ESPRITSYSTEMS

Esprit Workflows

Programmierung client- und serverseitiger Workflows

November 2015
Rainer Büsch

Inhaltsverzeichnis

1 Einführung

Dieses Dokument beschreibt die Implementierung von client- und serverseitigen *Workflows* im Esprit-System. Als „*Workflow“* betrachten wir einen Task (*WorkflowMainTask*), der eine vordefinierte Abfolge von asynchronen Tasks (*WorkflowTasks*) ausführt, die als Ganzes eine bestimmte Aufgabe erledigen. Ein Workflow kann wahlweise auf einem Client oder einem Server ablaufen. Der Ablauf des Workflows kann mit Hilfe eines graphischen *WorkflowMonitors* beobachtet und bei Bedarf abgebrochen werden.

Darüber hinaus können Workflows eine Interaktion mit dem Benutzer erfordern. Beispielsweise kann ein Workflow den Benutzer zur Eingabe eines Dateinamens auffordern, wobei er solange blockiert, bis die Eingabe erfolgt ist. Typischerweise verarbeitet dann der nächste Task des Workflows die Benutzereingabe.

Eine Benutzereingabe kann den nachfolgenden Ablauf des Workflows auf diverse Arten verändern:

- ➔ Es können bestimmte oder alle nachfolgenden Tasks übersprungen werden
- ➔ Es können neue Tasks eingefügt werden
- ➔ Der Workflow kann abgebrochen werden

2 Das Workflow-Konzept

2.1 Schnittstellen

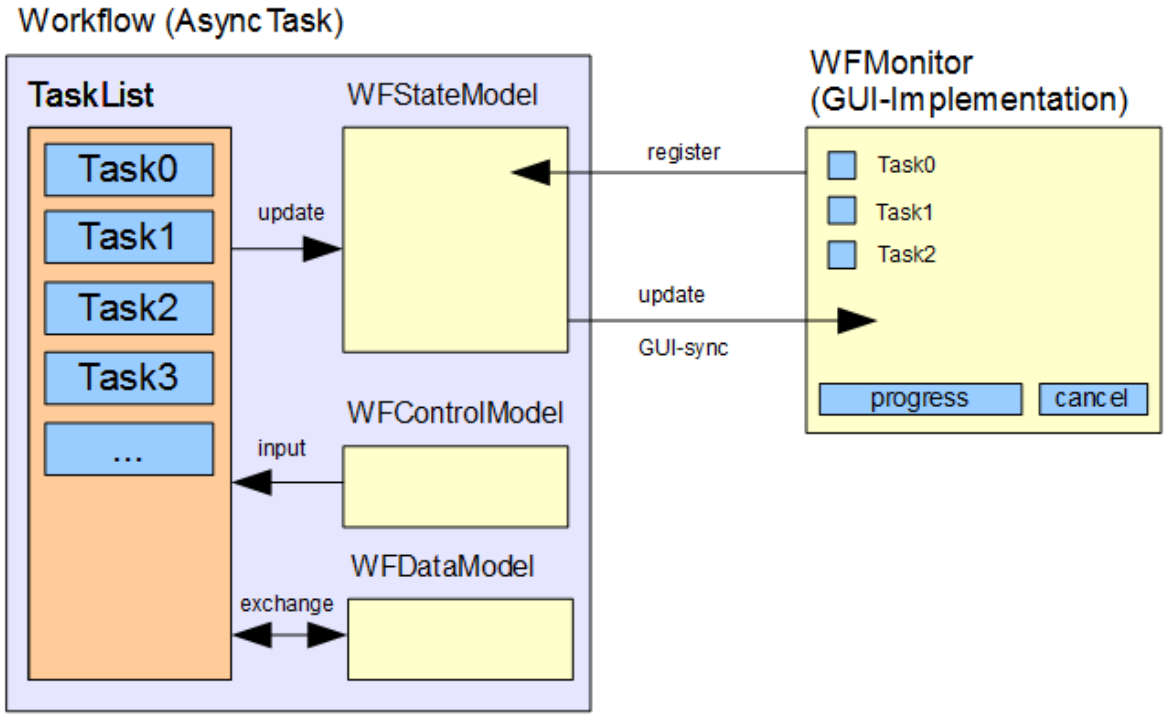
Damit Workflows unverändert sowohl auf Client, als auch auf Server lauffähig sind, müssen sie selbst GUI-frei sein. Zur Beobachtung ist jederzeit ein GUI-Monitor anschließbar sein, der den Zustand des Workflows anzeigt und verfolgt.

Der **Workflow** ist selbst ein **AsyncTask** (abgeleitet von *AbstractAsyncTask*) und kann als solcher in einen Threadpool eingestellt werden und dort asynchron laufen. Er enthält eine Liste einzelner *WorkflowTasks***,** die der Reihe nach als Sub-Tasks abgearbeitet werden. Ein *Workflow* beinhaltet die folgenden drei Modelle, die während des Ablaufs angesteuert werden.

- ➔ Interface **WorkflowStateModel** (implementiert als *DefaultWorkflowStateModel*) repräsentiert den vollständigen aktuellen Zustand des Workflows mit allen Informationen, die ein Workflow-Monitor (*WorkflowMonitor*) zur graphischen Darstellung benötigt. Z.B. welche Tasks sind bereits abgearbeitet, welche geskipped, welcher ist gerade am Laufen, wie steht der Progress, wie wurde terminiert? Etc...
- ➔ Interface **WorkflowControlModel** (implementiert als *DefaultWorkflowControlModel*) wird vor der jeweiligen Abarbeitung eines Tasks befragt. Es kann die Abarbeitung eines Workflows beeinflussen (skippen von Tasks, terminieren von Tasks, oder beliebige Optionen für die Weiterverarbeitung vorgeben).
- ➔ Interface **WorkflowDataModel** (implementiert als *DefaultWorkflowDataModel*) dient dem Austausch von Daten zwischen den einzelnen WorkflowTasks. Ein Task kann dort beliebige Informationen einstellen, die dann von einem nachfolgenden Task verarbeitet werden.
- ➔ Interface **WorkflowTask** definiert, was ein Task implementieren muss, damit er in einem Workflow laufen kann.

2.2 Aufbau eines Workflows

Das folgende Blockdiagramm zeigt den Aufbau eines *Workflows* mit der beinhalteten Task-Liste und seinen zugehörigen Modellen *WorkflowStateModel*, *WorkflowControlModel*, und *WorkflowDataModel*. Am *WorkflowStateModel* ist zusätzlich ein Workflow-Monitor (Interface *WorkflowMonitor*) angeflanscht, der das Zustandsmodell visualisiert. Die Implementierung als GUI-Dialog stellt die Zustände der einzelnen Tasks als Check-Box mit jeweiligem Label dar. Zusätzlich enthält er einen Progress-Bar zur Visualisierung der Task-Fortschritte, sowie einen Cancel-Button zum Abbruch des Workflows.



2.2.1 Das WorkflowStateModel

Das *WorkflowStateModel* repräsentiert den aktuellen Zustand des Workflows. Es beinhaltet sämtliche Information, die ein *WorkflowMonitor* benötigt, um den Zustand vollständig zu visualisieren. Der Zustand ändert sich zum Beispiel, sobald ein Task abgelaufen ist, und ein Neuer gestartet wurde. Auch die Progress-Meldungen eines Tasks bedeuten eine Zustandsänderung. Bei Änderung wird der *WorkflowMonitor* informiert, sofern ein solcher beim *WorkflowStateModel* registriert ist.

2.2.2 Der Workflow-Monitor

Der Workflow-Monitor registriert und initialisiert sich synchron beim *WorkflowStateModel* und erhält daraufhin asynchron die Aktualisierungen (die in einer grafischen Implementierung GUI-synchron gemacht werden müssen). Wichtig ist, dass der *WorkflowMonitor* sowohl bei der Initialisierung als auch bei den Aktualisierungen eine Snapshot-Kopie des *WorkflowStateModels* erhält. Dadurch ist garantiert, dass sich das *WorkflowStateModel* während der Monitor-Aktualisierung nicht verändert. Der *WorkflowMonitor* hat ausschließlich Verbindung zum *WorkflowStateModel*. Nichtsdestotrotz ist es möglich, dass der Monitor den Ablauf des Workflows

über das *WorkflowControlModel* steuert. Dies geschieht allerdings indirekt über einen laufenden *WorkflowTask*. Letzterer kann eine Benutzereingabe anfordern, die er dann als Steuerkommando an das *WorkflowControlModel* absetzt.

2.2.3 Das WorkflowControlModel

Das *WorkflowControlModel* dient dem Zweck, auf den laufenden Workflow-Ablauf Einfluss nehmen zu können. Dies kann mit Hilfe spezieller sogenannter *Decider-WorkflowTasks* geschehen. Ein solcher Task wird interaktiv mit dem Benutzer, indem er ihm eine Auswahl aus einer Liste von *CheckOption*s anbietet. Die Entscheidung des Benutzers wird mit der Methode *put(CheckOption)* in das *WorkflowControlModel* eingestellt.

Nach der Ausführung eines *Decider-WorkflowTasks* fragt der *WorkflowMain-Task* die Entscheidung aus dem *WorkflowControlModel* ab und bearbeitet sie, indem er seine *handleDecision(CheckOption)* Methode aufruft. Diese Methode wurde typischerweise vom Anwender überschrieben und führt die gewünschten Aktionen aus, bevor der nächste *WorkflowTask* aufgerufen wird. Mögliche Aktionen können sein:

- ➔ Einfach weitermachen (*DeciderStandardOption.CONTINUE*)
- ➔ Überspringen des nächten Tasks (*DeciderStandardOption.SKIP_NEXT*)
- ➔ Überspringen aller Tasks (*DeciderStandardOption.SKIP_ALL*)
- ➔ Neuen Workflow-Task einfügen (Kundenspezifische Option)

➔ In der *handleDecision(CheckOption)* Methode müssen nur kundenspezifische Optionen behandelt werden. Alle *DeciderStandardOptions* werden bereits intern behandelt.

2.2.4 Das WorkflowDataModel

Das *WorkflowDataModel* dient dem Austausch von Daten (in Form von Key-Value-Paaren) zwischen den einzelnen *WorkflowTasks*. Zum Beispiel könnte ein *WorkflowTask* den Benutzer interaktiv nach dem Namen für eine Ausgabedatei fragen. Die Eingabe des Benutzers wird dann mit Hilfe der Methode *put("fileNameKey", fileName)* in das *WorkflowDataModel* eingestellt. Ein nachfolgender *WorkflowTask* holt sich diese Information mit *get("fileNameKey")* wieder aus dem *WorkflowDataModel* heraus um sie zu verarbeiten.

Alternativ kann eine zweite Methode *getAndWait("fileNameKey")* aufgerufen werden, um auf eine Eingabe zu warten, falls sie z.B. durch den *Event-Dispatcher-Thread* erst später erfolgt. Für die Wartezeit kann mit *WorkflowDataModel.setTimeout(millis)* ein generelles Timeout definiert werden.

2.2.5 Workflow-Tasks

Ein Task muss das *WorkflowTask* Interface implementieren, um in einem Workflow laufen zu können. Dieses schreibt im Wesentlichen Methoden vor, die dem Rendern des Task-Labels in bestimmten Zuständen dienen. Nichtsdestotrotz können beliebige *AsyncExecutable* Instanzen in den Workflow eingestellt werden. Sie werden automatisch in einen *ExecutorWorkflowTask* eingebunden, der Default-Implementierungen für *WorkflowTask* besitzt.

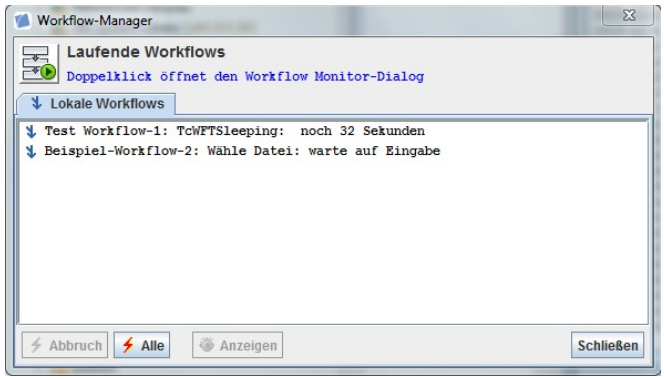
Es gibt eine Fülle von Standard-Workflowtasks für die unterschiedlichsten Zwecke. Sie sind in der Javadoc-Dokumentation im Package *stdwft* zu finden.

2.3 Logging von Workflows

Workflows besitzen einen eigenen *LogChannel*, in den die laufenden Workflow-Tasks loggen können. Da die Menge der Log-Zeilen erheblich werden kann, ist die Größe des Log-Puffers auf einige tausend Zeilen begrenzt.

3 Benutzeroberfläche

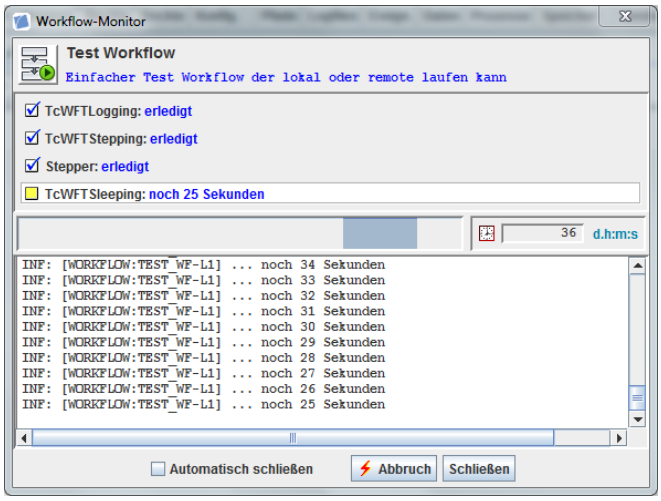
3.1 Workflow-Manager-GUI



Der *WorkflowManagerDialog* dient der Verwaltung laufender Workflows. Ein vom Benutzer gestarteter Workflow erscheint dort als Listeneintrag und verbleibt solange, bis der Workflow abgelaufen ist. Ein selektierter Workflow kann mit Hilfe des Cancel-Buttons abgebrochen werden.

Der Anzeigen-Button öffnet den zugehörigen *WorkflowMonitorDialog*, der den aktuellen Zustand des Workflows vollständig visualisiert.

3.2 Workflow-Monitor-GUI



Das nebenstehende Bild zeigt den *WorkflowMonitorDialog*, die GUI-Implementierung des *WorkflowMonitor* Interfaces.

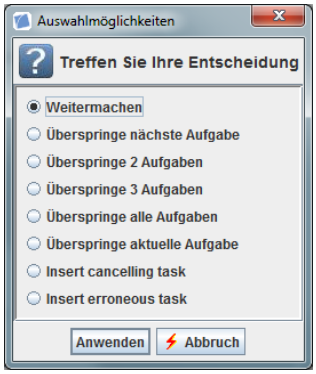
Man sieht die Task-Liste mit bereits erfolgreich durchlaufenen Tasks. Der letzte Task ist gerade in Bearbeitung und treibt den Progress-Bar in diesem Beispiel im *Floating*-Zustand.

Neben dem Progress-Bar befindet sich eine Anzeige, die angibt, wie lange der Workflow schon läuft.

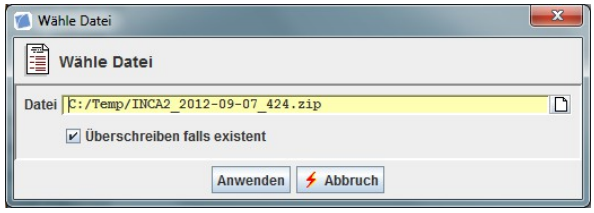
Über die Log-Console berichtet der laufende Task, was er gerade tut. Mit Hilfe des Abbruch-Buttons kann der Ablauf jederzeit abgebrochen werden.

Der Schließen-Button schließt den Monitor, allerdings ohne den Workflow zu beenden! Er läuft vielmehr im Hintergrund weiter. Wenn er fertig ist, erscheint der Monitor automatisch erneut, um dem Benutzer die Fertigstellung anzuzeigen.

3.3 Workflow-Tasks mit Benutzerinteraktion



Das linke Bild zeigt den Eingabedialog eines *Decider*-WorkflowTasks. Er erlaubt das Überspringen bestimmter Tasks, sowie das dynamische Einfügen neuer Tasks.



Das rechte Bild zeigt einen *OutputFileWorkflowTask*, der den Benutzer nach einer Ausgabedatei fragt. Die Benutzereingabe wird im *WorkflowDataModel* gespeichert, so dass sie von nachfolgenden Tasks verarbeitet werden kann.

4 Code Beispiele

4.1 Einfacher lokaler Workflow

Das folgende Beispiel zeigt die Konstruktion eines einfachen Workflows, der drei Tasks beinhaltet. Mit *setCloseMonitorOnCancel(...)* wird bestimmt, dass der *WorkflowMonitor* automatisch schließen soll, falls der Workflow abgebrochen wird. Mit *setWorkingDir(...)* wird das Arbeitsverzeichnis eingestellt. Die eingestellten Task können beliebige *AsyncExecutable*-Instanzen sein – ohne jegliche Besonderheit.

```
public class SimpleWorkflow extends WorkflowMainTask<ApplicationContext> {

  public SimpleWorkflow(ApplicationContext ctx) {
     super(ctx, TrEnsGui.SAMPLE_WF, TrEnsGui.WORKFLOW_WITH_THREE_TASKS);
     setCloseMonitorOnCancel(true);
     setWorkingDir(new File("C:/Temp"));

     addTask(new ExWFTStepping<>(this));
     addTask(new ExWFTLogging<>(this));
     addTask(new ExWFTSleeping<>(this, 10));
  }
}
```

Da der Workflow selbst ein *AsyncTask* ist, kann er einfach mit *submit()* gestartet werden:

```
new SampleWorkflow(ctx).submit();
```

Sobald der Workflow gestartet ist, erscheint der *WorkflowMonitor*-Dialog und der Ablauf kann visuell verfolgt werden.

4.2 Workflow mit Benutzer-Interaktion

Das folgende Beispiel zeigt einen Workflow mit Entscheider-Task. Dieser Task öffnet einen Benutzerdialog, der die angegebenen Optionen zur Entscheidung anbietet. Siehe Bild weiter oben.

```
public class InteractiveWorkflow extends WorkflowMainTask<ApplicationContext> {

  public InteractiveWorkflow(ApplicationContext ctx) {
     super(ctx, TrEnsGui.GUI_WF, TrEnsGui.GUI_WF_WITH_DECIDER);

     DeciderWFTask<C> decider = new DeciderWFTask<>(this);
     decider.addOptions(DeciderStandardOption.values());
     decider.addOption(ExInsertTaskOption.INSERT_OTHER);

     addTask(decider);
     addTask(new ExWFTSleeping<>(this, 10));
  }

  @Override
  public void handleDecision(CheckOption option) {
     if (option == ExInsertTaskOption.INSERT_OTHER ) {
        insertTask(new ExWFTLogging<>(this));
     }
  }
}
```

Die *DeciderStandardOptions* (CONTINUE, SKIP, SKIP_ALL) werden intern behandelt. Die kundenspezifische Option INSERT_OTHER wird durch Überschreiben der Methode *handleDecision(...)* bedient. Hier wird an der betreffenden Stelle dynamisch ein neuer Task eingefügt.

4.3 Beispiel einer WorkflowTask Implementierung

Im Allgemeinen kann ein Workflow jede beliebige *AsyncExecutable*-Instanz als *WorkflowTask* ausführen. Es bietet aber Vorteile, wenn der Task das *WorkflowTask* Interface implementiert bzw. von *AbstractWorkflowTask* ableitet. Dann nämlich können eine Reihe von Methoden überschrieben werden, die die Darstellung des Tasks im Workflow-Monitor beeinflussen. Im folgenden Beispiel wurde *getSucceededLabel()* überschrieben, so dass nach Ablauf des Tasks angezeigt wird, wie die Benutzer-Entscheidung ausgefallen ist. Entsprechende Methoden gibt es für alle anderen Zustände, die ein Workflow-Task annehmen kann: *getReadyLabel(), getStartedLabel(), getProceededLabel()* und *getFailedLabel()*.

```
public class AskForProceedingWFTask extends AbstractWorkflowTask<ApplicationContext> {

  private boolean wasConfirmed;

  public AskForProceedingWFTask(WorkflowMainTask<ApplicationContext> workflow) {
     super(workflow, new RawNlsKey("Confirm me"));
  }

  @Override
  public WorkflowTaskLabel getSucceededLabel() {
     return new WorkflowTaskLabel(getNameKey(), "wasConfirmed=" + wasConfirmed);
  }

  @Override
  public void executeAsync() throws Exception {
     invokeAndWait(() -> askForConfirmation());
     // there may be more code coming here...
  }

  private void askForContinue() {
     if (!ConfirmDialog.confirm("Do you want more?")) {
        throw new StopException();
     }
     wasConfirmed = true;
  }
}
```

In seiner *executAsync()* Methode ruft dieser Task einen *ConfirmDialog* auf. Da dies nur der *Event-Dispatcher-Thread* des GUIs tun darf, geschieht dieser Aufruf mit Hilfe von *invokeAndWait(...)*. An dieser Stelle blockiert der Workflow so lange bis der *ConfirmDialog* bedient wurde. Falls er nicht vom Benutzer bestätigt wurde, wird eine *StopException* geworfen, die dafür sorgt dass der Task sofort (ohne Fehler!) beendet und nachfolgender Code nicht mehr ausgeführt wird.

5 Remote Workflows

Eine wichtige Forderung ist, dass der identische Workflow wahlweise auf Client- oder auf Serverseite laufen kann. „Remote“ ist ein Workflow dann, wenn er auf dem Server abläuft.

➔ Da der Server keine Benutzeroberfläche besitzt, darf ein remote ablaufender Workflow keine Workflow-Tasks besitzen, die eine Benutzerinteraktion erfordern.

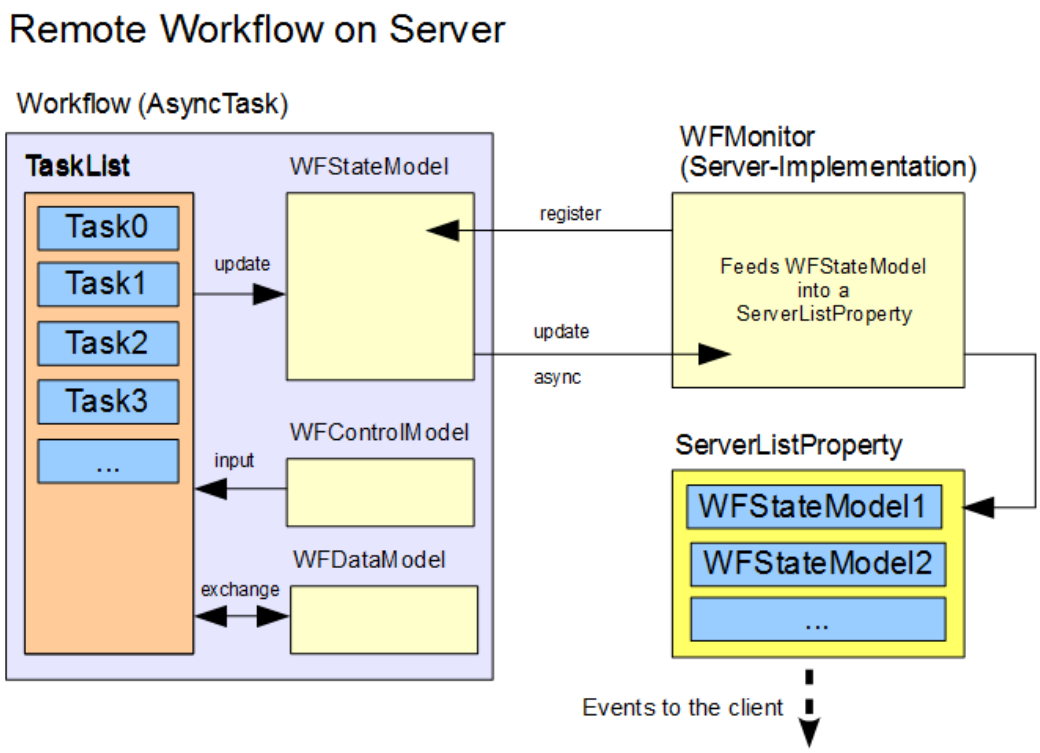
Der remote ablaufende Workflow kann auf Clientseite exakt genauso beobachtet werden, wie ein lokal Laufender. Wenn der Client die Verbindung zum Server trennt, läuft der Workflow auf dem Server weiter. Geht der Client wieder online, dann sieht er den noch laufenden Workflow wieder, als ob nichts geschehen wäre.

Sollte ein remoter Workflow fehlschlagen, während der Client offline war, dann wird er auf Serverseite solange vorgehalten, bis der Client sich wieder meldet. Das Gleiche gilt für remote Workflows, die vom Administrator gecancelled wurden. Der fehlerhafte bzw. abgebrochene Workflow wird dann automatisch angezeigt und muss vom Benutzer bestätigt werden.

Bei der Client/Server-Kommunikation bezüglich Workflows kommen die sogenannten *RemoteListProperties* des Esprit-Frameworks zum Einsatz. Diese sind in folgendem Dokument beschrieben:
http://www.esprit-systems.de/downloads/esprit/docu/EspritRemotePropertyBindingDE.pdf

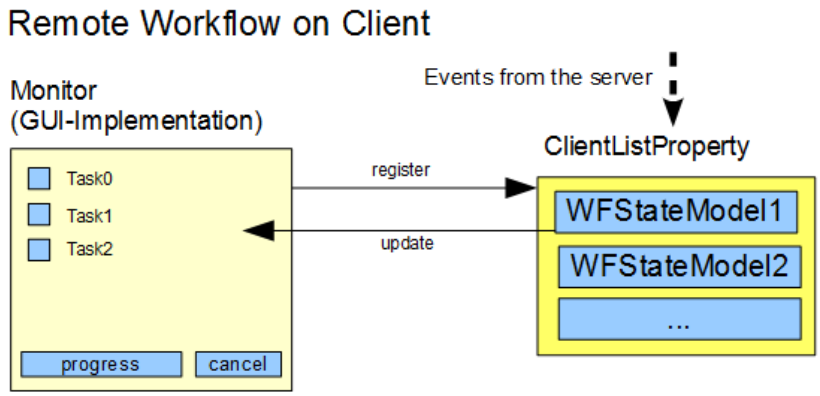
5.1 Funktionsweise serverseitig

Das folgende Bild zeigt die serverseitige Ausführung eines Workflows. Hier ist der *WorkflowMonitor* als ein *Delegator* implementiert, der das *WorkflowStateModel* nach jeder Änderung in einer *ServerListProperty* aktualisiert. Letztere beinhaltet sämtliche *WorkflowStateModels* von allen gerade auf dem Server laufenden Workflows. Dies können viele seine und sie können von unterschiedlichen Benutzern ausgeführt werden.



5.2 Funktionsweise clientseitig

Auf der Clientseite befindet sich die zugehörige *ClientListProperty*, die automatisch die jeweils aktuellsten Zustände der *WorkflowStateModels* reflektiert. Ein Benutzer sieht sämtliche von ihm gestarteten Workflows in einer UI-Liste im *WorkflowManagerDialog* (getrieben von der *ClientListProperty*). Ein Doppelklick öffnet die clientseitige Implementierung des *GUI-WorkflowMonitors*. Dieser registriert sich bei der *ClientListProperty*, um die jeweiligen Aktualisierungen des selektierten *WorkflowStateModels* zu erhalten.



Die *ListProperty* ist eingestellt auf *SessionControlMode.USER_BASED*. Dies bedeutet, dass nur solche Workflows angezeigt werden, die von dem betreffenden Benutzer gestartet wurden.

Zusätzlich arbeitet sie im *ActivationMode.ON_LOGIN*. Dies bedeutet, dass die *ClientListProperty* bei jedem Login reinitialisiert, und somit automatisch die aktuell laufenden Workflows anzeigt. Dies funktioniert auch nach einer zeitweiligen Trennung des Clients.

5.3 Start des remoten Workflows

Der remote Workflow wird über einen speziellen *WorkflowStartAgent* gestartet. Dieser benötigt eine *WorkflowCreator*-Instanz, die in der Lage ist, den gewünschten Workflow auf Serverseite zu instantiieren. Der Start-Agent ist ein asynchroner Agent vom Type *NO_RETURN*. Deshalb läuft der Workflow weiter, auch wenn der Client sich vom Server trennt. Sämtliches Feedback über den laufenden Workflow geschieht über die betreffende *ClientListProperty*.

5.4 Terminierung des remoten Workflows

Ähnlich wie ein lokaler Workflow kann ein remoter Workflow in drei verschiedenen Zuständen terminieren:

- SUCCEEDED
  In diesem Falle ist das Verhalten gleich, wie beim lokalen Workflow. Falls der GUI-Monitor zum Zeitpunkt der Fertigstellung nicht sichtbar war, wird er automatisch angezeigt, um dem Benutzer die Fertigstellung zu melden.
- CANCELLED
  Dies ist in der Regel die Folge einer Aktion des Benutzers selbst. Im remoten Falle kommt jedoch hinzu, dass auch der Server-Administrator den Workflow abbrechen kann. In diesem Falle erhält der Benutzer eine entsprechende Nachricht vom Administrator und der GUI-Monitor öffnet sich, um die Stelle anzuzeigen, wo der Workflow abgebrochen wurde.

➔ FAILED
Ist ein Workflow fehlgeschlagen, so wird sich in jedem Falle der GUI-Monitor öffnen, um den Fehler anzuzeigen. Ein besonderer Fall liegt vor, wenn der Client in diesem Moment offline ist. Dann wird der Workflow auf Serverseite vorgehalten, bis der Client sich wieder meldet. Sodann wird ihm der Fehler angezeigt und der Benutzer muss ihn aktiv bestätigen.

Ein abgelaufener remoter Workflow wird auf Serverseite weiterhin vorgehalten, bis er vom Client bestätigt wurde – sprich: bis er explizit von einem *WorflowDisposeAgent* beseitigt wurde. Dies geschieht implizit beim clientseitigen Schließen des *WorkflowMonitors* durch den Benutzer.

5.5 Logging von remoten Workflows

Remote Workflows loggen selbstverständlich auf Serverseite. Die Logmeldungen sollen aber ebenso auf Clientseite in der Log-Console des GUI-Monitors ausgegeben werden. Logmeldungen werden nicht im *WorkflowStateModel* übertragen. Dies wäre sehr ineffizient, da je nach angefallenen Logs mit jeder Aktualisierung mehr und mehr Daten übertragen werden müssten.

Stattdessen besitzt ein remoter Workflow mit dem *ServerWorkflowLogChannel* eine besondere *LogChannel*-Implementierung, die die Logmeldungen in einer speziellen temporären *ServerListProperty* einträgt. Auf Clientseite gibt es eine entsprechende *ClientListProperty*, die ihrerseits die Log-Console des GUI-Monitors befüttert.

Geht ein Client, nachdem er einen remoten Workflow abgesetzt hat, kurzzeitig offline und verbindet sich erneut, dann initialisiert er sich auf **zwei** *ServerListProperties*: die, die das aktuelle *WorkflowStateModel* beinhaltet und die, die die zugehörigen Logmeldungen beinhaltet. Bei der Initialisierung werden einmalig alle bisher aufgelaufenen Logmeldungen übertragen.

Da Logmeldungen ggf. in Massen anfallen, können sie einen heftigen Message-Verkehr auslösen. Deshalb werden sie vom Server nicht einzeln verschickt, sondern in sog. Chunks. Diese werden in kurzen regelmäßigen Zeitabständen versendet und übertragen jeweils die bis dahin angefallene Menge an Logmeldungen auf einmal.

5.6 Code-Beispiel

Zum Starten eines Workflows auf Serverseite muss eine *WorkflowCreator*-Instanz für den *WorkflowStartAgent* bereitgestellt werden. Dies kann auf einfachste Weise als Lambda-Expression geschehen. Das folgende Beispiel zeigt die *actionFired(...)* Methode eine GUI-Action, die den bereits gezeigten *SimpleWorkflow* serverseitig startet:

```
@Override
protected void actionFired(ActionEvent e) throws Exception {
   TaskId taskId = getClientContext().startRemoteWorkflow((s) -> new SimpleWorkflow(s));
}
```

Innerhalb der Methode *startRemoteWorkflow(...)* wird automatisch ein WorkflowStartAgent erzeugt und zum Server geschickt. Als Ergebnis erhält man eine *TaskId*, die zum Abbruch des serverseitigen Tasks mit Hilfe des *WorkflowCancelAgents* verwendet werden kann.