer
  Start des *RemoteTask*. Dieser erledigt seine Aufgabe und sendet dabei Fortschrittsmeldungen in Form von *RemoteTaskEvents* an seinen Client (in diesem Falle den *CoClientContext*).

Die *RemoteTaskEvents* werden - wie aus der Abbildung ersichtlich - vom CoServer über den Server quasi zum Client „zurück geroutet“, der dann auf die empfangenen Events synchronisierte Aktionen durchführen kann. Der beschriebene Vorgang ist codemäßig in einer einzigen Klasse abgebildet, wie das folgende Beispiel zeigt:

```
public class ExSimpleCoServerEDT extends AbstractClientEDT {
	public ExSimpleCoServerEDT(ClientContext ctx, String coServer) {
		super(ctx, coServerName);
	}

	@Override
	protected RemoteTaskCreator getRemoteTaskCreator() {
		return CoServerBindTaskCreator(coServerName, new MyTaskCreator());
	}

	private static class MyTaskCreator implements RemoteTaskCreator {
		public RemoteTask createRemoteTask(ServerContext srvCtx) {
			return new ExRemoteCounterTask(serverCtx, 100);
		}
	}
}
```

Der wesentliche Unterschied zu *ExSimpleEDT* besteht darin, dass *getRemoteTaskCreator()* eine Instanz von *CoServerBindTaskCreator* zurückgibt, die ihrerseits eine Instanz von *MyTaksCreator* beinhaltet. Dieser Creator erzeugt serverseitig das Task-Pärchen (*BindTask* /*CoClientEDT*), das die Erzeugung bzw. Kontrolle des eigentlichen *ExRemoteCounterTasks* an den CoServer weiterdelegiert. Das durchaus komplexe Zusammenspiel der involvierten Tasks wird vom Framework gehandhabt und bleibt dem Anwender vollständig verborgen.