Desktop

Het Swing Application Framework (JSR 296) is een framework dat momenteel nog in ontwikkeling is. Het eerste prototype is echter al veelbelovend en begint langzamerhand rijp genoeg te worden voor de bouw van een nieuwe generatie rijke Swing-applicaties.

Swing Application Framework

Swing is niet moeilijk meer…

Wie al eens met Swing heeft geprogrammeerd, weet dat het niet altijd eenvoudig is om bepaald gewenst userinterface-gedrag te creëren. Vooral voor de beginnende Swing-programmeur is Swing net als het beklimmen van de Mount Everest. Het is immens groot, er zijn te veel mogelijke paden om te bewandelen en het bereiken van de top is een echte uitdaging.
Sinds Java 1.2 maakt Swing onderdeel uit van de Java Standard Edition. Swing-componenten zien er op elk platform hetzelfde uit en gedragen zich ook hetzelfde, in tegenstelling tot de al langer bestaande user interface technologie AWT. De kracht van Swing is de grote flexibiliteit die het framework biedt. De kosten voor deze flexibiliteit zijn complexiteit en een grote mate van abstractie wat belangrijke redenen zijn dat Swing de desktop nog niet heeft weten te veroveren.
Voor veel problemen waar ontwikkelaars tijdens de bouw van een Swing-applicatie mee te maken krijgen, worden standaardoplossingen geboden door het Swing Application Framework (JSR 296). Het framework kan gezien worden als zekeringsapparaat (term uit de bergsport, red.), waarmee het een stuk eenvoudiger wordt om de top van de Mount Everest te bereiken. De huidige planning geeft aan dat het framework onderdeel zal zijn van Java SE 7. Dit artikel laat zien hoe dit zekeringsapparaat eruit ziet en hoe het gebruikt dient te worden.

Standaardoplossingen

In de basis richt het framework zich op een vijftal standaardoplossingen voor bekende probleemgebieden bij Swing-applicaties, te weten:

- De life-cycle van een Swing-applicatie
- Het gebruik van application resources
- Hergebruik van acties
- Het uitvoeren van taken op de achtergrond
- Sessieoverstijgende toestand van applicaties

Voordat wordt ingegaan op deze zaken is het belangrijk om eerst kort de basis van het framework te illustreren. De twee klassen die een centrale rol spelen in het framework zijn de klassen `Application` en `ApplicationContext`. `Application` is de klasse waar je typisch (indirect) van afleidt bij het bouwen van de Swing-applicatie. De `getContext()` methode van de `Application`-klasse geeft de singleton-instantie van de klasse ApplicationContext terug. Deze beheert gedeelde objecten, zoals actions, resources en tasks voor de applicatie.

Application life-cycle

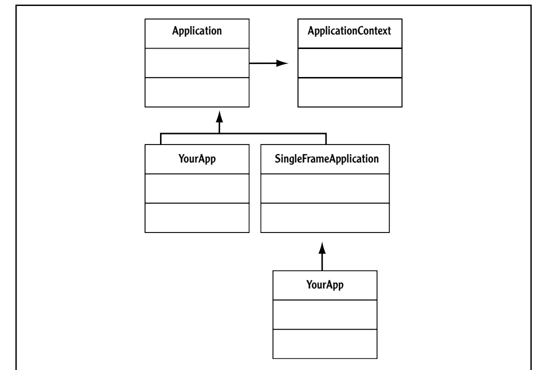
De life-cycle van een Swing-applicatie was tot voor kort nog niet expliciet gemaakt. De komst van de Application-klasse brengt hier verandering in. Deze klasse definieert de life-cycle van een Swing-applicatie in een zestal fasen, te weten: launch, initialize, start-up, ready, exit en shutdown. Dit dwingt af dat dingen op de juiste plaats en tijd gebeuren.
Om gebruik te kunnen maken van life-cycle-management dient de Swing-programmeur een klasse te definiëren die (indirect) is afgeleid van de `Application`-klasse. Indirect, omdat het framework al afgeleide klassen biedt zoals de

Bob Schonenberg is ontwikkelaar bij Info Support B.V. Hij is actief in de Rich Client werkgroep en doet hiervoor onder andere onderzoek naar verschillende Rich Client frameworks. E-mail: bobs@infosupport.com.

SingleFrameApplication-klasse waar je typisch wel direct van afleidt. De statische methode `launch` wordt door de programmeur aangeroepen met twee argumenten. Het eerste argument is van het type `Class<T extends Application>` wat de applicatieklasse is.
Het tweede argument is van het type `String[]` en dient als doorgeefluik voor de `String[] args` die door de `main`-methode wordt aangeboden. Onderstaand voorbeeld toont de aanroep van de launch-methode.

```
public class MyApp extends
    SingleFrameApplication {

  public static void main(String[] args) {
    launch(MyApp.class, args);
  }

  @Override
  public void initialize(String[] args) {
    // initializatie voor bouw van UI.
  }

  @Override
  public void startup() {
    // bouw de UI en toon deze.
  }

  @Override
  public void ready() {
    // de UI is zichtbaar en kan invoer
    // ontvangen.
  }

  @Override
  public void shutdown() {
    // ruim resources op.
  }
}
```

De `launch`-methode van de `Application`-klasse initialiseert de `ApplicationContext` en roept daarna achter elkaar de `initialize`- en de de `startup`-methode aan op de Event Dispatching Thread (EDT).
Initialisatie die moet gebeuren voordat de user interface wordt opgebouwd, dient te worden uitgevoerd in de `initialize`-methode. In de `startup`-methode is het typisch de bedoeling dat de user interface wordt opgebouwd en getoond. Dit gebeurt allemaal automatisch op de EDT, waar de `launch`-methode voor heeft gezorgd. Nadat de user interface getoond is, roept de `Application`-klasse de `ready`-methode aan. Dit geeft aan dat de applicatie gereed is om gebruikersacties te verwerken.
Voor het afsluiten van de Swing-applicatie biedt de `Application`-klasse de `exit`-methode. Dit is geen methode die de programmeur moet overriden, maar deze is aan te roepen zoals bij `launch`. Na aanroep van de `exit`-methode worden alle `ExitListeners` geraadpleegd of het mogelijk is om de applicatie af te sluiten. Als alle ExitListeners instemmen wordt de shutdown-methode aangeroepen en wordt de applicatie afgesloten met `System.exit(0);`.



Context van Application class

ExitListeners

Een applicatie kan niet altijd vanzelfsprekend zomaar afgesloten worden. Zo kan deze nog bezig zijn met het wegschrijven van data naar een bestand op de harde schijf, of met het uitvoeren van een tijdrovende taak op de achtergrond. Om te bepalen of de applicatie mag worden afgesloten is speciaal de `ExitListener`-interface geïntroduceerd. Deze interface biedt de methoden `canExit` en `willExit` waarmee met een two-phase strategie kan worden bepaald of de applicatie mag worden afgesloten. Onderstaand codevoorbeeld is een `ExitListener` die aan de gebruiker vraagt of de applicatie mag worden afgesloten.

```
Class UserExitListener implements
    ExitListener {

  public boolean canExit(EventObject evt) {
    int answer = JOptionPane.
     showConfirmDialog(getMainFrame(),
     message, title,
     JOptionPane.YES_NO_OPTION);

     return (answer ==
       JOptionPane.YES_OPTION);
  }

  public void willExit(EventObject evt) {
    // ruim resources op.
  }
}
```

In eerste instantie zal de applicatie de `canExit()`-methode aanroepen van geregistreerde `ExitListeners`. Als al deze `ExitListeners` instemmen, zal daarna de `willExit()`-methode worden aangeroepen voor eventuele opruimwerkzaamheden.

Application resources

Rijke user interface-applicaties maken vrijwel altijd gebruik van diverse soorten content. Zo kunnen naast afbeeldingen bijvoorbeeld ook geluidsfragmenten gebruikt worden in een applicatie. Tekst is uiteraard de meest toegepaste content in user interface-applicaties. Een slech-

De life-cycle van een Swing-applicatie was tot voor kort nog niet expliciet gemaakt

te, maar jammer genoeg nog wel eens voorkomende manier van werken met teksten is het direct verwerken ervan in de broncode. Dit heeft als nadeel dat een applicatie niet gemakkelijk meertalig kan zijn, en bovendien is het een onderhoudsnachtmerrie. Zo zal voor het wijzigen of verbeteren van tekst een nieuwe versie van de applicatie moeten worden gecompileerd en uitgeleverd.
Standaard biedt Java ondersteuning voor het *externalizen* van deze content met `java.util.ResourceBundle`. Deze ondersteuning is echter beperkt. Het Swing Application Framework levert klassen die het werken met externalized content gemakkelijker maakt en biedt daar bovenop ondersteuning voor:

- Internationalisatie (`ResourceBundle`)
- Typeconversie
- Resource-merging (`ResourceBundle`)
- Resource-injectie

Voor internationalisatie en resource-merging wordt gebruik gemaakt van de functionaliteit die al aangeboden wordt door `java.util.ResourceBundle`. Hierbij worden externe teksten gelezen uit een properties-bestand. Type-conversie is nieuw en zorgt ervoor dat de waarden in een properties-bestand geïnterpreteerd kunnen worden als een ander object dan String, zoals Font of Color. Onderstaand voorbeeld laat zien hoe dit er in de praktijk uitziet.

```
MyForm.properties
aString = Just a string
aMessage = Hello {0}
anInteger = 123
aBoolean = True
anIcon = myIcon.png
aFont = Arial-PLAIN-12
colorRGBA = 5, 6, 7, 8
color0xRGB = #556677
```

Met behulp van de klasse `ResourceMap` kunnen deze waarden vervolgens worden geïnterpreteerd als de bedoelde Java-typen:

```
ApplicationContext c =
  getContext();
ResourceMap r =
  c.getResourceMap(MyForm.class);

r.getString("aMessage", "World") =>
  "Hello World"
r.getColor("colorRBGA") =>
  new Color(5, 6, 7, 8)
r.getFont("aFont") =>
  new Font("Arial", Font.PLAIN, 12)
```

Uiteraard is het bij het gebruik van type-conversie uiterst belangrijk dat er alleen geldige waarden in een properties-bestand staan. Als dit niet het geval is, leidt dit tot ongewenste runtime-exceptions en dus potentieel instabiele code.
De `ResourceMap` kan ook gebruikt worden voor resource-injectie. Dit houdt in dat de eigenschappen van Swing-componenten uitgelezen worden uit een properties-bestand en daarna worden geïnjecteerd in de user interface. Het volgende voorbeeld illustreert het gebruik van resource-injectie.

De teksten en lettertypen van het label en de knop zijn geïnjecteerd vanuit een properties-bestand.

Het bijbehorende MyForm.properties-bestand heeft de volgende inhoud:

```
aLabel.text = Injected!
aLabel.font = Courier-BOLD-12
aButton.text = "My Favorite Button"
aButton.font = Arial-PLAIN-12
```

Het vinden van de juiste component tijdens de injectie gebeurt op basis van de naam van de Swing-component. Dus het `JLabel` in dit voorbeeld heeft de naam 'aLabel' die aan de `JLabel`-instantie wordt toegekend met de `.setName()`-methode van `JComponent`. Hetzelfde geldt voor de `JButton`. Met de volgende code wordt de externe tekst geïnjecteerd in de Swing-componenten.

```
ApplicationContext ctx = getContext();
ResourceMap rm =
  ctx.getResourceMap(MyFrame.class);
rm.injectComponents(frame);
```

Ook hier is het belangrijk dat er alleen geldige waarden in het properties-bestand staan. De resource-injectie-feature is daarmee met name handig voor prototypen, waarbij de inhoud snel en gemakkelijk gewijzigd dient te kunnen worden om zodoende snel een gewenst resultaat te bereiken.

Actions

In Swing wordt het action framework gebruikt voor het uitvoeren van (gebruikers)acties. Het

meest triviale voorbeeld met betrekking tot dit action framework is de ActionListener-interface die onder andere door de JButton gebruikt wordt. Vaak wordt deze interface dan geïmplementeerd door de klasse (JFrame) waarin de JButton gebruikt wordt. Vervolgens wordt aan de methode addActionListener van JButton de this-pointer meegegeven om de ActionListener voor deze JButton te initialiseren. Een andere veel toegepaste manier is het maken van een anonymous inner class die de interface implementeert. Minder vaak wordt een aparte (niet anonieme) klasse gemaakt als ActionListener.

Aan deze manieren van werken kleven diverse nadelen. Als eerste levert het maken van de ActionListener wat overhead. Het resultaat is altijd een klasse, terwijl alleen een methode goed (functie pointer, of delegate) genoeg zou zijn. Daarnaast is de instantie van een ActionListener op deze manieren moeilijk te hergebruiken. De hoofdreden hiervoor is dat de scope van deze instantie vaak niet verder reikt dan de klasse waarin ook de action component (JButton) leeft. De oorzaak ligt hier bij het ontwerp. Gewenst is om een action eenmalig te definiëren, en op meer plaatsen te gebruiken. Zo kan een printactie beschikbaar zijn in een menubar, maar daarnaast ook in een toolbar en in de pop-up van de rechtermuisknop.

Het Swing Application Framework biedt een natuurlijkere manier van het werken met acties, met als basis de @Action-annotatie. Het volgende codevoorbeeld toont het gebruik van deze annotatie bij het definiëren van de sayHello-actie.

```
@Action
public void sayHello(ActionEvent evt) {
  String s = textField.getText();
  JOptionPane.showMessageDialog(s);
}
```

De @Action-annotatie zorgt er voor dat voor deze methode een javax.swing.Action wordt gemaakt die via de ApplicationContext op iedere plek in de applicatie te benaderen is. De returnwaarde van een actiemethode dient void te zijn. Verderop bij het onderwerp Tasks is te zien dat een returnwaarde bij een @Action-methode een speciale betekenis heeft.

```
ApplicationContext c =
  Application.getInstance().getContext();
ActionMap actionMap =
  c.getActionMap(getClass(), this);
Action action = actionMap.get("sayHello");
textField.setAction(action);
button.setAction(action);
```

Standaard wordt de javax.swing.Action-instantie in de ActionMap opgeslagen met als sleutel de naam van de methode. In dit voorbeeld is dat sayHello. Met behulp van de name-parameter van de @Action-annotatie kan hiervan worden afgeweken.

```
@Action(name="hello")
```

Enabling en selectie

Vaak is het noodzakelijk om op basis van een bepaalde conditie een actie te kunnen *enablen/disablen*, of te selecteren/deselecteren. De @Action-annotatie biedt hiervoor ondersteuning met de attributen enabledProperty en selectedProperty. Beide attributen werken op dezelfde manier, namelijk op basis van zogenaamde bound properties. Het volgende voorbeeld maakt de werking voor de enabledProperty duidelijk. Voor de selectedProperty geldt hetzelfde principe.

```
private boolean busy;

@Action(enabledProperty="busy") {
public void cancel() {
  this.doCancel();
}

public void setBusy(boolean busy) {
  boolean oldValue = this.busy;
  this.busy = busy;
  firePropertyChange("busy", oldValue,
    busy);
}

public boolean isBusy() {
  return busy;
}
```

In dit voorbeeld is de cancel-actie alleen enabled als de busy-property de waarde true heeft. Voorwaarde is wel dat de klasse waarin dit gebruikt wordt (indirect) ondersteuning biedt voor property-changes. Voor alle Swing-componenten is dit standaard al geregeld in de klasse JComponent, die de firePropertyChange-methode aanbiedt. Een Swing-component die nu de cancel-actie als action krijgt, zal *enablen en disablen* op basis van de waarde van de busy-property.

Het selectedProperty-attribuut werkt voor Swing-componenten die selected kunnen zijn, zoals een JCheckBox en een JToggleButton.

Tasks

Gedurende de uitvoering van een Action is de user interface niet responsief, aangezien de actie op de Event Dispatching Thread wordt uitgevoerd. Actions zoals tot nu toe behandeld, zijn daarom eigenlijk alleen bedoeld voor user interface-taken, zoals het bijwerken van een label en het verbergen van een tekstveld. Alle andere acties dienen niet op de user interface-thread uitgevoerd te worden, aangezien daarmee de user interface geblokkeerd wordt.

Er is binnen Swing geen voorziening die de toestand van stateful componenten over gebruikersessies heen helpt bewaren

Als oplossing voor dit probleem zijn diverse alternatieven in omloop, zoals de SwingWorker en het open source framework Spin. Sinds Java SE 6 is SwingWorker onderdeel van de JDK. Ook het Swing Application Framework komt met een oplossing voor dit probleem: tasks. De implementatie hiervoor, de `Task` klasse, is direct afgeleid van de SwingWorker.
De onderstaande `Task` roept op een achtergrondthread een webservice aan en geeft het resultaat terug.

```
public class CallWebServiceTask extends
  Task<String, Void> {

  @Override
  protected String doInBackground() {
    // Call webservice ...
    return "result";
  }
}

De CallWebServiceTask kan dan als volgt worden
gebruikt.

@Action {
public CallWebServiceTask callService() {
  return new CallWebServiceTask();
}
```

In het bovenstaande codevoorbeeld van het gebruik van een `Task` is te zien dat dit nagenoeg hetzelfde is als het gebruik van een `@Action`. Het verschil zit hem in de returnwaarde van de methode. Wanneer dit `void` is, wordt de methode op de user interface-thread uitgevoerd. Als de returnwaarde van het type `Task` is, wordt de `doInBackground()`-methode van de `Task` op een aparte thread uitgevoerd.
Naast het gemakkelijk uitvoeren van taken op de achtergrond biedt het framework ook ondersteuning om deze taken te monitoren. Hiervoor kunnen de `TaskMonitor`-klasse en de `TaskListener`-interface gebruikt worden.

Blocking-granulariteit

Een ander veel toegepast principe is om na het activeren van een actie een deel van de applicatie te blokkeren gedurende de uitvoering van de actie. Dit om te voorkomen dat een actie meer dan één keer tegelijkertijd wordt uitgevoerd of om te voorkomen dat meer acties tegelijkertijd door elkaar heen lopen. Als dit wel gebeurt, kan dit leiden tot ongewenste situaties.
De `@Action`-annotatie biedt de mogelijkheid om een bepaald gedeelte van de applicatie waar de actie in wordt uitgevoerd te blokkeren. Hiervoor wordt het `block`-attribuut gebruikt, wat een `enumeratie`-type is. Het `block`-attribuut kan de volgende mogelijke waarden hebben:

- NONE
- COMPONENT
- ACTION
- WINDOW
- APPLICATION

De waarde NONE is de standaardwaarde waarbij geen enkele blokkering optreedt. Met de waarde COMPONENT wordt gezorgd dat de component die de actie heeft geïnitieerd, geblokkeerd (disabled) wordt. Bij de waarde ACTION worden alle componenten geblokkeerd die deze actie hebben, niet alleen de component die de actie heeft geïnitieerd, zoals bij COMPONENT . De waarden WINDOW en APPLICATION zijn op dit moment gelijk aan elkaar. Hierbij wordt een modal dialog getoond die de gehele applicatie blokkeert. Onderstaand codevoorbeeld toont het gebruik van het `block`-attribuut.

```
@Action(block=Block.ACTION)
public Task findMeaningOfLife() {
  return seeker.findMeaningOfLife();
}
```

Hier worden alle componenten geblokkeerd die de `findMeaningOfLife`-actie als action hebben voor zolang de actie duurt.

Session State

Er is binnen Swing geen voorziening die de toestand van stateful componenten over gebruikersessies heen helpt bewaren. Een `JCheckBox` is initieel altijd unchecked, een `JToggleButton` is niet pressed en de kolombreedten van een `JTable` zijn allemaal gelijk.
In bepaalde situaties is het echter wenselijk dat deze state over sessies heen bewaard blijft en bij (her)start van de applicatie wordt teruggezet zoals deze de vorige keer was bij het afsluiten. De klasse `SessionStorage` biedt de methoden `save()` en `restore()` waarmee dit mogelijk wordt gemaakt. Deze zorgt er voor dat de applicatie-state per gebruiker wordt bewaard in een te specificeren XML-bestand.
Een instantie van de `SessionStorage` wordt verkregen bij de `ApplicationContext`. Het aanroepen van de `save()`- en de `restore()`-methoden wordt respectievelijk typisch gedaan in de shutdown- en de startup-fase van de levenscyclus van de applicatie. Onderstaand codevoorbeeld toont deze toepassing.

```
@Override
public void startup() {
  SessionStorage storage =
    appCtx.getSessionStorage();
  storage.restore(getMainFrame(),
    "session.xml");
}

@Override
```

```
public void shutdown() {
  SessionStorage storage =
    appCtx.getSessionStorage();
  storage.save(getMainFrame(),
    "session.xml");
}
```

Het meegeven van het mainframe is noodzakelijk, omdat de methoden een top-level-component verwachten waarvan de child-componenten in het aangegeven bestand `session.xml` bewaard worden. Sessies kunnen per applicatie en per gebruiker bewaard worden. Op een Windows-systeem zal het bestand session.xml terug te vinden zijn in het pad
"`C:\Documents and Settings\<ingelogde-gebruiker>\Application Data\<vendor-naam>\<applicatie-naam>\session.xml`".
Standaard biedt het framework ondersteuning aan een aantal Swing-componenten voor het bewaren van de state. Voor custom Swing-componenten waarvan de state bewaard moet worden, dient de programmeur dit zelf te implementeren met behulp van de aangeboden `Property`-interface.

Eenduidig en gestructureerd

Het Swing Application Framework maakt het bouwen van Swing-applicaties een stuk gemakkelijker. Voor de vijf belangrijkste probleemgebieden bij de bouw van (rijke) Swing-applicaties worden bruikbare en eenvoudig toepasbare oplossingen geboden. De expliciet gemaakte levenscyclus dwingt af dat dingen op de juiste plaats en tijd gebeuren.
Gebruikersacties kunnen door de hele applicatie worden gedeeld en, bij zware bewerkingen, gemakkelijk op de achtergrond worden uitgevoerd. Getypeerde resources *kunnen* in de user interface geïnjecteerd worden (maar dit hoeft niet) en de toestand van Swing-componenten kan over gebruikerssessies heen bewaard worden.

Er is momenteel al een prototype van het Swing Application Framework beschikbaar (zie referenties). De oplevering is gepland voor 2008, met Java SE 7. Voor grote projecten waarvan de oplevering na de release van Java SE 7 gepland is, is het zeker mogelijk om het prototype nu al in te zetten.
Tot de uiteindelijke oplevering zal dan wel rekening gehouden moeten worden met kleine aanpassingen in de API, maar dit weegt niet op tegen de voordelen die het framework biedt.
Voor ontwikkelaars betekent dit framework een eenduidige en gestructureerde manier van werken, wat vooral de beginnende Swing-ontwikkelaar erg zal aanspreken. Ervaren Swing-ontwikkelaars zullen het framework met open armen ontvangen, omdat zij daardoor niet zelf telkens weer het wiel opnieuw hoeven uit te vinden en efficiënter kunnen werken.

Referenties

JSR 296 – Swing Application Framework: http://jcp.org/en/jsr/detail?id=296
Prototype van het framework: http://appframework.dev.java.net/