Hintergrundverarbeitung mit dem Eclipse Job API

Arbeit für alle

VON PETER FRIESE

Ob eine Anwendung von den Anwendern akzeptiert wird, hängt zu einem großen Teil von ihrem Erscheinungsbild ab. Eine gut strukturierte Oberfläche zählt ebenso dazu wie das Antwortverhalten der Applikation. Wer hat das nicht schon selbst erlebt: Muss man länger als ein paar Sekunden auf das Ergebnis einer Interaktion warten, so zuckt der Finger schnell zum Abbrechen-Button. Für die Lösung dieses Problems gibt es zwei grundlegende Strategien.

Zunächst könnte man versuchen, die Performance der Anwendung zu verbessern. Dies ist jedoch nicht immer möglich – insbesondere, wenn das Netzwerk der Flaschenhals ist. Nicht nur in diesem Fall kann es sinnvoll sein, Verarbeitungslogik in den Hintergrund zu verlagern und den Anwender im Vordergrund weiterarbeiten zu lassen. Technisch wird dieser Ansatz üblicherweise mithilfe von Threads gelöst. Die Programmierung von Threads ist jedoch nicht jedermanns Sache. Verfügt man zudem nicht bereits über ein geeignetes Framework für die Hintergrundverarbeitung, muss man viele grundlegende Mechanismen (z.B. Fortschrittsanzeige und sonstiges Feedback) selbst implementieren.

Ende 2003 stand man mit der Eclipse-Plattform vor dem Problem, dass viele Plug-ins länger dauernde Verarbeitungen durchführten, wodurch das GUI oft für Benutzerinteraktionen blockiert war. Zwei Beispiele sind das Kompilieren großer Workspaces („Scrubbing workspace, please wait“ ist wohl eine der am meisten gefürchteten Meldungen geworden) und das Herunterladen von Projekten aus einem Repository. Beide Vorgänge dauern relativ lange und verlangen eigentlich keine Interaktion mit dem Anwender, der untätig vor dem Bildschirm sitzt und vom Fortschrittsbalken hypnotisiert wird. Im Eclipse Project 3.0 Plan [1] wird *Responsive UI* daher auch als eines der vorrangigen Ziele für die Eclipse 3.0-Plattform genannt. In diesem Plan wird auch schon die grundlegende Idee für die Umsetzung genannt: Eclipse soll asynchrone Hintergrundverarbeitung unterstützen. Die Umsetzung dieser Idee hat der Community das Eclipse Job API beschert und dazu geführt, dass Eclipse 3.x User deutlich entspannter vor ihren Bildschirmen sitzen. Dieser Artikel beschreibt die Grundzüge des API und zeigt anhand einiger Beispiele aus der Eclipse-Plattform und aus anderen Projekten, wie man das Job API in eigenen Plug-ins gewinnbringend einsetzen kann.

Kollegen

Bevor mit der Arbeit begonnen werden kann, sollten die Verantwortlichkeiten geregelt werden. Die Beteiligten am Job API

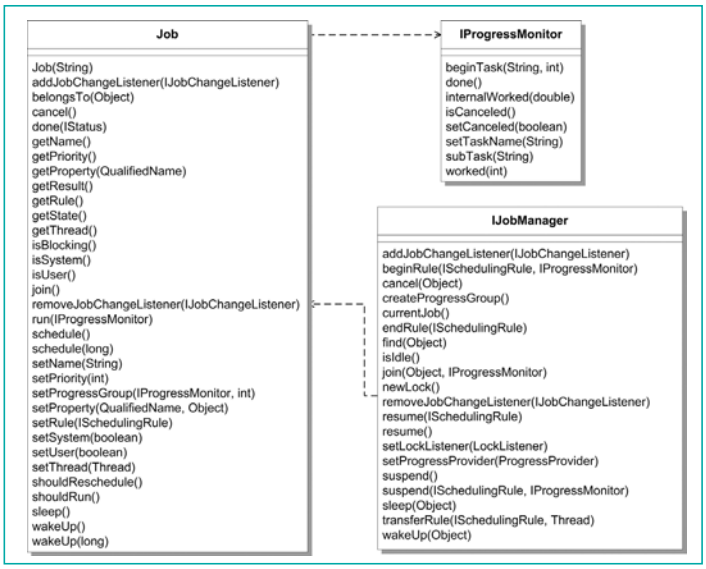


Abb. 1: Die wichtigsten Klassen des Job API

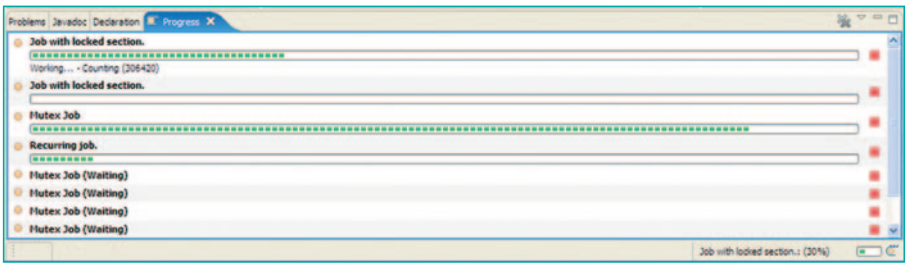


Abb 2: Die Progress View zeigt alle laufenden User Jobs an.

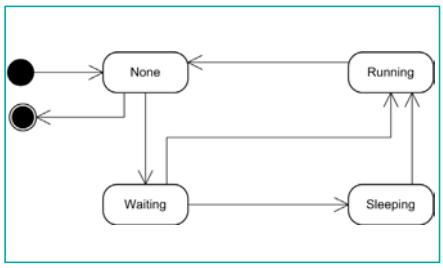


Abb. 3: Mögliche Zustände von Jobs

sind die einzelnen Jobs, der *JobManager* sowie das über das Interface *IProgressMonitor* erreichbare GUI (Abb. 1). Die Jobs führen die eigentliche Arbeit aus. Sie melden sich beim *JobManager* an und können mithilfe eines *IProgressMonitor* eine Rückmeldung über den aktuellen Fertigstellungsgrad der Arbeit geben. Ob und wie der Fortschritt auf dem GUI dargestellt wird, hängt von einer Vielzahl von Faktoren ab. Sofern der Anwender seine Workbench nicht umkonfiguriert hat, sieht er nur einen kleinen Fortschrittsbalken sowie einen Button am unteren rechten Rand des Hauptfensters. Durch einen Klick auf diesen Button bzw. über WINDOW | SHOW VIEW | PROGRESS VIEW kann der Anwender die Progress View öffnen, die Details zu den derzeit laufenden Jobs anzeigt (Abb. 2).

Jobs haben einen Lebenszyklus, der in Abbildung 3 dargestellt wird. Die dazugehörigen Zustandsübergänge können Tabelle 1 entnommen werden. Der Anwender selbst kann in den Ablauf eines Jobs eingreifen und je nach Art des Jobs die Beendigung der Verarbeitung anfordern. Eine vom Benutzer ausgelöste Pausierung ist hingegen nicht vorgesehen, eine entsprechende Feature Request wurde bereits im Eclipse Bugzilla eingetragen [7].

Arbeitsbeginn und -ende

Ein Job kann an jeder beliebigen Stelle im Programmablauf gestartet werden. Im Gegensatz zu den meisten anderen Eclipse APIs gibt es beim Job API keinen Extension Point, an den man sich andocken müsste. Das bedeutet: Ein Job kann nicht nur an jeder beliebigen Stelle im Programm gestartet werden, sondern man kann einen Job an jeder beliebigen Stelle definieren. Listing 1 zeigt einen einfachen Job, welcher der Einfachheit halber direkt aus einer Action gestartet wird.

Beim Erzeugen eines Jobs wird der Name festgelegt, mit dem der Job im GUI angezeigt wird (in unserem Beispiel "Hello Job"). Dieser Name muss zwar auf jeden Fall gesetzt werden, braucht aber nicht eindeutig sein. Hat ein Job seine Arbeit beendet oder wurde er abgebrochen, so muss er dem *JobManager* seinen Status melden. Dies geschieht durch Rückgabe des entsprechenden Statuscodes. Im Erfolgsfall sollte dies auf jeden Fall *Status.OK_STATUS* sein. Für den Fall, dass der Job abgebrochen wurde, sollte hier der Wert *Status.CANCEL_STATUS* zurückgeliefert werden. Der Abbruch eines Jobs kann entweder durch den Benutzer oder durch das Programm herbeigeführt werden: Im GUI besitzt jeder Job einen ABBRECHEN-Button. (In der Progress View ist dies eine rote STOP-Schaltfläche, im Progress-Dialog handelt es sich um einen CANCEL-Button.) Programmatisch kann ein Job durch den Aufruf der Methode *cancel()* abgebrochen werden. Wann ein Job tatsächlich abgebrochen wird, hängt von seiner Implementierung ab – so kann der Job einen günstigen Zeitpunkt für den Abbruch der Arbeit festlegen und gegebenenfalls noch Aufräumarbeiten durchführen, bevor er tatsächlich terminiert.

Tritt während der Verarbeitung eines Jobs eine Exception auf, so wird der Job abgebrochen und ein Fehlerdialog angezeigt. Endet ein Job ohne Fehler, so verschwindet er aus dem GUI: Eclipse schließt einen eventuell offenen Fortschrittsdialog und entfernt den Fortschrittsbalken des Jobs aus der Progress View. Soll nach Beendigung des Jobs eine Statusmeldung in der Progress View verbleiben, so muss der Entwickler eine entsprechende Eigenschaft des Jobs setzen: *job.setProperty*

Auslöser	Zustand/ Übergänge
Job wird erzeugt	*NONE*
Aufruf von job.*schedule()*	*NONE > WAITING*
Job Manager startet den Job	*WAITING > RUNNING*
Job läuft	*RUNNING*
Job endet	*NONE*
Job wird schlafen gelegt	*SLEEPING*
Job wird aufgeweckt	*WAITING*

Tabelle 1: Zustandsübergänge

Listing 1

Ein einfacher Job

```
public class HelloJobAction extends AbstractJobAction {
 public void run(IAction action) {
  new Job("Hello Job") {
   protected IStatus run(IProgressMonitor monitor) {
    SimpleLogger.log ("Job [" + getName() + "]
                                    has been started.");
    return Status.OK_STATUS;
   }
  }.schedule();
 }
}
```

Listing 2

Wiederkehrender Start eines Jobs

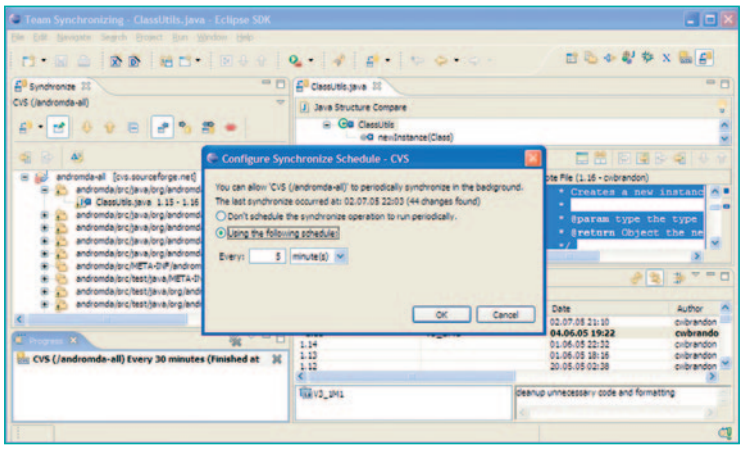
```
protected IStatus run(IProgressMonitor monitor) {
 monitor.beginTask("Working...", 100);
 //...
 monitor.done();

 // reschedule to be run in 10 seconds:
 this.schedule(10000L);
 return Status.OK_STATUS;
}
```

(IProgressConstants.KEEP_PROPERTY, Boolean.TRUE);.

Nachdem ein Job seine Arbeit erledigt hat, kann er neu gestartet werden. Durch die Angabe einer Wartezeit beim Starten bzw. Neustarten eines Jobs lassen sich regelmäßig wiederkehrende Abläufe realisieren (Listing 2). Dieses wird z.B. vom Eclipse CVS Plug-in eingesetzt: Hier kann man einstellen, dass bestimmte Projekte in regelmäßigen Abständen mit dem CVS Server synchronisiert werden (Abb. 4).

Abb. 4: Konfiguration der regelmäßigen CVS-Synchronisation



Arbeitskontrolle

Durch die Verlagerung der Arbeit in den Hintergrund verschwindet der Job zunächst aus der Aufmerksamkeit des Anwenders. Das Job API sieht Mechanismen vor, mit denen ein Job den Anwender über den Fortschritt der Arbeit auf dem Laufenden halten kann. Der zentrale Zugriffspunkt für das Melden des Arbeitsfortschritts ist der Progress-Monitor. Jeder Job hat Zugriff auf einen Progress-Monitor – er wird beim Start des Jobs als Parameter der Methode *run()* übergeben. Über diesen Monitor kann nicht nur der prozentuale Fortschritt gesetzt werden, sondern auch der Name des aktuellen Vorgangs. Außerdem kann man über die Methode *subTask()* einen Teilschritt einleiten. Ob der Benutzer einen Job abbrechen möchte, wird übrigens auch über den Progress-Monitor signalisiert – der Job sollte also in regelmäßigen Abständen die Methode *isCanceled()* aufrufen und zeitnah auf einen vom Benutzer gewünschten Abbruch reagieren (Listing 4).

Listing 3

Zentrale Verwaltung eines Lock

```
public class JobsPlugin extends AbstractUIPlugin {

 //...

 private ILock lock = Platform.getJobManager().
                                        newLock();
 public ILock getLock() {
  return lock;
 }

 //...

}
```

Bestimmte Jobs dürfen aber nicht abgebrochen werden, da sie für den Ablauf der Applikation essenziell sind. Durch den Aufruf der Methode *setSystem()* kann ein Job solchermaßen ausgezeichnet werden. System Jobs erscheinen nicht im GUI, es sei denn, der Anwender aktiviert die Option SHOW SLEEPING AND SYSTEM TASKS in den Voreinstellungen der Progress View. Das Gegenteil eines System Jobs ist ein User Job. Jeder Job kann durch Aufruf der Methode *setUser()* zum User Job gemacht werden. Er erscheint dann zunächst in einem Dialog, der durch Klicken des Buttons RUN IN BACKGROUND in den Hintergrund gesendet werden kann (Abb. 5). Hat der Anwender jedoch die Einstellung ALWAYS RUN IN BACKGROUND in den Eclipse-Voreinstellungen aktiviert, laufen auch User Jobs sofort im Hintergrund ab.

Unter Umständen ist es erforderlich, dass ein Teil eines Programms benachrichtigt wird, sobald ein Job seine Arbeit beendet hat. Hier gibt es zwei Möglichkeiten: Durch einen Aufruf der Methode *join()* kann ein beliebiger Teil eines Programms sich an die Ausführung eines Jobs ankoppeln. Der Aufrufer blockiert dann solange,

Listing 4

Job mit durch ein Lock geschützter Region

```
package de.eclipsemagazin.jobs.jobs;

import org.eclipse.core.runtime.IProgressMonitor;
import org.eclipse.core.runtime.IStatus;
import org.eclipse.core.runtime.Status;
import org.eclipse.core.runtime.jobs.ILock;
import org.eclipse.core.runtime.jobs.Job;

import de.eclipsemagazin.jobs.JobsPlugin;
import de.eclipsemagazin.jobs.util.SimpleLogger;

public class LockedSectionJob extends Job {

 public LockedSectionJob(String name) {
  super(name);
  lock = JobsPlugin.getDefault().getLock();
 }

 public IStatus run(IProgressMonitor monitor) {
  monitor.beginTask("Working...", 100);
  SimpleLogger.log("Job [" + getName() + "]
                    has been started. Prio [" + getPriority() + "] State:
                    " + getState() + " Thread: " + getThread());

  lock.acquire();
  boolean aborted = false;
  int work = 0;
  while (!aborted) {
   if (monitor.isCanceled()) {
    aborted = true;
   }
   work++;
   monitor.worked(1);
   monitor.subTask("Counting (" + work + ")");
   if (work > 100) {
    aborted = true;
   }
  }
  monitor.done();
  lock.release();
  return Status.OK_STATUS;
 }
}
```

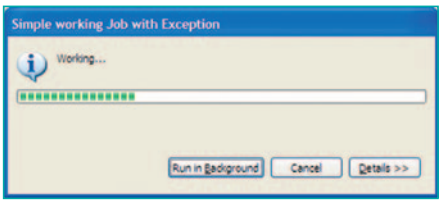


Abb. 5: Der Fortschrittsdialog kann mit RUN IN BACKGROUND in den Hintergrund gesendet werden.

bis der Job seine Arbeit beendet hat. Sinnvoller als dieses synchrone Warten ist jedoch, einen *IJobChangeListener* bei einem Job anzumelden: Erstens wird nicht der Programmablauf blockiert und zweitens kann man sich über alle Lebenszyklusstadien eines Jobs unterrichten lassen. Um nicht alle Methoden des Interface implementieren zu müssen, kann man auch die Klasse *JobChangeAdapter* verwenden, die leere Standardmethoden für das Interface *IJobChangeListener* implementiert. So muss nur die jeweils passende Methode überschrieben werden.

Arbeitskoordination

Die Parallelisierung von Arbeit kann leicht zu Konflikten führen. Auch beim Einsatz von Jobs muss dieser Aspekt bedacht werden. Erfreulicherweise enthält das Job API gleich zwei Ansätze für das Management von Engpässen: Locks und Scheduling Rules. Zunächst einmal muss analysiert werden, wo Engpässe auftreten könnten: Kann z.B. auf eine Ressource nur exklusiv zugegriffen werden? Beispiele hierfür wären das Ändern eines Datensatzes in einer Datenbank oder das Löschen einer Datei. Andererseits könnte auch der Fall eintreten, dass eine von einem Job verwendete Bibliothek nicht Multithreadingfähig ist. Diese Situation entsteht vor allem dann, wenn die Konfiguration der Bibliothek über ein Singleton vorgenommen wird. Soll die Bibliothek dann in einem Plug-in eingesetzt werden, um unterschiedlich konfigurierte Eclipse-Projekte zu bearbeiten, ist eine Sequenzialisierung der Abläufe notwendig: Die Bibliothek muss dann vor jeder Verwendung neu initialisiert werden.

Um den exklusiven Zugriff auf eine (oder mehrere) Ressourcen sicherzustellen, verwendet man ein Lock. Der *JobManager* verwaltet alle Locks und stellt für die Erzeugung eines neuen Locks die Methode *newLock()* zur Verfügung. Ein Lock kann genutzt werden, um Critical Sections zu implementieren [6]: Hat ein Prozess den durch ein Lock geschützten Bereich betreten, garantiert der *JobManager*, dass kein anderer Prozess diesen Bereich betreten wird, bevor nicht der erste Prozess den Bereich verlassen hat. Damit das funktionieren kann, müssen alle betroffenen Jobs Zugriff auf das identische Lock haben (Listings 3 und 4).

Reicht ein Lock zur Reglementierung des Ressourcenzugriffs nicht aus, kann mittels Scheduling Rules ausgedrückt werden, ob ein Job mit einem anderen kollidiert. Stehen zwei Jobs nicht im Konflikt miteinander, können sie parallel zueinander ausgeführt werden. Stellt der *JobManager* einen Konflikt fest, so verzögert er den Start des zweiten Jobs so lange, bis der erste seine Arbeit beendet hat. Erst wenn kein Konflikt mehr festgestellt werden kann, wird der neue Job gestartet.

Eine Scheduling Rule kann in einer anderen Rule enthalten sein – so können hierarchische Abhängigkeiten ausgedrückt werden. Listing 5 zeigt eine einfache Regel, die mit sich selbst in Konflikt steht. Durch eine solche Regel kann die parallele Ausführung mehrere Instanzen eines Jobs effektiv verhindert werden (Listing 6).

Für die Steuerung des Zugriffs auf alle Elemente, die das Interface *IResource* implementieren, müssen keine eigenen Scheduling Rules geschrieben werden, da alle von *Resource* abgeleiteten Klassen (d.h. alle Projekte, Ordner, Dateien und Klassen eines Workspace) automatisch auch Scheduling Rules sind. Mittels einer Factory können sehr einfach passende Scheduling Rules für die gebräuchlichen Ressourcenoperationen (Erzeugen, Löschen, Umbenennen, Ändern, Verschieben, Aktualisieren etc.) erzeugt werden:

```
IResourceRuleFactory ruleFactory = workspace.
                                        getRuleFactory();
modifyFileJob.setRule(ruleFactory.modifyRule(file);
```

Während Locks vorwiegend dazu genutzt werden, den Zugriff auf eine geteilte Ressource zu regeln, wird durch die Verwendung von Scheduling Rules bereits verhindert, dass mehrere miteinander möglicherweise in Konflikt stehende Jobs starten können.

Werksbesichtigung

Nachdem wir uns nun die grundlegenden Aspekte des Job API zu Gemüte geführt haben, ist es an der Zeit, sich ein paar Beispiele aus dem echten Leben anzuschauen. Dabei wird sowohl Eclipse selbst Modell stehen als auch das ein oder andere Plug-in sowie eine Enterprise-Applikation.

Der wohl prominenteste Job ist *AutoBuildJob* – er sorgt gemeinsam mit dem *BuildManager* dafür, dass der Workspace nach jeder Änderung einer Ressource neu gebaut wird. Prinzipiell ist es eine gute Idee, den Workspace immer im lauffähigen Zustand zu halten und alle Dateien bereits beim Speichern zu kompilieren. Leider dauert das Bauen eines großen Workspace recht lange, sodass die Verwendung eines modalen Fortschrittsdialogs (so wie es in allen Eclipse-Versionen vor 3.0 der Fall war) eine schlechte Idee ist. Als Ausweg aus diesem Dilemma wird in diversen Foren und Artikeln [2] empfohlen, das

Listing 5

***MutexRule* steht mit sich selbst in Konflikt**

```
public class MutexRule implements ISchedulingRule {
 public boolean isConflicting(ISchedulingRule rule) {
  return rule == this;
 }
 public boolean contains(ISchedulingRule rule) {
  return rule == this;
 }
}
```

Listing 6

Scheduling Rules vor Aufruf von *schedule()*

```
public class MutexJobAction extends
                                    AbstractJobAction {
 private static MutexRule mutextRule =
                                        new MutexRule();
 public void run(IAction action) {
  MutexJob mutexJob = new MutexJob("Mutex Job");
  mutexJob.setRule(mutextRule);
  mutexJob.schedule();
 }

}
```

AutoBuild-Feature einfach abzuschalten und das Kompilieren manuell zu starten. Ab Eclipse 3 gehören diese Probleme aber der Vergangenheit an – dank des Job API. Genau wie in den Vorgängerversionen gibt es auch in Eclipse 3 das Konzept der Incremental Project Builder, diese werden jedoch im Kontext des *JobManager* ausgeführt, der den gesamten Build-Prozess innerhalb eines Job durchführt. Somit ist es nun möglich, dass der Build im Vordergrund startet und dann vom Anwender in den Hintergrund gesendet wird (mit der Schaltfläche RUN IN BACKGROUND). Das Abbrechen eines Build war schon seit jeher möglich, allerdings scheint es mit Eclipse 3.0 besser zu funktionieren.

Ein weiterer interessanter Anwendungsfall für das Job API ist der *DeferredTreeContentManager*, der unter anderem in der CVS Repositories View verwendet wird: DieseView listet zunächst die Top-Level-Module eines CVS Repository. Klickt man auf das Pluszeichen vor einem solchen Modul, so startet im Hintergrund ein Job, der die jeweiligen Submodule im Repository ermittelt. Während der Job läuft, wird in der Baumansicht ein Dummy-Knoten mit der Aufschrift „Pending…“ angezeigt. Sobald alle Unterknoten geladen sind, wird dieser Knoten durch die geladenen Submodule ersetzt. Da der *DeferredTreeContentManager* ein freigegebenes API ist, kann er auch in eigenen Plug-ins genutzt werden. Ein Nachteil soll nicht verschwiegen werden: Bei einem Refresh (z.B. durch Drücken von F5) kann die entsprechende Tree View den Zustand (auf- oder zugeklappt) der Knoten nicht wieder herstellen. Ursache dafür ist die Reihenfolge der Verarbeitung: In dem Moment, in dem die Tree View normalerweise den zwischengespeicherten Zustand wieder reaktiviert, befinden sich noch nicht die endgültigen Knoten im Baum, sondern die „Pending…“ Dummies.

Die Verwendung von Jobs in der Eclipse CVS-Unterstützung weist den Weg für eigene Anwendungen des Job API: Es eignet sich hervorragend, um lang laufende Kommunikationsvorgänge mit entfernten Systemen zu kapseln. Ein denkbarer Einsatz soll anhand einer Enterprise-Applikation geschildert werden: Es handelt sich hierbei um ein System zur Erfassung von Buchungen und Reservierungen sowie zur Erstellung von Produktionsplänen. Das Backend wurde auf Basis von Spring [3] und Hibernate [4] realisiert und in großen Teilen mit AndroMDA [5] generiert. Das Frontend ist eine Eclipse RCP-Applikation, die mittels des Spring HTTP Invoker Protocol mit dem Backend kommuniziert.

Das System enthält etliche Erfassungsmasken, in denen der Anwender teilweise freie Informationen und teilweise fest vorgegebene Stammdaten eingeben muss. Beim Ausfüllen der Maske kann der Anwender in den entsprechenden Feldern auf die Stammdatenlisten zugreifen, indem er mit CTRL + SPACE die Content-Assist-Funktion aufruft. Um dieses Feature sinnvoll einsetzen zu können, ist es erforderlich, dass die entsprechenden Stammdaten auf dem Client vorliegen. Eine mögliche Strategie wäre, die Daten beim Start der Applikation zu laden, doch leider können sich die Stammdaten im Laufe eines Tages ändern. Also müssen die Stammdaten jeweils beim Öffnen einer Erfassungsmaske geladen werden, was eine kurze Zeit in Anspruch nimmt. Um den Benutzer während des Ladevorgangs nicht zu blockieren, wurde das Laden der Stammdaten in einen Job ausgelagert. Der Benutzer kann direkt nach dem Öffnen des Editors mit der Eingabe von Daten beginnen, während im Hintergrund noch Nachschlagewerte geladen werden. Die meisten Anwender müssen nur selten auf die Nachschlagefunktion zurückgreifen, da sie die erlaubten Werte für die einzelnen Felder auswendig kennen. Würde die Applikation das Laden nicht in einem Hintergrundjob durchführen, sondern in einem modalen Dialog, würden diese Benutzer stark ausgebremst. Mit dem jetzigen Vorgehen werden sie jedoch nicht behindert. Dennoch kann mit der Content-Assist-Funktion auf Basis der nachgeladenen Nachschlagewerte eine Unterstützung für unerfahrene bzw. neue Anwender geboten werden.

Allen drei Beispielen ist gemein, dass sie die volle Handlungsfähigkeit des Anwenders erhalten: Während im Hintergrund ein Job Daten verarbeitet, kann der Anwender mit der Anwendung weiterarbeiten. Im Fall der geschilderten Enterprise-Applikation könnte er z.B. eine weitere Erfassungsmaske öffnen oder eine Suchanfrage starten (die natürlich auch mittels eines Jobs im Hintergrund verarbeitet wird). Auch für Anwender der Eclipse IDE hat sich der Komfort mit Einführung des Job API deutlich erhöht: Während im Hintergrund ein CVS Update läuft und der Workspace neu gebaut wird, kann man z.B. mit dem Update Manager nach der neuesten Version eines bestimmten Plug-ins suchen oder an den bereits aus dem Repository geladenen Dateien weiterarbeiten.

Die Eintrittshürde für die Verwendung von Jobs in eigenen Plug-ins oder RCP-Applikationen liegt relativ niedrig.

Fazit

Das Jobs API bietet umfangreiche Möglichkeiten, Verarbeitungsschritte in den Hintergrund zu verlagern, um so das Ansprechverhalten der Applikation spürbar zu verbessern. Die Eintrittshürde für die Verwendung von Jobs in eigenen Plug-ins oder RCP-Applikationen liegt relativ niedrig und die Investition macht sich sofort bezahlt. Auch komplexe Anforderungen können einfach realisiert werden – insbesondere die Koordination mehrere Jobs ist einfach zu handhaben und erleichtert den Umgang mit dem Job API.

Peter Friese (peter.friese@lhsystems.com, f3.tobject.de) arbeitet als Softwarearchitekt bei Lufthansa Systems in Hamburg. Seine Leidenschaft gilt der Entwicklung von Tools, die das Leben für Entwickler einfacher machen. Peter ist Autor des Findbugs-Plug-ins und AndroMDA-Committer.

Links & Literatur

[1] Eclipse 3.0 Project Plan: www.eclipse.org/eclipse/development/eclipse_project_plan_3_0.html

[2] Eclipse FAQ: www.eclipse.org/eclipse/faq/eclipse-faq.html

[3] Spring Framework: www.springframework.org

[4] Hibernate: www.hibernate.org

[5] AndroMDA: www.andromda.org

[6] Andrew S. Tanenbaum: Modern Operating Systems, Prentice Hall, 2001

[7] Eclipse Bug ID 101352: bugs.eclipse.org/bugs/show_bug.cgi?id=101352