ESPRITSYSTEMS

Applikations-Programmierung mit EsprIT

Einführung in die Programmierung von lokalen Anwendungen mit dem Esprit Application Suite Framework

Oktober 2013
Rainer Buesch
www.esprit-systems.de

Inhaltsverzeichnis

1 Einführung

Dieses Dokument ist eine Einführung in die Entwicklung von lokalen (nicht netzwerkfähigen) Java-Applikationen basierend auf der *Esprit Application Suite*[1] Software.

Ein Schulungsteilnehmer hat mir einmal gesagt: „*Es gibt hunderte Bücher über Java. Irgendwo finde ich immer ein Beispiel, wie etwas funktioniert, aber eigentlich nützt mir das nichts. Ich kann mir aus vielen Bruchstücken etwas zusammenstricken, aber ich weiß nicht wirklich, wie man eine Applikation programmiert. Was mir fehlt, ist der große Rahmen, in den man alles einbaut*."

Mit der Esprit-Software steht ein solcher Rahmen zur Verfügung. Sie bildet ein umfangreiches und leistungsfähiges Framework, in welchem über viele Jahre hinweg praxistaugliche Problemlösungen gesammelt und unter einem einheitlichen Konzept eingebunden wurden. Sie bildet den Rahmen und beinhaltet die Bausteine für kundenspezifische Anwendungen aller Art. Die Funktionalität der Esprit-Software ist zudem über ausgiebige automatische Tests abgesichert.

1.1 Robuste Software

Java ist eine phantastische Programmiersprache mit einer enormen Fülle an Klassenbibliotheken. Es gibt eine unüberschaubare Menge von Tools und Frameworks, die das Programmiererleben vereinfachen sollen. Aber auch mit der besten Programmiersprache kann man schlechte Programme schreiben.

Man kann eine Software so entwickeln, dass sie *funktioniert* oder so, dass sie *robust funktioniert*. Dazwischen liegen Welten an Erfahrung! Zu *robuster* Software gehört, dass sie ein klares Design besitzt, durch und durch Objekt-orientiert strukturiert und damit redundanzfrei ist, dass sie auch ohne Kommentare leicht zu lesen und zu verstehen ist und last not least, dass ihre Funktionalität über automatische Tests abgesichert ist. Auch Dinge, wie eine ausführliche Fehlerbehandlung sowie gutes Logging und eine Unterstützung fürs Debugging sind wichtige Aspekte. Auf eine solche Software kann man bauen und mit ihr wachsen. Mit lediglich „funktionierender" Software wird man langfristig untergehen, weil der Aufwand für Änderungen überproportional wächst. Jede Änderung könnte wieder neue, unerwünschte Seiteneffekte mit sich bringen. Ich habe nicht selten erlebt, dass Entwickler den größten Teil ihrer Zeit im Debugger verbringen - da hört dann der Spaß auf. Mit dem Esprit-Framework bleibt er hoffentlich erhalten.

1.2 Programmierstil

Programmierer sind kreative Menschen und als solche immer auch Individualisten. Wenn in einem Projekt fünf Programmierer zusammenarbeiten, dann haben wir fünf unterschiedliche Programmierstile. Jeder hat seine eigene Meinung, wie etwas am besten gemacht werden sollte. Das geht soweit, dass einer den Code des anderen erst umformatieren muss, um ihn lesen zu können. Ich halte es für sehr wichtig, dass alle Beteiligten erst einmal auf den gleichen Stil eingeschworen werden. Das *Esprit-Framework* leistet diesbezüglich sehr gute Dienste. Bitte halten Sie sich unbedingt an die *Java Coding Conventions*, denn Sie sind in der Regel nicht der Einzige, der ihren Code lesen und verstehen muss. Für die Esprit-Programmierung gibt darüber hinaus einen Satz von sehr nützlichen Grundregeln, um durchgehend konsistenten Code zu erreichen[2].

---

1 Die *Esprit Application Suite* bildet die Grundlage für die *Esprit Network Suite* Software, ein Framework zur Entwicklung von netzwerkfähigen Applikationen basierend auf der Esprit-Client/Server-Technologie.

2 Wird im Rahmen von Projekten zur Verfügung gestellt.

1.3 Codierungs-Sprache

Um es gleich zu Anfang ganz deutlich zu sagen: Wir schreiben unseren Code nicht nur in Java, sondern auch in **englisch**! In den Code gehört kein einziges deutsches Wort – auch nicht als Kommentar! Sonst kommen dabei solche Absurditäten heraus, wie g*etSpaltenWidth()* oder *getFahrzeugLetter()*. Ich habe schon Methodennamen gesehen, wie *löscheÜbertrag()*. Witzigerweise funktionieren Umlaute in Methoden- oder Klassennamen sogar, aber leider nicht auf allen Entwicklungsumgebungen! Stellen Sie sich vor, ein ausländischer Mitarbeiter muss solchen Code weiterentwickeln. Was soll denn dabei herauskommen? Wenn Sie kein englisch können, dann lernen sie es bei dieser Gelegenheit! Bei all dem, was wir lernen müssen, um Software zu entwickeln, sollte englisch das geringste Problem sein.

Und noch etwas möchte ich Ihnen sagen, was Sie entweder erfreuen oder erschrecken wird: verzichten Sie auf Dokumentation im Code (insbesondere den automatisch generierten)! Ein guter Code braucht keine Dokumentation, da er selbsterklärend ist! Nur, wenn etwas anders ist, als intuitiv zu erwarten, muss dokumentiert werden. Jede Dokumentation sollte eine echte Aussage beinhalten! Sie wissen was ich meine, wenn sie schon mal gewisse Hilfe-Systeme verwendet haben, die Ihnen immer nur das sagen, was Sie ohnehin schon wissen.

1.4 Framework-Philosophie

Erfahrungsgemäß werden gelegentlich Klassen in einer Art und Weise verwendet, die zwar funktioniert, jedoch der Philosophie des Frameworks zuwiderläuft. Dann erscheinen Dinge plötzlich schwierig und umständlich. Mit einer etwas anderen Denkart aber lösen sich die Probleme auf. So spielt im Esprit-Framework z.B. der *ApplicationContext* eine zentrale Rolle. In anderen Frameworks gibt es evtl. gar keine vergleichbare Klasse. Ein Programmierer, der auf die Idee käme, den *ApplicationContext* nicht verwenden zu wollen (weil er ggf. das häufige Durchreichen der *ApplicationContext*-Referenz nicht mag), würde der Philosophie entgegenlaufen und zwangsläufig früher oder später Probleme bekommen.

Um wirklich guten Java-Code zu produzieren, bedarf es einer Menge Erfahrung. Für mich als Framework-Entwickler ist es wichtig, zu erfahren, was der Anwender mit meinem Framework entwickelt und wie er es tut. Deshalb möchte ich von ihm mit ins Boot genommen werden; einerseits, um Falsch-Anwendungen zu vermeiden, andererseits, um zu erfahren, was an Bedarf wirklich besteht. Nicht selten sind Probleme, an denen ein Anwendungsentwickler sich die Zähne ausbeisst, durch eine kleine Erweiterung im Basis-Framework auf einfache Weise lösbar.

1.5 Pair-Programming

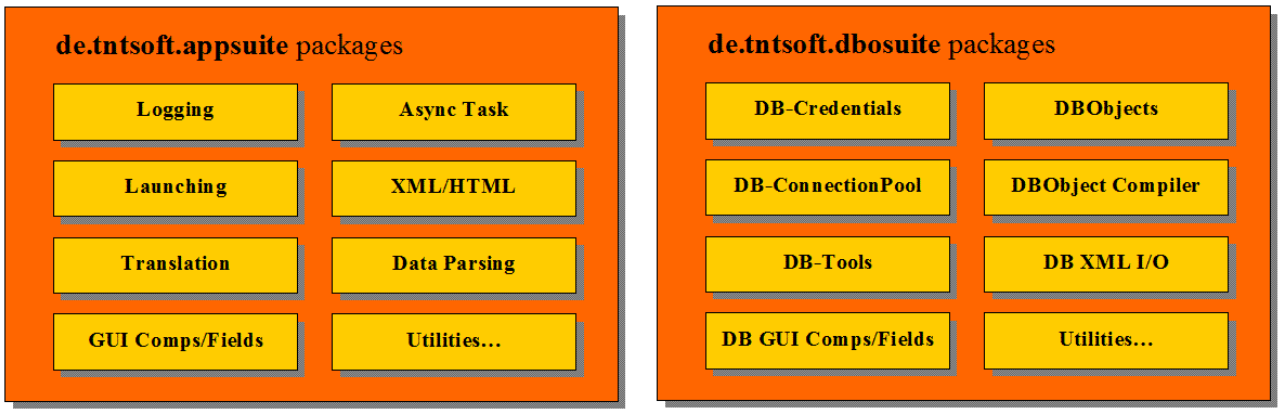
An dieser Stelle möchte ich meine Überzeugung für eine besonders effiziente Art des Programmierens zum Ausdruck bringen: das *Pair-Programming*. Was zunächst wie Ressourcenverschwendung aussehen mag (nämlich zwei Programmierer an einem Terminal – sie könnten ja auch parallel arbeiten) hat sich in der Praxis als hocheffizient erwiesen – und zwar aus folgenden Gründen:

- ➔ Lösungsansätze werden besser diskutiert und durchdacht
- ➔ Fehler und Sackgassen-Entwicklungen werden vermieden
- ➔ Bereits Vorhandenes wird besser wiederverwendet
- ➔ Es gibt mehrere Personen, die sich im Code auskennen
- ➔ Know How wird effizient ausgetauscht
- ➔ Entwickeln zu zweit macht erheblich mehr Spaß

„Tauche nie allein“ ist eine lebenswichtige Regel unter Sporttauchern. „Programmiere nie allein“ heißt die entsprechende Regel bei Programmieren – denn es ist einfach zu gefährlich.

1.6 Esprit Application Suite Übersicht

Das Esprit-Framework zur Erstellung von lokalen Applikationen wird in der Jar-Datei *EspritAppSuite.jar* ausgeliefert. Darin enthalten sind die zwei Grundmodule, aus denen es sich zusammensetzt:



- **Esprit Application Suite (de.tntsoft.appsuite)**
  Hierin ist alles enthalten, was zum Aufbau von lokalen (nicht netzwerkfähigen) Applikationen notwendig ist. Dazu gehören eine Fülle von Utility-Klassen, die allgemein nützliche Funktionalitäten bereitstellen, sowie eine ganze Reihe von API's, die Problemlösungen für häufig gestellte Herausforderungen anbieten. Die Blöcke *Translation*, *Logging* und *Launching* werden in diesem Dokument besprochen. Andere, wie *DataParsing, Asyc-Task*[3]*, AppStore* und weitere hier nicht genannte sind umfangreicher und sprengen den Rahmen einer Einführung. Das obenstehende Bild ist nur eine grobe Einteilung und bei Weitem nicht vollständig.
- **Esprit Database Suite (de.tntsoft.dbosuite)**
  Dies ist eine Datenbank-Persistenzschicht basierend auf *DBObjects*, die das Arbeiten mit Datenbanken erheblich vereinfacht. Ein *DBObject* ist eine Klasse, die den Datensatz einer Datenbank-Tabelle modelliert. Es weiß selbst, wie seine Daten aus der betreffenden Datenbank-Tabelle zu lesen bzw. zu schreiben sind. Der Anwender benötigt praktisch kein SQL mehr, sondern hat vielmehr einen Objekt-orientierten Zugriff auf die Datenbank. Die *DBObject*-Klassen werden von einem speziellen DBObject-Compiler direkt aus der Datenbank heraus generiert und sind deshalb per Definitionem stets konsistent. Der Umgang mit *DBObjects* ist in einem anderen Dokument beschrieben[4].

Auf diesen Modulen bauen alle Kundenapplikationen auf.

1.7 Entwicklungsumgebung

Als Entwicklungsumgebung verwenden wir Eclipse (www.eclipse.org), heutzutage das sicherlich meistgebrauchte Werkzeug zur Java-Entwicklung. Eclipse-Projekte werden in einem eigens erstellten Projektverzeichnis (z.B. *C:\Projects*) abgelegt, das beim Start bekannt gegeben wird, z.B:

```
eclipse.exe -data C:\Projects -vmargs -Xms80m -Xmx200m
```

3 Siehe: http://esprit-systems.de/downloads/esprit/EspritAsyncTaskFrameworkDE.pdf

4 Siehe: http://esprit-systems.de/downloads/esprit/EspritDatabaseProgrammingEN.pdf

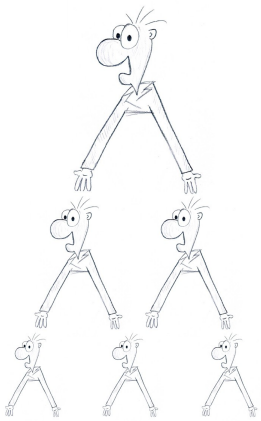
Die *-data* Option gibt an, wo das Projektverzeichnis liegt. Die hinter *-vmargs* folgenden Optionen steuern die Ausführung von Java-Programmen innerhalb von Eclipse. Mit *-Xms80m* und *-Xmx200m* wird der minimale bzw. maximale Speicherverbrauch der Java-VM voreingestellt.

Zum Arbeiten mit dem *Esprit-Framework* muss die Datei *EspritAppSuite.jar* im Build-Pfad von Eclipse eingebaut werden.

1.8 Beispiel-Projekt: *EspritWorkbench*

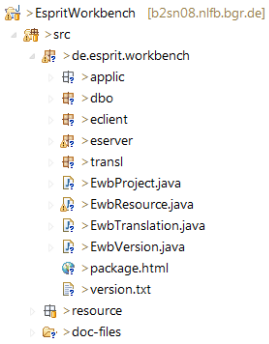
Die meisten in diesem Dokument aufgeführten Quellcode-Beispiele finden Sie (in vollständiger Form) im *EspritWorkbench* Projekt, das als Demonstrationsprojekt für Studienzwecke entwickelt wurde. Es beinhaltet vieles an Technologie, was auch „große“ Applikationen typischerweise aufweisen. Ein ganz wesentlicher Aspekt bei diesem Beispielprojekt ist, dass es von Anfang an eine professionelle Struktur besitzt, so dass es sich als Ausgangsprojekt (Template) für ein reales Kundenprojekt eignet. Im Folgenden gehe ich davon aus, dass Sie das *EspritWorkbench* Projekt in Ihrer Eclipse Umgebung lauffähig eingebunden haben.

1.8.1 Projektstruktur



Der weiter unten abgebildete Verzeichnisbaum zeigt den Grundaufbau unseres Beispiel-Projekts *EspritWorkbench.* Für die Package-Benamung der Java Quelldateien hat sich das Schema *<country>.<company>.<project>.<...>* bewährt, in unserem Fall also *de.esprit.workbench...* In dem Package *resource* sind sämtliche Ressource-Dateien abgelegt, wie z.B. Übersetzungsdateien, Icons, Audio-Dateien etc.. Wir definieren zusätzlich ein Projektnamen-Kürzel *Ewb* (**E**sprit**W**ork**b**ench), das wir als Prefix für spezialisierte Klassen verwenden wollen (z.B: *EwbTranslation*). Dies macht die Benamung vieler Klassen einfacher.

1.8.2 Die Projekt-Definitionsklassen



Das Projekt enthält auf Top-Ebene einige spezielle Klassen wie *EwbVersion, EwbProject, EwbResource* und *EwbTranslation,* die im Folgenden erläutert werden.

➔ ***EwbVersion***
Jedes Softwareprojekt entwickelt sich über einen längeren Zeitraum Schritt für Schritt weiter. Immer, wenn eine Release freigegeben wird, erhält sie eine eindeutige Versionsnummer. Die abstrakte Klasse *Version* kapselt im Esprit-Framework sämtliche versionsbezogenen Daten. Die konkrete Subklasse *EwbVersion* für unser Projekt sieht extrem einfach aus:

```
package de.esprit.workbench;
import de.tntsoft.global.util.*;

public class EwbVersion extends Version {
  // reads the local version.txt file which resides in the same package as this class
}
```

Der Aufruf `new EwbVersion()` liest lediglich die Datei *version.txt* im gleichen Verzeichnis ein, eine Leistung der Superklasse *Version*. Diese Datei hat für unser Beispiel-Projekt folgenden Inhalt:

```
#Thu Jun 23 12:15:20 CEST 2011
```

```
build.number=59
build.date=2011-06-23 12\:15\:20
required.java.version=1.6
version.major=1
version.minor=0
version.suffix=a
product.name=EspritWorkbench
software.owner=EsprIT-Systems
author.email=contact@esprit-systems.de
author.name=Rainer Buesch
author.website=http\://www.esprit-systems.de
```

Die beiden Properties *build.number* und *build.date* werden beim späteren Erzeugen der Release (mit Hilfe des *Ant* Werkzeugs) automatisch vergeben. Die Properties *version.major*, *version.minor* und *version.suffix* definieren den eindeutigen Versions-String für die Release – in unserem Falle *'1.0a'*. Die Property *required.java.version* legt fest, welche Java Runtime Version mindestens zur Verfügung stehen muss, damit die Software laufen kann. Die Information über den Autor ist selbstredend. Die hier eingetragene Information wird automatisch beim Start einer Esprit-Applikation auf der Konsole wie folgt ausgegeben.

```
EspritWorkbench Version 1.0a (Java 1.6.0_25), built 2011-06-23 #59 (Copyright www.esprit-systems.de 2013)
```

Die *EwbVersion* Klasse hat eine main-Methode, die nichts anderes tut, als eine Instanz der Klasse zu erzeugen, die dann die oben gezeigte Ausgabe macht. Das *EspritWorkbench*-Projekt wird später als Produkt „gebaut“, d.h. es wird in eine Jar Datei namens *EspritWorkbench.jar* gepackt. Es empfiehlt sich in der *Manifest.mf* Datei die *EwbVersion*-Klasse als *Main-Class* einzutragen. Dies hat den Vorteil, dass man die Jar-Datei quasi nach ihrer Version fragen kann z.B. mit dem Auftruf:

```
   java -jar EspritWorkbench.jar
```

➔ ***EwbProject***

Dies ist eine Hilfsklasse, die sich allgemein in Projekten als praktisch erwiesen hat. Sie definiert an zentraler Stelle absolute File-Pfade unseres Projekts und macht diese für Entwickler-Tools verfügbar. Wir werden sie z.B. benötigen, wenn wir ResourceBundle-Dateien für Übersetzungen erzeugen (siehe weiter unten). Auch zum Generieren von *DBObjects* zur Datenbankpersistenz wird sie benötigt.

```
package de.esprit.workbench;

import de.tntsoft.appsuite.project.*;

/**
 * This class encapsulates project specific file paths.
 * Note that this class is NOT supposed to be used in production software.
 *
 * @author Rainer Buesch
 */
public class EwbProject extends Project {
  private static final EwbProject instance = new EwbProject();

  public static final EwbProject get() {
      return instance;
  }

  private EwbProject() {} // Singleton

  @Override
  public final String getProjectName() {
      return "EspritWorkbench"; // must match the real Project name in Eclipse
  }
}
```

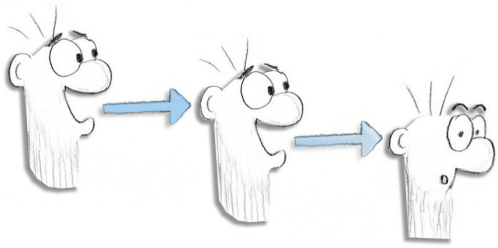
➔ Es ist zu betonen, dass *EwbProject* eine reine Hilfsklasse ist, die nur zu entwicklungs-internen Zwecken dient. Sie sollte nicht in Produktiv-Code verwendet werden.

➔ ***EwbResource***
Diese Klasse definiert, welche Ressource-Dateien im Projekt vorhanden sind. Da sie eine *main(...)* Methode besitzt, ist sie ausführbar: sie überprüft ob alle in Enum-Konstanten definierten Ressource-Dateien auch wirklich existieren.

➔ ***EwbTranslation***
Die *EwbTranslation* Klasse ist ebenfalls per *main(...)* Methode ausführbar. Sie generiert Ressource-Dateien für die Internationalisierung. Das Internationalisierungs-System wird weiter unten genauer erklärt.

2 Grundlegende Konzepte

2.1 GUI-Translations



Eine Benutzeroberfläche sollte selbstverständlich mehrsprachig sein. Java unterstützt dies mit dem *ResourceBundle* Mechanismus. In welcher Sprache sich die Benutzeroberfläche präsentiert, hängt von der Default-Einstellung der *Locale* ab, die Sie mit *Locale.getDefault()* erfragen können. Für die betreffende *Locale* wird die passende *ResourceBundle*-Datei gesucht, die im Property-Format *key=value* die Übersetzungen für die jeweilige Sprache beschreibt. Diese Dateien müssen erstellt und gewartet werden unter ständiger Kontrolle, ob die dort verwendeten Property-Namen mit denen im Java-Code verwendeten Strings übereinstimmen.

Mehrsprachigkeit bedeutet also für den Entwickler auch eine erhebliche Mehrarbeit. Das Esprit-Framework besitzt ein auf *Enum* Konstanten basierendes Übersetzungssystem, das den größten Teil der Arbeit automatisiert. *ResourceBundle*-Dateien werden im Esprit-Framework automatisch generiert und konsistent gehalten. Daher kann auf spezielle Tools, wie *ResourceBundle*-Editoren verzichtet werden.

2.1.1 NlsKey Konstanten

Den Kern des Übersetzungs-Systems bildet das *NlsKey* Interface, das die Methoden *getAppKey(), getSubKey()* und *getText()* vorschreibt. Es wird typischerweise von *Enum*-Konstanten implementiert, wie das folgende Beispiel der *Translation* Klasse *TrEwbGui* zeigt:

```
package de.esprit.workbench.transl;

/**
 * Translations for GUI labels and text
 */
public enum TrEwbGui implements NlsKey {

  TANK              ("Tank"),
  TANK_DIALOG_TITLE ("Change tank fill"),
  ...
  ;

  EwbGui(String defaultTransl) {
      Translation.put(this, defaultTransl); // stores the default translation in the Translation class
  }
  public String getAppKey() {
      return EwbTranslation.getAppKey();   // returns a progject specific prefix
  }
  public String getSubKey() {
      return SUB_KEY_LABEL;                // returns a predifined 2nd level prefix
  }
  public String getText(Object ... args) {
      return Translation.get(this, args);   // returns the translated text (according to system Locale)
```

```
    }
}
```

Das Ziel dieses Enums ist es, aus den hier festgelegten Konstanten automatisch eine *ResourceBundle*-Datei für jede unterstützte Sprache zu erzeugen. Im Konstruktor der Konstante wird eine Default-Übersetzung (in englisch) mitgegeben, die in der Singleton-Klasse *Translation* gespeichert wird. Letztere beinhaltet im Wesentlichen eine *Map*, in der alle übersetzungsbezogenen Key-Value Paare gesammelt werden. Ein Key setzt sich dabei zusammen aus dem projektspezifischen Prefix *getAppKey()* einem zweiten Prefix *getSubKey()* und der Enum-Definition selbst. Das Übersetzungs-Key-Value Paar für die Konstante *TANK* würde also folgendermaßen aussehen: *'ewb.label.TANK=Tank'*.

2.1.2 Erzeugung der ResourceBundle-Dateien

Jetzt benötigen wir die schon erwähnte Hilfsklasse *EwbTranslation*, die einfach ausgeführt werden muss, um die ResourceBundle-Dateien für die Sprachübersetzung zu erzeugen.

```
package de.esprit.workbench;

import ...;

  public EwbTranslation() {
      super(EwbProject.get()); // needs to know the project path for resource bundle generation
  }

  @Override
  protected void loadTranslationClasses() {
      TrEwbGui.values();       // translations for GUI text and labels
      TrEwbAction.values();    // translations for GUI Actions
      TrEwbField.values();     // translations for input fields
      TrEwbDbo.values();       // translations for DBObjects
      TrEwbMsg.values();       // translations for parametrized error messages
      Mood.values();           // translations for any particular constants
      ...
  }

  @Override
  protected String getTranslationAppKey() {
      return getAppKey();
  }

  public static String getAppKey() {
      return "ewb";  // defines the project-specific base key for translations
  }

  public static void main(String[] args) throws Exception {
      EwbTranslation tr = new EwbTranslation();
      tr.save(Language.DE);  // generates: resource/transl/ewb_de.properties (german)
      tr.save(Language.ES);  // generates: resource/transl/ewb_es.properties (spanish)
      tr.save(Language.FR);  // generates: resource/transl/ewb_fr.properties (french)
      ...
  }
}
```

Mit *loadTranslationClasses()* werden alle Enum-Klassen, die *NlsKeys* definieren, geladen und so die *Translation*-Klasse mit allen Defaultwerten gefüllt. Daraufhin speichern wir durch den Aufruf der *main(...)* Methode die geladenen Werte für jede gewünschte Sprache in den ResourceBundle-Dateien des Projektes. Deren Dateinamen werden sprachabhängig automatisch erzeugt. Für die Sprache *Language.DE* erhalten wir beispielsweise die Datei *resource/transl/ewb_de.properties* mit folgendem Inhalt:

```
ewb.label.TANK=## Tank
ewb.label.TANK_DIALOG_TITLE=## Change tank fill
...
```

Hier finden Sie alle definierten Keys wieder, inklusive ihrer jeweiligen Defaultwerte, die natürlich nun manuell zu übersetzen sind. Die Markierung mit *'##'* besagt, dass dies ein noch zu übersetzender Wert ist. Übersetzt sieht die Datei dann so aus:

```
ewb.label.TANK=Tank
ewb.label.TANK_DIALOG_TITLE=Ändern des Tank-Füllstandes
...
```

Um die Übersetzungen zu aktualisieren, muss die Klasse *EwbTranslation* einmal ausgeführt werden. Dadurch wird die der System-Locale entsprechende *ResourceBundle*-Datei geladen und damit die Defaultwerte mit den Übersetzungen überschrieben. In Ihrer Java Applikation kann nun sehr einfach auf einen übersetzten Wert zugegriffen werden. Der folgende Aufruf gibt den übersetzten Wert wieder:

```
String translatedTitle = TrEwbGui.TANK_DIALOG_TITLE.getText();
```

Fügen Sie noch einen neuen Übersetzungs-Key in *EwbGui* hinzu, z.B. `TANK_FULL("Tank full")` und führen die *EwbTranslation* Klasse noch einmal aus. Dann erhalten Sie folgenden Inhalt in der aktualisierten ResourceBundle-Datei:

```
ewb.label.TANK=Tank
ewb.label.TANK_DIALOG_TITLE=Ändern des Tank-Füllstandes
ewb.label.TANK_FULL=## Tank full
...
```

Wie Sie sehen, wurden die bereits übersetzten Einträge beibehalten. Der neue Eintrag ist mit *'##'* markiert, damit er als noch nicht übersetzt auffällt. Gelöschte Enum-Konstanten werden automatisch aus der *ResourceBundle*-Datei entfernt. Damit ist also die Konsistenz zwischen den Enum-Konstanten und dem Inhalt der *ResourceBundle*-Dateien gewährleistet – ein unschätzbarer Vorteil!

Die Verwendung von *NlsKeys* wird von vielen GUI-Komponenten des Esprit-Frameworks unterstützt. Wenn Sie z.B. ein *ZLabel* (Ableitung von *JLabel*) verwenden, dann können sie einen Text wie folgt zuweisen:

```
import static de.esprit.workbench.transl.TrEwbGui.*;
...
ZLabel label = new ZLabel(TANK_FULL);  // creates a label with translated text
...
label.setText(TANK_EMPTY);             // sets a translated text
```

Das *ZLabel* holt sich zur Laufzeit selbst den übersetzten Text aus der *NlsKey* Konstanten. Von den meisten Swing-Komponenten existieren Ableitungen, deren Methoden und Konstruktoren äquivalent funktionieren.

2.1.3 Parametrisierte NlsKeys

NlsKeys können auch parametrisiert sein. Die folgende *TrEwbGui* Konstante wird zur Laufzeit mit Parametern versehen:

```
public enum TrEwbGui implements NlsKey {
     FILLED_0_TANKS_WITH_1 ("{0} tanks were filled with {1}"),  // format string with place holders
     ...
```

Intern wird die *MessageFormat* Klasse verwendet, um die Parameter-Werte in die Platzhalter einzufüllen. Vielleicht ist es Ihnen schon aufgefallen, dass man der *NlsKey.getText(...)* Methode mehrere Parameter übergeben kann. Den übersetzten Text erhält man so:

```
int amount = 10;
String liquid = "Wasser";
String translatedText = TrEwbGui.FILLED_0_TANKS_WITH_1.getText(amount, liquid);
```

Die Anzahl der Parameter muss natürlich mit der Anzahl der Platzhalter im Format-String übereinstimmen. Es werden praktischerweise alle Datentypen akzeptiert.

2.1.4 NlsActionKey Konstanten

Die Enum-Klasse *TrEwbAction* implementiert das Interface *NlsActionKey* und erweitert damit *NlsKey* um drei weitere Methoden: *getButtonText(), getMenuText()* und *getTipText()*. Schauen wir uns ein Beispiel einer solchen Konstante an:

```
public enum TrEwbAction implements NlsActionKey {
    CLEAR_TANK ("Clear", "Clear tank", "Removes all fluid from the tank"),
    ...
```

Hier werden drei Übersetzungs-Keys auf einmal definiert entsprechend den drei oben genannten Methoden. Diese Konstanten werden von *GUI-Actions* (siehe weiter unten XXX) verwendet. Wird die GUI-Action einem *ZMenuItem* zugewiesen erscheint der Menü-Text, in einem *ZButton* entsprechend der Button-Text. Beide zeigen beim Überstreichen mit der Maus den Tooltip-Text an. In der Übersetzungs-ResourceBundle-Datei finden Sie pro Konstante dann drei Einträge:

```
ewb.action.CLEAR_TANK.but=## Leeren
ewb.action.CLEAR_TANK.mnu=## Tank leeren
ewb.action.CLEAR_TANK.tip=## Removes all fluid from the tank
...
```

Die '##' Markierung besagt, dass die Einträge noch übersetzt werden müssen. Wird dies vergessen, dann erscheint der betreffende Button in unserem Falle mit dem Text '## Leeren', was deutlich in der Benutzeroberfläche auffällt!

2.1.5 NlsDatabaseKey Konstanten

Die Enum-Klasse *TrEwbField* implementiert das Interface *NlsDatabaseKey* und erweitert damit *NlsKey* um weitere Methoden: *getTableName(), getColumnName()* und *getFullName()*. Auch aus diesem Enum schauen wir uns ein Beispiel an:

```
public enum TrEwbField implements NlsDatabaseKey {
    LOGIN$USERNAME ("Username"),
    LOGIN$PASSWORD ("Password"),
    ...
```

Auf diese Weise werden Übersetzungs-Keys mit der Struktur <*tabelle*>.<*spalte*> definiert. Sie werden hauptsächlich für Übersetzungen von Datenbankfeldern, bzw. zur Beschriftung von GUI-Eingabekomponenten verwendet (siehe FieldPanel XXX). In der Übersetzungs-ResourceBundle-Datei finden sich die Konstanten wie folgt wieder:

```
ewb.field.LOGIN.USERNAME=Benutzername
ewb.field.LOGIN.PASSWORD=Passwort
...
```

Wie Sie sehen wurde das '$' Zeichen aus dem Konstanten-Namen in '.' umgewandelt[5].

2.1.6 Nicht-Übersetzte Keys

Von allen *NlsKey*-Typen gibt es sogenannte Raw-Keys, wie *RawNlsKey, RawActionKey* und *RawDatabaseKey*, die **nicht** über *ResourceBundle*-Dateien übersetzt werden. Sie können jeweils mit beliebigem Text (auch ohne) erzeugt werden, wie im folgenden Beispiel, wo der Titel eines Dialogs gesetzt wird:

```
dialog.setTitle(new RawNlsKey("Preliminary"));
```

Die Verwendung von Raw-Keys ist recht praktisch bei temporären Textzuweisungen, wo man nicht ständig gezwungen sein möchte, neue Übersetzungs-Keys zu definieren. Man kann dann sehr leicht

5 Die Sonderzeichen '_' und '$' sind die einzigen, die in Java-Namen verwendet werden dürfen.

zu einem späteren Zeitpunkt nach allen vergebenen Raw-Keys suchen und sie in einem Zug in echte Übersetzungs-Keys umwandeln.

2.1.7 Projektübergreifende Übersetzung

Ein wesentlicher Vorteil des EsprIT Übersetzungssystems ist die Tatsache, dass *ResourceBundle*-Dateien von verschiedenen Projekten aufeinander aufbauen können. Die Bundles aller Projekte werden in die gleiche Singleton-Instanz der *Translation*-Klasse geladen. Dabei spielt der für ein Projekt eindeutige *Application-Key* eine wichtige Rolle: er dient als Prefix für alle Translation-Keys des betreffenden Projekts und vermeidet dadurch Überschneidungen mit den Keys anderer Projekte. Es kann auf alle Übersetzungskonstanten aller eingebundenen Projekte direkt zugegriffen werden, wie z.B.:

```
String translated = TrEasGui.RESULT.getText(); // TrEasGui is part of the underlying Esprit Application Suite
```

2.2 Resources

Resources sind zusätzliche Dateien, die zum Quellcode dazugehören, wie Textdateien, Icons, Audio-Clips etc. Im Esprit-Framework sind solche Resources per Enum-Konstanten definiert. Dadurch wird ermöglicht, dass ihre Existenz in einem automatischen Test überprüft und sichergestellt werden kann.

2.2.1 Icon Resources

Benutzeroberflächen werden gerne mit Icons (kleine graphische Symbole) dekoriert. Ein Icon auf einem Button macht dessen Funktion schneller für den Benutzer ersichtlich. Icons sind typischerweise kleine GIF-Dateien, die als Resource im Projekt eingebunden sind. Im Esprit-Framework sind Icons über *IconKey*-Konstanten definiert:

```
package resource.image;

import ...;

/**
 * This enum defines the available icons
 */
public enum EwbIconDef implements ResourceDef, IconKey {

  ACTION$TANK,
  TOOL$TANK,
  ...
  ;

  private final ResourceDefAdapter adapter;

  EwbIconDef() {
      adapter = new ResourceDefAdapter(this);
  }

  public ResourceDefAdapter get() {
      return adapter;
  }
  public Icon getIcon() {
      return Icons.getIcon(this);
  }
  public Image getImage() {
      return Icons.getImage(this);
  }
}
```

Diese Enum-Klasse definiert die beiden Icons *ACTION$TANK* und *TOOL$TANK*. Die zugehörigen GIF-Dateien sind im gleichen Basis-Directory wie die Klasse selbst (*resource/image*) abgelegt. Der Name der Icon-Konstanten definiert einen Sub-Pfad. Für *ACTION$TANK* wird in Wirklichkeit die Datei *resource/image/**action/tank**.gif* als System-Resource gesucht. Innerhalb der Applikation kann

dann wie folgt auf das Icon zugegriffen werden:

```
import static resource.image.EwbIconDef.*;
  ...
  Icon icon = TOOL$TANK.getIcon(); // accessing an icon
  ...
  ZLabel label = new ZLabel(TOOL$TANK); // or using a ZLabel with an icon
```

Der *getIcon()* Aufruf delegiert die Anfrage an die *Icons*-Klasse. Diese fungiert als Cache für die bereits geladenen Icons. Nur beim allerersten Zugriff wird ein Icon wirklich geladen.

2.2.2 Prüfung von Resourcen

Ob alle definieren Icon-Konstanten auch wirklich als Datei existieren, kann durch einen „Resource-Check“ verifiziert werden. Die Klasse *EwbIconDef* ist im Konstruktor der Top-Level Klasse *EwbResource* registriert. Wird letztere ausgeführt (sie hat eine main-Methode), dann prüft sie die Existenz aller definierten Icons. Damit ist die Konsistenz zwischen den definierten Icon-Konstanten und den wirklichen existierenden Icon-Dateien gewährleistet.

2.3 GUI-Actions

Eine *Action* ist eine gekapselte Funktion, die wir sehr einfach in ein GUI einbauen können, indem wir sie einer GUI-Komponenten wie z.B. einem *JButton* oder einem *JMenuItem* oder auch beiden zuweisen. GUI-Komponenten horchen auf den Zustand einer Action. Wird eine Action mit *setEnabled(false)* deaktiviert, dann werden alle betroffenen GUI Komponenten ausgrauen. So ist also an zentraler Stelle – nämlich in der Action - definiert, ob eine bestimmte Funktion gerade aktiv ist oder nicht. Viele GUI-Klassen besitzen bereits eingebaute innere Actions und bieten diese mit entsprechenden *getActionXX()* Methoden zur Verwendung an.

2.3.1 Verwendung von Actions

In unserem *ButtonPanel* haben wir bereits zwei Actions verwendet - die *ClearAction* und die *ExitAction*. Folgendermaßen könnten wir diese Actions auch in einem Menü zur Verfügung stellen. Wir definieren im *EwbMainFrame* einen *JMenuBar* als innere Klasse:

```
  private class MyMenuBar extends JMenuBar {
      MyMenuBar(MyLogConsole logConsole) {
          JMenu fileMenu = new ZMenu(TntGui.FILE);
          fileMenu.add(logConsole.getActionClear());
          fileMenu.addSeparator();
          fileMenu.add(getActionExit());
          add(fileMenu);
      }
  }
```

Wir platzieren dieses Menü in der Top-Region des *EwbMainFrame* durch folgenden Aufruf im Konstruktor:

```
setJMenuBar(new MyMenuBar(logConsole));
```

Und schon können wir auch per Menü die Console löschen oder die Applikation verlassen.

2.3.2 Actions selbstgemacht

Um den Umgang mit Actions zu lernen, wollen wir eine eigene *TankClearAction* erstellen, mit der

wir z.B. den Tankinhalt leeren können. Diese *Action* soll nur dann scharf sein, wenn der Tank auch wirklich Flüssigkeit enthält, ansonsten soll sie ausgegraut erscheinen – sie ist dann im *disabled* Zustand. Hier ist das funktionierende Beispiel:

```
public class TankClearAction extends ZAction<EwbApplicationContext> {

  private final TankModel tankModel;

  public TankClearAction(EwbApplicationContext ctx, TankModel tankModel) {
      super(ctx, EwbAction.);
      this.tankModel = tankModel;
      tankModel.addTankListener(new MyTankListener());
      performAvailabilityCheck(); // initialize to current value
  }

  @Override
  protected boolean checkEnableCondition() {
      return tankModel.getLevel() > 0;
  }

  @Override
  protected void actionFired(ActionEvent e) throws Exception {
      tankModel.setLevel(0);
  }

  private class MyTankListener implements TankEvent.Listener {
      public void tankChanged(TankEvent e) {
          performAvailabilityCheck(); // react on changes
      }
  }
}
```

Die *TankClearAction* ist abgeleitet von der abstrakten Superklasse *ZAction*, die eine ganze Menge an Grundfunktionalität bereitstellt. In der Methode *actionFired(ActionEvent)* muss implementiert werden, was die Action tun soll – in unserem Falle das Leeren des Tankinhalts. Damit die Action nur bei vorhandenem Füllstand „scharf“ ist, muss sie sich als Interessent beim *TankModel* registrieren. Sobald sich dort etwas ändert wird sie informiert und löst durch Aufruf der *performAvailabilityCheck()* Methode die Prüfung aus, ob sie „scharf“ sein soll. Die Bedingung dazu ist in der Methode *checkEnableCondition()* formuliert, die nur dann *true* zurückgibt, wenn der Füllstand größer als 0 ist.

Beachten Sie, dass *performAvailabilityCheck()* bereits im Konstruktor einmal aufgerufen wurde, damit sich die *Action* gleich von Anfang an richtig auf ihr *TankModel* einstellt. Diese Initialisierung darf keinesfalls vergessen werden! Lassen sie uns die neue Action in den *TankDialog* einbauen:

```
getButtonPanel().add(new TankClearAction(ctx, tankModel), 0);
```

2.3.3 Innere Actions

Es ist sehr praktisch, wenn eine Klasse alles, was sie kann, bereits selbst als *Action* zur Verfügung stellt. Dann kann man alle Funktionen dieser Klasse sehr einfach ins GUI einbauen. Wir wollen dieses Muster einmal beispielhaft mit unserer *TankClearAction* im *TankModel* anwenden. Als erstes erweitern wir das *TankModel* Interface um die Methode:

```
TankClearAction getActionClear();
```

Nun sind wir natürlich gezwungen, die neue Methode im *DefaultTankModel* zu implementieren. Dies tun wir wie folgt:

```
public class DefaultTankModel implements TankModel {
  private final TankClearAction actionClear;
  private final List<TankEvent.Listener> listenerList = new ArrayList();
  private final int capacity;
  private int level;
```

```
public DefaultTankModel(int capacity, int level) {
    this.capacity = capacity;
    setLevel(level);
    actionClear = new TankClearAction(this);
}

@Override
protected TankClearAction getActionClear() {
    return actionClear;
}
...
```

Der Einbau der *TankClearAction* im *TankDialog* vereinfacht sich damit wie folgt:

```
getButtonPanel().add(tankModel.getClearAction(), 0);
```

2.4 Logging

Ein Logging Framework ist Grundbestandteil einer Applikation. Da Java diesbezüglich lange Zeit nichts zu bieten hatte, sind diverse Lösungen entstanden, die unterschiedliche Verbreitung erlangt haben. Mittlerweile gibt es auch im Standard-Java eine Logging API, die natürlich verwendet werden kann. Nichtsdestotrotz enthält die Esprit-Software eine eigenes Logging-Framework, welches einige Vorzüge hat, die der Standard leider nicht bietet. Insbesondere ist es einfacher zu benutzen und mehr an dem orientiert, was in der Praxis wirklich gebraucht wird.

2.4.1 LogChannel

Zur Ausgabe von Logmeldungen bedarf es einer *LogChannel* Instanz. Das Beispiel *SimpleLogging1* zeigt, wie eine solche Instanz erzeugt und benutzt werden kann. *LogChannel* implementiert das Interface *LogSupport*, welches die Methoden zur Ausgabe verschiedener *LogLevels* logError(...), logWarning(...), logInfo(...) etc. vorschreibt. Im oben genannten Beispiel wird eine Logmeldung von jedem *LogLevel* ausgegeben.

➔ Im standard-Java Logging gibt es pro Klasse eine statische Logger-Instanz. Dieses Konzept führt dazu, dass in jeder loggenden Klasse statischer Code stehen muss, um den betreffenden Logger zu erzeugen und u.U. sehr viele gecachte Logger-Instanzen existieren. Im Unterschied dazu gibt es in einer Esprit-Applikation typischerweise nur eine einzige *LogChannel* Instanz im zentralen *ApplicationContext*.

2.4.2 LogLevel

Die Esprit-Logging API definiert 7 verschiedene log-Level, die die Wichtigkeit der Meldung bestimmen (siehe *LogLevel* Enum-Konstante).

- ➔ **Fatal**
  Meldet fatale Fehler. Nach einem solchen Fehler kann das System nicht mehr korrekt weiterarbeiten und sollte terminiert werden.
- ➔ **Error**
  Meldet einen Fehler, ggf. mit Ausgabe des Exception-Stacktraces.
- ➔ **Warning**
  Meldet eine Warnung, die den Benutzer darauf hinweist, dass etwas nicht in Ordnung sein könnte.

➔ **Info**
Meldet eine Systemaktivität zur reinen Information.

➔ **Verbose**
Dies ist der Geschwätzigkeits-Mode. Es wird detaillierte Information ausgegeben.

➔ **Debug**
Gibt Debug-Meldungen aus, die nur für den Entwickler eine Bedeutung haben.

➔ **Dump**
Dies ist ein spezieller Debug-Mode. Eine Dump-Meldung wird immer ausgegeben, egal auf welches Level das Logging eingestellt ist. Ist das Dump-Level selbst als aktuelles Level eingestellt, so werden ausschließlich Dump-Meldungen ausgegeben, alle anderen werden unterdrückt. Dies hat sich beim Debuggen sehr bewährt, da man ausschließlich die Dump-Ausgaben sieht, die man hineinprogrammiert hat und nicht durch andere Meldungen verwirrt wird. Dump-Ausgaben sollten nur temporär sein, d.h. sie sollten in produktiv-Code nicht mehr vorkommen.

Das im *LogChannel* mit *setLogLevel(...)* eingestellte *LogLevel* wirkt als Filter für Logmeldungen. Wird z.B. das *LogLevel* WARNING eingestellt, dann werden alle darüberliegenden Meldungen (WARNING, ERROR, FATAL) ausgegeben, aber alle darunterliegenden (INFO, VERBOSE, DEBUG) unterdrückt. Eine DUMP Meldung wird unabhängig davon *immer* ausgegeben.

2.4.3 LogPrinter

Standardmäßig werden Logmeldungen auf der Systemkonsole ausgegeben. Oft ist aber auch eine evtl. zusätzliche Ausgabe in einer Datei oder einer Text-Konsole der Applikation erwünscht. Einem *LogChannel* können deshalb verschiedene *LogPrinter*-Implementierungen hinzugefügt werden, die jeweils die Ausgabe auf ein ein anderes Ziel leiten. Jeder *LogPrinter* kann individuell auf ein unterschiedliches LogLevel oder Logging-Format eingestellt werden. Hat er keine individuelle Einstellung, dann benutzt er als Default die Einstellung des LogChannels, der ihn ansteuert.

Drei verschiedene *LogPrinter* sind im *LogChannel* bereits eingebaut:

➔ **LogPrinterConsole**
Macht Ausgaben auf die Systemkonsole. Dieser *LogPrinter* ist standardmäßig aktiv.
Siehe *LogChannel* Methoden: *openLogConsole(), closeLogConsole()*

➔ **LogPrinterFile**
Dieser *LogPrinter* schreibt Logmeldungen in eine Logdatei in einem vorgegebenen Logverzeichnis. Der Dateiname wird dabei automatisch generiert. Wird eine vorgegebene maximale Größe erreicht, so wird automatisch eine neue Logdatei angelegt.
Siehe *LogChannel* Methoden: *openLogFile(File logDir), closeLogFile()*

➔ **LogPrinterFrame**
Dieser *LogPrinter* öffnet einen eigenen Frame mit einer Textkonsole.
Siehe *LogChannel* Methoden: *openLogFrame(String title), closeLogFrame()*

Natürlich können beliebige eigene Implementierungen von *LogPrinter* als Ableitungen von *AbstractLogPrinter* erstellt und im *LogChannel* eingebunden werden.

2.4.4 LogMessageFormatter

Jeder *LogPrinter* besitzt einen *LogMessageFormatter*, der für die Formatierung der Logmeldungen verantwortlich ist. Dort kann mit entsprechenden setter-Methoden bestimmt werden, welche Informationen in einer Logmeldung mit ausgegeben werden, wie z.B. Stacktrace, Zeitstempel, Klassenname, Thread etc.

➔ Die Logmeldungen einer Applikation können sich von Version zu Version stark verändern. Deshalb ist es keine gute Idee, Logausgaben mit speziell geschriebenen Programmen auszuwerten.

2.4.5 SimpleLogSupport

Längst nicht alle Klassen haben etwas zu loggen. Diejenigen aber, die etwas zu „sagen“ haben, sollten dies auf eine möglichst bequeme Art tun können. Alle Klassen, die *SimpleLogSupport* implementieren (dies sind viele), haben es besonders einfach, denn sie besitzen selbst die dort vorgeschriebenen Methoden *logInfo(String), logWarning(String)* etc. Es gibt eine Hilfsklasse namens *SimpleLogSupportAdapter*, die das Implementieren von *SimpleLogSupport* stark vereinfacht.

2.4.6 Loggen über den ApplicationContext

Viele Klassen einer Applikation besitzen typischerweise eine Referenz auf den *ApplicationContext* (die am meisten „herumgereichte“ Referenz). Dieser besitzt einen eingebauten Standard-LogChannel, implementiert *LogSupport* und kann deshalb von jeder beliebigen Klasse zum Loggen verwendet werden. Dabei muss die loggende Klasse lediglich ihre *this*-Referenz mitgeben:

```
ctx.logWarning(this, ”Something may be wrong, but please keep smiling”);
```

2.5 Utilities

2.5.1 Debugging-Hilfen

Fehlersuche in Software ist zeitraubend und anstrengend. Sie ist immer noch eine der unangenehmsten Herausforderungen, mit denen wir Programmierer uns herumplagen müssen. Und das Schlimme ist, dass wir stets selbst an daran schuld sind. In diesem Zusammenhang möchte ich ausdrücklich die Test-getriebene Entwicklung empfehlen, bei der zunächst ein Testprogramm erstellt wird, das die zu erstellende Klasse in allen Funktionalitäten – auch Fehlerfälle – durch testet. Diese Art der Entwicklung ist zwar aufwändig, aber absolut lohnend, zumal die Funktionalität durch die Tests auch **langfristig abgesichert** ist. Die Test-getriebene Entwicklung zu beschreiben, sprengt den Rahmen dieses Dokumentes. Hier liegt vielmehr der Schwerpunkt auf den Hilfen zur Fehlervermeidung und Fehlersuche, also den Debugging-Tools, die das Esprit-Framework anbietet.

2.5.2 Das *Dumpable* Interface

Das *Dumpable* Interface schreibt die folgende Methoden vor, die für die Ausgabe von Debug-Informationen gedacht sind. Dies ist sehr praktisch als Kontrolle und fürs Debuggen!

➔ *toString(), toDumpString()*
diese Methoden sind in der abstrakten Klasse *DumpableObject* einmalig implementiert

➔ *toString(ToString)*
wird überschrieben und gibt den Zustand einer Klasse als einen einzeiligen Text aus. Hier werden typischerweise nicht alle, sondern nur die wichtigsten Klassenvariablen ausgegeben.

➔ *toDumpString(DumpString)*
wird überschrieben und gibt den Zustand der Klasse als einen mehrzeiligen Text aus. Hier wird typischerweise der Zustand aller Klassenvariablen ausgegeben.

Nehmen wir als Beispiel die Klasse *PongPaddle*, eine einfache Java-Bean Klasse, die ein paar Instanzvariablen kapselt. Sie ist von *DumpableObject* abgeleitet, in der *toString()* und *toDumpString()* bereits implementiert sind und zwar so, dass sie die anderen beiden Methoden aufrufen. *PongPaddle* überschreibt nun sowohl *toString(ToString)* als auch *toDumpString(DumpString)*, um die Ausgabe mit seinen konkreten Werten zu füllen.

```
public class PongPaddle extends DumpableObject {
  private final int playerId;
  private final Color color;
  private int currentPos;
  ...

  @Override  // creates a single line text, most important info
  public ToString toString(ToString s) {
      super.toString(s);
      s.add("playerId", playerId);
      s.add("currentPos", currentPos);
      return s;
  }

  @Override  // creates a multi line text, more detailed info
  public DumpString toDumpString(DumpString s) {
      super.toDumpString(s);
      s.add("playerId", playerId);
      s.add("currentPos", currentPos);
      s.add("color", color);
      return s;
  }
}
```

Lassen Sie uns ein *PongPaddle* bauen, und uns die Debug-Ausgabe auf der Konsole anschauen:

```
  PongPaddle paddle = new PongPaddle(1, Color.BLUE);
  paddle.setPosition(100);

  System.out.println(paddle.toString());       // sinlge line dump output
  System.out.println(paddle.toDumpString());   // multi line dump output (more detailed)
```

Die Ausgabe erscheint wie folgt formattiert:

```
PongPaddle[playerId=1, currentPos=100]

 Content of: PongPaddle
   playerId: 1
 currentPos: 100
      color: java.awt.Color[r=0,g=0,b=255]
```

Eine von *PongPaddle* abgeleitete Klasse könnte die Dump-Methoden ihrerseits überschreiben und in das übergebene *ToString*- bzw. *DumpString*-Objekt weitere Werte einfüllen, so dass die Ausgabe vollständig über die gesamte Vererbungshierarchie ist.

➔ Diese Technik hat sich in der Praxis sehr bewährt. Sie ist beim Debuggen äußerst hilfreich, um den Zustand eines Objekts auf der Systemkonsole sichtbar zu machen.

2.5.3 Weitere Utilities

Nicht selten entdeckt man in der Softwareentwicklung, dass man eine bestimmte Lösung bereits irgendwo anders im Code implementiert hat und sie nun wiederverwenden könnte, da sie eine

gewisse Allgemeingültigkeit hat. Um Code-Redundanz zu vermeiden empfiehlt es sich, solche Lösungen in Form von statischen Methoden in Utility-Klassen allgemein verfügbar zu machen. Im Laufe der Zeit sind im Esprit-Framework eine ganze Reihe solcher Utility-Klassen entstanden. Die Funktionalität dieser Hilfsklassen sollte möglichst konsistent benutzt werden. Die wichtigsten Utility-Klassen sind:

- **Util**
  Enthält besonders häufig benutzte Prüfungen, wie z.B. die mehrfach überladene Methode *isValid(...)*, die auf auf konsistente Weise die Gültigkeit eines Strings überprüft und im Fehlerfalle *false* zurückgibt. Die korrespondierende Methode *checkValid(...)* hingegen wirft im Fehlerfalle eine *RuntimeException*.
- **OsUtil**
  Enthält Betriebssystem-bezogene Hilfsmethoden z.B. zur Erfragung allgemeiner Betriebssystem-Eigenschaften, wie Name, Version und Architektur.
- **FileUtil**
  Enthält nützliche Methoden zum Umgang mit Dateien und Directories wie Anlegen, Lesen, Kopieren etc.
- **DebugUtil**
  Enthält Hilfsroutinen zum Debuggen, wie z.B. Dumps von Arrays, Collections, Maps etc.
- **GuiUtil**
  Enthält nützliche Hilfen für den Umgang mit GUI-Komponenten.
- **TaskUtil**
  Enthält nützliche Hilfen im Zusammenhang mit asynchronen Tasks.
- **TextUtil**
  Enthält häufig verwendete Funktionen zur Text-Auswertung oder Konvertierung.
- **TimeUtil**
  Enthält Hilfsroutinen zum Umgang mit Zeit.
- **EnumUtil**
  Enthält Hilfsroutinen zum Umgang mit Enumeration Konstanten.
- **ClassUtil**
  Enthält Hilfsroutinen zum Umgang mit Klassen.
- **Convert**
  Enthält Funktionen zu diversen Datenkonvertierungen
- **NullSafe**
  Enthält eine Reihe von überladenen *close(...)* Methoden z.B. zum Schließen diverser Objekt-Typen. Alle Methoden sind null-safe, wie der Klassenname impliziert.

3 Lokale Applikationen

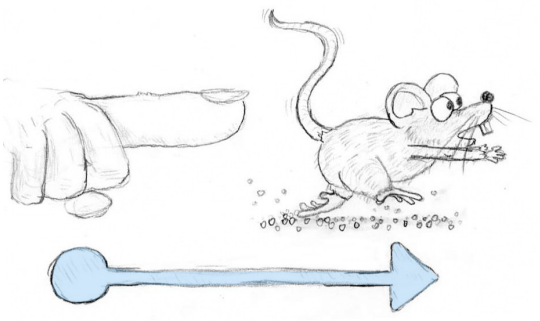
Im Folgenden wollen wir eine kleine Applikation entwickeln, um die wichtigsten Klassen und technischen Prinzipien des Esprit-Frameworks kennenzulernen. Die Applikation soll eine Benutzeroberfläche haben, mit der wir den Füllstand eines Tanks verändern können.

Dazu werden wir eine *TankModel*-Klasse erstellen, die den Füllstand speichert (Model) und eine *TankView*-Klasse, die ihn visualisiert (View). Schließlich brauchen wie noch den *TankController,* eine GUI-Komponente, die es erlaubt, den Füllstand zu verändern (Controller). Alle drei Klassen werden zum *TankTool* nach dem bewährten *Model-View-Controller* Prinzip zusammengebaut. Das

TankTool seinerseits soll als ein eigenständiger *TankDialog* innerhalb unserer weiter ausbaufähigen Applikation eingebaut werden.

Wir werden im Folgenden viele neue Klassen erstellen, die alle für unsere Applikation spezifische Funktionalität besitzen, sprich, die Subklassen von bereits vorhandenen allgemeineren Klassen sind. Es ist bewährte Praxis, sich für seine Applikation ein Prefix zu überlegen, um die Namensvergabe für Subklassen zu vereinfachen. Unser Prefix soll *Ewb* sein (für EspritWorkbench). Die Hauptklasse, die wir starten wollen, soll also *EwbMainFrame* heissen (eine Ableitung von *ApplicationMainFrame*). Sie wird das gesamte GUI beinhalten.

3.1 Starten der Applikation (Launching)



In modernen Benutzeroberflächen wird eine Applikation per Mausklick auf ein Desktop-Icon oder im Startmenü gestartet. Zusätzlich kann man den Start per Kommandozeile veranlassen, wobei typischerweise Startoptionen mitgegeben werden können. Das Starten (engl. Launchen) einer Appliaktion (evtl. auf verschiedenen Plattformen) kann durchaus eine Herausforderung sein. Das Esprit-Framework stellt dazu eine Reihe von Hilfsklassen zur Verfügung, die im Folgenden vorgestellt werden.

Unsere Applikation wollen wir mit folgenden Kommandozeilen-Optionen starten können:

```
-level 50 -capacity 100
```

wobei *-level* den initialen Füllstand und *-capacity* die maximale Kapazität des Tanks angeben.

Dazu erstellen wir zunächst ein neues Package *de.tntsoft.ewb.launch* in dem wir die alle Klassen anlegen, die zum Starten der Applikation erforderlich sind.

3.1.1 Command line Parsing

Als Erstes brauchen wir einen *EwbCommandLineParser*:

```
package de.tntsoft.ewb.launch;

import de.tntsoft.appsuite.launch.*;
import de.tntsoft.global.clp.*;

/**
 * This command line parser defines the two command line options -location and -dimension
 *
 * @author Rainer Buesch
 */
public class EwbCommandLineParser extends CommandLineParser {
  private final MyStartLevelOption optStartLevel;
  private final MyCapacityOption optCapacity;

  public EwbCommandLineParser(Class mainClass, String[] args) {
      super(mainClass, args);
      optStartLevel = addOption(new MyStartLevelOption());
      optCapacity = addOption(new MyCapacityOption());
  }

  @Override
  protected void checkDependencies() throws Exception {
      if (getLevel() > getCapacity()) {
          throw new Exception("Start-level <"+getStartLevel()+"> exceeds max capacity <"+getCapacity()+">");
      }
  }

  private static class MyStartLevelOption extends AbstractIntegerOption {
      MyStartLevelOption() {
```

```
            super("-level", "level", "Defines the initial fluid level in the tank");
        }
    }
    private static class MyCapacityOption extends AbstractIntegerOption {
        MyCapacityOption() {
            super("-capacity", "capacity", "Defines the capacity of the tank");
            setMandatory(true);
        }
    }
}
```

Dieser Parser enthält die zwei Optionen *MyStartLevelOption* und *MyCapacityOption*, die wir als statische innere Klassen definiert haben. Da die Optionen nicht unabhängig voneinander sind, wurde die *checkDependencies()* Methode so überschrieben, dass sie überprüft, ob der angegebene Füllstand auch innerhalb des angegebenen Fassungsvermögens liegt. Bauen wir uns eine kleine Main-Klasse zum Testen:

```
package de.tntsoft.ewb.launch;

/**
 * This class launches the application
 *
 * @author Rainer Buesch
 */
public class EwbLaunch {

    public static void main(String[] args) {
        EwbCommandLineParser parser = new EwbCommandLineParser(EwbLaunch.class, args);
        parser.parse();
    }
}
```

Da diese Klasse eine *main()* Methode besitzt, können wir sie starten. Hier ist das Ergebnis:

```
java.lang.Exception: Missing mandatory option: -capacity <capacity>
        at de.tntsoft.global.clp.CommandLineParser.checkForMissingOptions(CommandLineParser.java:364)
        at de.tntsoft.global.clp.CommandLineParser.parseCommandLine(CommandLineParser.java:312)
        at de.tntsoft.global.clp.CommandLineParser.parse(CommandLineParser.java:221)
        at ApplicationCommandLineParser.getConfig(ApplicationCommandLineParser.java:35)
        at de.tntsoft.appsuite.launch.ApplicationLaunch.<init>(ApplicationLaunch.java:39)
        at de.tntsoft.ewb.launch.EwbLaunch.<init>(EwbLaunch.java:15)
        at de.tntsoft.ewb.launch.EwbLaunch.main(EwbLaunch.java:44)

java class de.tntsoft.ewb.launch.EwbLaunch -level <level> -capacity <capacity>
   -logdir <log-directory>    Defines the base directory for logging
   -level <level>             Defines the initial fluid level in the tank
   -capacity <capacity>       Defines the capacity of the tank
```

Der Start ging schief. Der Parser beklagt sich darüber, dass die Option *-capacity* nicht gefunden wurde. Klar, sie wurde ja auch mit *setMandatory(true)* für unbedingt notwendig erklärt. Im automatisch erzeugten Hilfetext taucht noch eine Option *-logdir* auf, die wir gar nicht definiert hatten. Diese ist standardmäßig im *CommandLineParser* enthalten. Ist sie nicht gewünscht, dann löschen sie Sie einfach mit *removeOption("-logdir")*. Wiederholen Sie den Startversuch nochmals mit den angegebenen Optionen: *-level 200 -capacity 100*, dann erhalten Sie folgende Exception:

```
java.lang.Exception: Given level <200> exceeds maximum capacity <100>
        at de.tntsoft.ewb.launch.EwbCommandLineParser.checkDependencies(EwbCommandLineParser.java:39)
        at de.tntsoft.global.clp.CommandLineParser.parseCommandLine(CommandLineParser.java:315)
        at de.tntsoft.global.clp.CommandLineParser.parse(CommandLineParser.java:221)
        at ApplicationCommandLineParser.getConfig(ApplicationCommandLineParser.java:35)
        at de.tntsoft.appsuite.launch.ApplicationLaunch.<init>(ApplicationLaunch.java:39)
        at de.tntsoft.ewb.launch.EwbLaunch.<init>(EwbLaunch.java:15)
        at de.tntsoft.ewb.launch.EwbLaunch.main(EwbLaunch.java:44)
```

Wie Sie sehen, wirkt jetzt die Konsistenzprüfung wie in *checkDependency()* definiert. Geben Sie den Wert für die *-level* Option richtig an und alles ist in Ordnung.

3.1.2 Configuration-Bean

Ist das Parsen der Kommandozeile erfolgreich durchlaufen, dann können alle definierten Optionen

mit *getValue()* nach ihrem Eingabewert befragt werden. Viel praktischer aber ist es, wenn uns der Parser ein fertiges *EwbApplicationConfig* Objekt gibt, das alle diese Werte schon beinhaltet. Wir definieren dieses wie folgt:

```
package de.tntsoft.ewb.launch;

import de.tntsoft.appsuite.launch.*;
import de.tntsoft.global.util.*;

/**
 * This class contains the startup values for our application
 *
 * @author Rainer Buesch
 */
public class EwbApplicationConfig extends ApplicationConfig {
  private final int startLevel;
  private final int capacity;

  public EwbApplicationConfig(int startLevel, int capacity) {
      super(clp);
      this.startLevel = startLevel;
      this.capacity = capacity;
  }

  public final int getStartLevel() {
      return startLevel;
  }

  public final int getCapacity() {
      return capacity;
  }

  @Override
  public ToString toString(ToString s) {
      super.toString(s);
      s.add("level", startLevel);
      s.add("capacity", capacity);
      return s;
  }
}
```

Diese Klasse ist eine einfache java-Bean, die sich im Konstruktor mit Werten aus unserem *EwbCommandlineParser* befüllt. Letzterer muss ein wenig erweitert werden, damit die Werte abfragbar sind:

```
public class EwbCommandLineParser extends ApplicationCommandLineParser {
  ...

  @Override
  protected EwbApplicationConfig createConfig() throws Exception {
      return new EwbApplicationConfig(getStartLevel(), getCapacity());
  }

  public int getStartLevel() {
      return optStartLevel.getValue();
  }

  public int getCapacity() {
      return optCapacity.getValue();
  }
  ...
}
```

Wir merken uns die vergebenen Optionen, damit wir getter-Methoden für deren Werte einbauen können. Eine wichtige Änderung ist, dass wir nun von der Mutterklasse *ApplicationCommandLineParser* ableiten, die das Überschreiben von *createConfig()* erzwingt. An dieser Stelle erzeugen wir unsere spezielle *EwbApplicationConfig* Instanz. Sie kann dann einfach mit *getConfig()* vom Parser erfragt werden.

Testen wir dies mit unserer leicht erweiterten *EwbLaunch* Klasse:

```
package de.tntsoft.ewb.launch;

public class EwbLaunch {
```

```
  public static void main(String[] args) throws Exception {
      EwbCommandLineParser parser = new EwbCommandLineParser(EwbLaunch.class, args);
      parser.parse();

      EwbApplicationConfig cfg = parser.getConfig();
      System.out.println(cfg.toDumpString());
  }
}
```

Als Ergebnis sehen Sie folgende Ausgabe:

```
Content of: EwbApplicationConfig
 applicationName: esprit-application
    dataDir: C:\Users\rainer
     logDir: <null>
      level: 50
   capacity: 100
```

Der Aufruf von *toDumpString()* in *EwbApplicationConfig* erzeugt diese wunderschön formatierte Ausgabe. Da unsere Bean-Klasse von *AbstractDumpableObject* ableitet und daher das *Dumpable* Interface implementiert, konnten wir *toString(ToString)* als auch *toDumpString(DumpString)* überschreiben, um die Ausgabe mit Werten zu füllen – dies ist praktisch und sollte konsequent bei allen Modell-Klassen ebenso gemacht werden!

➔ Es hat sich in der Praxis bewährt, alle Objekte, die Daten beinhalten (JavaBeans), von *AbstractDumpableObject* abzuleiten. Dies ist sehr hilfreich beim Debuggen zur Datenausgabe auf der Systemkonsole.

Der Hauptzweck des *EwbApplicationConfig* Objektes ist es, uns vom *CommandLineParser* unabhängig zu machen[6].

3.1.3 Der ApplicationContext

Nun, wo wir das fertige *EwbApplicationConfig*-Objekt haben, könnten wir theoretisch unseren *EwbMainFrame* starten und ihm das Konfigurationsobjekt übergeben. Bevor wir dies tun, wollen wir aber noch ein anderes *EwbApplicationContext* Objekt definieren, das wie folgt aussieht:

```
package de.tntsoft.ewb.launch;

import de.tntsoft.appsuite.launch.*;

public class EwbApplicationContext extends ApplicationContext {

  public EwbApplicationContext(EwbApplicationConfig appCfg) throws Exception {
      super(appCfg);
  }

  @Override
  public EwbApplicationConfig getApplicationConfig() {
      return super.getApplicationConfig();
  }
}
```

Dieses Objekt mag Ihnen zunächst unnötig erscheinen, aber es wird später in unserer Applikation eine zentrale Rolle spielen, die sich aber erst nach und nach vollständig erschließt. Es stellt so etwas wie die „Runtime-Umgebung“ der Applikation dar und kann z.B. benutzt werden, um Applikations-globale Variablen zu speichern. Eine speichert es bereits: das *EwbApplicationConfig*-Objekt. Die Methode *getApplicationConfig()* wurde so überschieben, dass sie dieses Objekt vom richtigen Typ zurückgibt (die Superklasse speichert es als *ApplicationConfig*) – so eine Methode könnte man einen „Generic Delegator“ nennen, sie dient lediglich zum automatisch richtigen Casten.

---

6 Eine Applikation muss nicht immer zwangsläufig per Kommandozeile gestartet werden. Zum Testen beispielsweise ist es praktisch, den *CommandLineParser* zu umgehen.

3.1.4 Der ApplicationMainFrame

Jetzt können wir endlich den *EwbMainFrame* erzeugen, dem wir dann den *EwbApplicationContext* übergeben. Als MainFrame verwenden wir erst mal zum Testen folgende Klasse, die wir in weiser Voraussicht in einem neuen Package namens *de.tntsoft.ewb.gui* anlegen:

```
package de.tntsoft.ewb.gui;

import javax.swing.*;
import de.tntsoft.ewb.launch.*;
import de.tntsoft.appsuite.gui.*;

/**
 * This is the MainFrame of our application
 */
public class EwbMainFrame extends ApplicationMainFrame {

  public EwbMainFrame(EwbApplicationContext ctx) {
      super(ctx);
      add(new JButton("Welcome to the EsprIT Application"));
  }
}
```

Die *main()* Methode unserer *EwbLaunch* Klasse erweitern wir entsprechend wie folgt und probieren es aus:

```
  public static void main(String[] args) throws Exception {
      EwbCommandLineParser parser = new EwbCommandLineParser(EwbLaunch.class, args);
      parser.parse();

      EwbApplicationConfig cfg = parser.getConfig();
      EwbApplicationContext ctx = new EwbApplicationContext(cfg);

      new EwbMainFrame(ctx).popup();
  }
```

Was sehen Sie? Unser *EwbMainFrame* erscheint zwar aber er ist offensichtlich nicht richtig gepackt, denn er hat sich nicht auf die Größe des enthaltenen Buttons eingestellt. Ah, wir haben den *pack()* Aufruf vergessen, werden Sie jetzt denken. Leider stimmt das nicht. Die *popup()* Methode unserer *ApplicationMainFrame* Superklasse ist so implementiert, dass sie den *pack()*-Aufruf nachholt, falls er vergessen wurde. Das funktioniert auch, wie man im Debugger nachvollziehen kann.

Wir haben einen ganz anderen, viel subtileren und wirklich schlimmen Fehler gemacht, der leider häufig im Zusammenhang mit Swing-GUIs gemacht wird: Der *EwbMainFrame* wurde im Java main-Thread instantiiert – und das darf nicht sein! Swing ist per Definitionem single-Threaded, d.h. es darf nur **ein** Thread in Swing aktiv sein, und dies ist der berühmt berüchtigte Event-Dispatcher-Thread (kurz ET). Wir **müssen** den *EwbMainFrame* vom ET aufbauen lassen und das geht so:

```
  public static void main(String[] args) throws Exception {
      EwbCommandLineParser parser = new EwbCommandLineParser(EwbLaunch.class, args);
      parser.parse();

      EwbApplicationConfig cfg = parser.getConfig();
      final EwbApplicationContext ctx = new EwbApplicationContext(cfg);

      EventQueue.invokeLater(new Runnable() {
          public void run() {
              new EwbMainFrame(ctx).popup();
          }
      });
  }
```

Durch den *invokeLater(...)* Aufruf wird die Erzeugung des *EwbMainFrames* an den ET zur Ausführung übergeben. Als Ergebnis sehen wir einen perfekt gepackten Frame.

➔ Es ist essentiell wichtig, dass jede Aktion an einer GUI-Komponente vom Event-Dispatcher-Thread (ET) ausgeführt wird. Wird diese Regel nicht beachtet, dann kann sich Ihr Programm spontan aufhängen. Solche Fehler sind kaum reproduzierbar und äußerst schwer auffindbar.

3.1.5 Application-Launching

Das Starten einer Applikation besteht, wie wir gesehen haben, aus folgenden Schritten:

- ➔ Parsen der Kommandozeile (*EwbCommandLineParser*)
- ➔ Erzeugen der Konfigurations-Bean (*EwbApplicationConfig*)
- ➔ Erzeugen des Runtime-Kontextes (*EwbApplicationContext*)
- ➔ Erzeugen des GUI-MainFrames (*EwbMainFrame*)

Da diese Prozedur immer gleich ist, wird sie in Form der *AbstractApplicationLaunch* Klasse im Esprit-Framework bereitgestellt. Damit lässt sich unser *EwbLaunch* folgendermaßen formulieren.

```
package de.tntsoft.ewb.launch;

import ...;

/**
 * This class launches the Application
 */
public class EwbLaunch extends AbstractApplicationLaunch<EwbApplicationConfig, EwbApplicationContext> {

  public EwbLaunch(String[] args) {
      super(true, args); // true means: this is a GUI application
  }

  @Override
  protected EwbCommandLineParser createCommandLineParser(String[] args) {
      return new EwbCommandLineParser(EwbLaunch.class, args);
  }

  @Override
  protected EwbApplicationContext createContext(EwbApplicationConfig cfg) throws Exception {
      return new EwbApplicationContext(cfg);
  }

  @Override
  protected ZFrame launchMainFrame(EwbApplicationContext ctx) throws Exception {
      return new EwbMainFrame(ctx);
  }

  @Override
  protected IconKey createSplashIcon() {
      return TntIcon.LOGO$ESPRIT$TECHNOLOGY;
  }

  public static void main(String[] args) {
      new EwbLaunch(args);
  }
}
```

Diese Klasse tut im Prinzip das Gleiche wie unsere erste Launcher-Version, nur dass hier die Struktur von der Superklasse *AbstractApplicationLaunch* zwingend vorgegeben ist. Aber sie leistet noch einiges mehr. Das im Konstruktor übergebene *true* bedeutet, dass eine GUI-Applikation gestartet werden soll - dadurch ist sichergestellt, dass der *EwbMainFrame* im ET augerufen wird. Durch Überschreiben von *createSplashIcon()* erreichen wir, dass während des Starts ein Splash-Bild angezeigt wird. Passiert beim Launchen ein Fehler (z.B. durch eine falsche Kommandozeilen-Option) dann wird dieser nicht nur auf der Konsole, sondern auch in einem Fehlerdialog angezeigt.

3.2 Aufbau des MainFrames

Nun können wir damit beginnen, unseren *EwbMainFrame* mit Leben zu füllen. Dazu wollen wir im oberen Bereich einen Toolbar einbauen, im mittleren Bereich soll eine Text-Konsole eingebaut werden und im unteren ein *ButtonPanel* mit zunächst zwei Buttons: einer zum Leeren der Text-

Konsole und einen Weiteren zum Beenden der Applikation. Schauen wir uns den Code dazu an.

```
package de.tntsoft.ewb.gui;

import de.tntsoft.ewb.launch.*;
import de.tntsoft.appsuite.gui.*;
import de.tntsoft.appsuite.gui.comp.*;
import de.tntsoft.appsuite.gui.mem.*;
import de.tntsoft.global.gui.*;

/**
 * This is the MainFrame of our application
 */
public class EwbMainFrame extends MainFrame {

  public EwbMainFrame(EwbApplicationContext ctx) {
      super(ctx);
      MyLogConsole logConsole = new MyLogConsole(ctx);

      add(new MyToolBar(ctx), NORTH);
      add(new JScrollPane(logConsole), CENTER);
      add(new MyButtonPanel(logConsole), SOUTH);
      logInfo("Application started successfully");
      logVerbose(ctx.getApplicationConfig().toDumpString());
  }

  private class MyToolBar extends ApplicationToolBar {
      public MyToolBar(EwbApplicationContext ctx) {
          super(ctx);
          add(new MemoryMonitorAction(ctx));
      }
  }
  private class MyLogConsole extends MessageConsole {
      public MyLogConsole(EwbApplicationContext ctx) {
          super(ctx.getLogChannel(), 10, 60);
      }
  }
  private class MyButtonPanel extends ButtonPanel {
      public MyButtonPanel(MyLogConsole logConsole) {
          add(logConsole.getActionClear());
          add(getActionExit());
      }
  }
}
```

Wie Sie sehen, haben wir drei spezielle GUI-Klassen erzeugt und sie im *BorderLayout* des *EwbMainFrames* plaziert. Es ist bewährte Praxis, alle Komponenten, die Spezialfunktionalität haben, als innere Klassen zu definieren (die alle mit dem Prefix *My* beginnen), so wie wir es mit *MyToolBar*, *MyLogConsole* und *MyButtonPanel* gemacht haben. Dies erhöht die Übersichtlichkeit im Layout und erleichtert spätere Erweiterungen. Starten wir unser Programm, dann erhalten wir folgenden Frame:

Im Toolbar haben wir eine *MemoryMonitorAction* eingebaut, die wir schon testen können. Sie öffnet den *MemoryMonitorDialog,* der den aktuellen Speicherverbrauch auf einer Zeitachse anzeigt. Nach dem gleichen Muster wollen wir bald unser *TankTool* einbauen.

Die *MyLogConsole* Klasse ist eine *JTextArea*, allerdings mit der Besonderheit, dass sie sich bei dem im *ApplicationContext* enthaltenen *LogChannel* registriert hat und deshalb alle Logmeldungen mitschreibt. Sie wurde in eine *JScrollPane* eingebettet, damit man auch größere Texte bequem handhaben kann. Der Inhalt der Konsole wird aber nicht endlos wachsen. Er ist limitiert auf eine maximale Zeilenzahl, die man mit *setBufferSize(int)* einstellen kann.

Das *ButtonPanel* wurde mit zwei *Actions* befüllt: die *ExitAction*, die aus dem *ApplicationContext* erfragt wurde und die *ClearAction,* die von der *LogConsole* bereit gestellt wird. Das *ButtonPanel* selbst erzeugt die Buttons für die Actions. Es ist äußerst praktisch, wenn Klassen das, was man mit ihnen machen kann, bereits in Form einer Action selbst zur Verfügung stellen. Man kann diese Funktionen dann sehr leicht ins GUI einbauen.

Lassen Sie uns nun unseren *TankDialog* erstellen, den wir dann mit Hilfe einer *TankAction* sichtbar schalten wollen:

```
package de.tntsoft.ewb.gui.tank;

import java.awt.*;
import javax.swing.*;
import de.tntsoft.ewb.launch.*;
import de.tntsoft.appsuite.gui.*;
import de.tntsoft.global.gui.transl.*;

/**
 * This Dialog will present our TankTool
 */
public class TankDialog extends ToolDialog {

  public TankDialog(EwbApplicationContext ctx, Component caller) {
      super(ctx, new RawNlsKey("Tank Dialog"), caller);
      add(new JButton("Here we will add the tank"));
  }

  @Override
  protected boolean hasCloseButton() {
      return true;
  }
  @Override
  protected boolean hasApplyButton() {
      return true;
  }

  @Override
  protected void performApplyAction() throws Exception {
      getApplicationContext().logInfo(this, "Apply pressed");
  }
}
```

Der Dialog beinhaltet bereits ein *ButtonPanel* mit einem Apply- und Close-Button wie es durch Überschreiben von *hasApplyButton()* und *hasCloseButton()* konfiguriert wurde. Ein Klick auf den Apply-Button führt zum Aufruf von *performApplyAction()*. Hier können wir implementieren, was dann passieren soll – in unserem Fall erst einmal eine einfache Logausgabe.

Im Konstruktor wird dem Dialog eine *caller*-Referenz übergeben. Dies ist in unserem Fall der Button des Toolbars, der den Dialog aufruft. An ihm wird er sich automatisch ausrichten. Jeder Dialog braucht zusätzlich eine Referenz auf ein Parent-Window, damit klar ist, wer über wem zu stehen hat. Wird kein Parent angegeben, dann ist dies automatisch der *EwbMainFrame*. Eine *NlsKey* Instanz definiert den Titel des Dialogs. Der hier angegebene *RawNlsKey* gibt – sozusagen als schnelle Lösung - den übergebenen Wert unverändert zurück. Er sollte später allerdings durch eine echte *NlsKey*-Konstante ersetzt werden (man kann später leicht nach allen *RawNlsKeys* suchen um sie durch Enum-Konstanten zu ersetzen).

Jetzt brauchen wir noch die *TankAction*, die den Dialog anzeigt:

```
package de.tntsoft.ewb.gui.tank;
```

```
import java.awt.*;
import de.tntsoft.ewb.launch.*;
import de.tntsoft.appsuite.gui.*;
import de.tntsoft.global.gui.transl.*;

/**
 * This action pops up the {@link TankDialog}
 */
public class TankToolAction extends ToolAction<EwbApplicationContext> {

  public TankToolAction(EwbApplicationContext ctx) {
      super(ctx, new RawActionKey("Tank"), AppSuiteIconDef.TOOL$MVC);
  }

  @Override
  protected TankDialog createWindow(EwbApplicationContext ctx, Component caller) throws Exception {
      return new TankDialog(ctx, caller);
  }
}
```

Da diese Action von *ToolAction* ableitet, brauchen wir nur *createWindow(...)* zu überschreiben und eine Instanz unseres *TankDialogs* zurückzugeben. Alles Weitere kann diese Action per se. Insbesondere wird der Dialog nur einmal erzeugt und dann intern gecached. Bauen wir die Action noch in den ToolBar des *EwbMainFrame* ein:

```
  private class MyToolBar extends ApplicationToolBar {
      public MyToolBar(EwbApplicationContext ctx) {
          super(ctx);
          add(new MemoryMonitorAction(ctx));
          add(new TankToolAction(ctx));
      }
  }
```

Nach dem Neustart enthält unser Toolbar die *TankAction* und beim Anklicken erscheint unser *TankDialog*. Ein Klick auf den Apply-Button sollte die gewünschte Logausgabe produzieren.

4 Anwendung des Model View Controller Prinzips

Jetzt geht es darum, unseren Tank zu entwickeln und zwar nach dem bewährten *Model-View-Controller Prinzip*. Dazu formulieren wir zunächst eine *TankModel*-Klasse, die die Daten unseres Tanks – das *Level* und die *Capacity* – speichert. Diese Werte werden über getter-Methoden zugreifbar und über setter-Methoden veränderbar gemacht. Allerdings soll bei jeder Änderung ein *TankEvent* gefeuert werden um eventuell daran interessierte Klassen (die Views) zu informieren.

4.1.1 Die TankEvent-Klasse

Beginnen wir mit der Formulierung eines geeigneten *TankEvents*:

```
package de.tntsoft.ewb.gui.tank;

import de.tntsoft.global.gui.event.*;
import de.tntsoft.global.util.*;

/**
 * This event carries the current values of the TankModel
 */
public class TankEvent extends GenericEvent {
  private final int capacity;
  private final int level;

  public TankEvent(int capacity, int level) {
      this.capacity = capacity;
      this.level = level;
  }

  public final int getCapacity() {
      return capacity;
  }

  public final int getLevel() {
      return level;
  }

  public interface Listener extends GenericEvent.Listener {
      void tankChanged(TankEvent e);
  }

  public interface Source {
      void addTankListener(TankEvent.Listener l);
      void removeTankListener(TankEvent.Listener l);
  }

  @Override
  public ToString toString(ToString s) {
      super.toString(s);
      s.add("capacity", capacity);
      s.add("level", level);
      return s;
  }
}
```

Diese *TankEvent*-Klasse ist nichts anderes, als eine Bean, die die aktuellen Werte des *TankModels* transportiert. Nach bewährtem Muster hat sie die *toString(ToString)* Methode zur Vereinfachung des Debuggings überschrieben. Das Interessante an dieser Klasse sind die beiden inneren Interfaces *Source* und *Listener*. Sie geben vor, wie mit diesem Event umzugehen ist. *Source* bestimmt, welche Methoden die Klasse haben soll, die dieses Event erzeugt. *Listener* schreibt vor, mit welcher Methode auf dieses Event reagiert werden soll. Unser *TankModel* ist natürlich die Klasse, die *TankEvents* erzeugt. Sie muss also *TankEvent.Source* implementieren. Alle Klassen die auf Veränderungen des Models reagieren sollen (die Views), müssen *TankEvent.Listener* implementieren.

4.1.2 Das Model-Objekt

Jetzt, wo wir das *TankEvent* haben, können wir unser *TankModel* formulieren, natürlich in guter Java-Manier erst mal als Interface:

```
package de.tntsoft.ewb.gui.tank;

/**
 * This interface methods for a tank implementation
 */
public interface TankModel extends TankEvent.Source {
  void setLevel(int level);
  int getLevel();
  int getCapacity();
}
```

Die Formulierung als Interface lässt uns die Freiheit, später mit verschiedenen Implementierungen zu experimentieren. Unsere erste Implementierung ist das folgende *DefaultTankModel*:

```java
package de.tntsoft.ewb.gui.tank;

import java.util.*;

/**
 * This model stores the tank values and fires a {@link TankEvent}
 * whenever the tank level changes.
 */
public class DefaultTankModel implements TankModel {
  private final GenericEventListenerList listenerList = new GenericEventListenerList();
  private final int capacity;
  private int level;

  public DefaultTankModel(int capacity, int level) {
      this.capacity = capacity;
      setLevel(level);
  }

  public int getCapacity() {
      return capacity;
  }

  public int getLevel() {
      return level;
  }

  public void setLevel(int level) {
      if (level < 0) {
          throw new IllegalArgumentException("Invalid level <"+level+"> - must be a positive value");
      }
      if (level > capacity) {
          throw new IllegalArgumentException("Invalid level <"+level+"> - exceeds capacity <"+capacity+">");
      }
      this.level = level;
      fireEvent();
  }

  private void fireEvent() {
      listenerList.fireEvent(new TankEvent(capacity, level));
  }

  public void addTankListener(TankEvent.Listener l) {
      listenerList.addListener(l);
  }

  public void removeTankListener(TankEvent.Listener l) {
      listenerList.removeListener(l);
  }
}
```

Unser *DefaultTankModel* speichert die beiden Werte *capacity* und *level*. Beide werden über den Konstruktor vorgegeben, wobei der *capacity*-Wert im Nachhinein nicht mehr veränderbar und daher als *final* deklariert ist. Die getter-Methoden für beide Werte sind trivial. Die setter-Methode für den *level*-Wert muss allerdings den hereinkommenden Wert prüfen. Gegen falsche Werte wehren wir uns mit einer *IllegalArgumentException*.

Nachdem ein neuer *level*-Wert gesetzt wurde, müssen alle Objekte, die an Änderungen interessiert sind (die Views) informiert werden, diese haben sich gegebenenfalls mit Hilfe von *addTankListener(TankEvent.Listener)* in der *listenerList* eingetragen. Von diesen Interessenten wissen wir nicht viel, aber eines wissen wir ganz bestimmt: sie implementieren das *TankEvent.Listener*-Interface! Also können wir in ihnen die Methode *tankChanged(TankEvent)* aufrufen. Was darin passiert, ist dann Sache des jeweiligen View-Objekts selbst. Allerdings können Sie merkwürdigerweise diesen Aufruf im oben gezeigten Code nirgends finden. Nun, hier ist etwas Magie im Spiel: die verwendete *GenericEventListenerList* erfährt vom Event selbst, welche Listener-Methode aufzurufen ist! Hier profitieren wir von davon, dass das Listener-Interface innerhalb des Events definiert wurde.

4.1.3 Das View-Objekt

Nachdem unsere *TankModel*-Implementierung fertig ist, brauchen wir ein View-Objekt, das den Tank-Füllstand graphisch anzeigen kann. Zur Visualisierung machen wir es uns einfach. Wir

„missbrauchen“ einen *JProgressBar* und machen ihn zm *TankView*:

```
package de.tntsoft.ewb.gui.tank;

import java.awt.*;
import javax.swing.*;
import de.tntsoft.global.*;

/**
 * We 'abuse' a progress bar for visualizing the tank level
 */
public class TankView extends JProgressBar {

  public TankView(TankModel tankModel) {
      super(VERTICAL, 0, tankModel.getCapacity());
      setBorder(SysConfig.getImageBorder());
      setValue(tankModel.getLevel());
      tankModel.addTankListener(new MyTankListener());
  }

  @Override
  public Dimension getPreferredSize() {
      return new Dimension(50, 150);
  }

  private class MyTankListener implements TankEvent.Listener {
      public void tankChanged(TankEvent e) {
          setValue(e.getLevel());
      }
  }
}
```

Der *TankView* bekommt im Konstruktor die Referenz auf ein *TankModel*, damit er sich mit *addTankListener(...)* dort als Interessent registrieren kann. Genau genommen registriert er sich nicht selbst, sondern eine Instanz seiner inneren Klasse *MyTankListener*, die das dazu vorgeschriebene Interface *TankEvent.Listener* implementiert. Auf diese Weise ist die *tankChanged(TankEvent)* Methode, die ja *public* deklariert sein *muss*, nicht nach außen hin sichtbar. Die Reaktion auf ein *TankEvent* ist definitiv eine innere Angelegenheit des *TankView* Objekts. Die Reaktion besteht darin, dass der *ProgressBar*-Wert auf den aktuellen *level*-Wert aus dem *TankEvent* gesetzt wird. Übrigens wurde der *ProgressBar* schon vorher im Konstruktor auf die *TankModel*-Werte *level* und *capacity* initialisiert – das sollte man nie vergessen! Bauen wir den *TankView* im Constructor unseres *TankDialogs* ein:

```
  ...
  public TankDialog(EwbApplicationContext ctx, Component caller) {
      super(ctx, new RawNlsKey("Tank Dialog"), caller);
      int intitCapacity = ctx.getApplicationConfig().getCapacity();
      int intitLevel = ctx.getApplicationConfig().getLevel();

      TankModel tankModel = new DefaultTankModel(intitCapacity, intitLevel);
      add(new TankView(tankModel), CENTER);
  }
...
```

Die Initialwerte für das *TankModel* erhalten wir aus dem *EwbApplicationConfig*-Objekt, das ja, wie Sie sich sicher erinnern, aus dem *EwbCommandLineParser* stammte. Wenn wir nun den Dialog sichtbar machen, sollten wir also bereits die per Kommandozeile eingegebenen Werte sehen.

4.1.4 Das Controller-Objekt

Nun fehlt nur noch die Controller-Komponente, die es uns ermöglicht, über die Benutzeroberfläche den Füllstand des Tanks zu verändern. Hier können wir einen *JSlider* verwenden und zum *TankController* umfunktionieren:

```
package de.tntsoft.ewb.gui.tank;

import javax.swing.*;
import javax.swing.event.*;
```

```
/**
 * This slider changes the tank level
 */
public class TankController extends JSlider {

  private final TankModel tankModel;

  public TankController(TankModel tankModel) {
      super(VERTICAL);
      this.tankModel = tankModel;
      setMajorTickSpacing(20);
      setPaintTicks(true);
      setPaintLabels(true);
      setMaximum(tankModel.getCapacity());
      setValue(tankModel.getLevel());
      addChangeListener(new MySliderListener());
  }

  private class MySliderListener implements ChangeListener {
      public void stateChanged(ChangeEvent e) {
          tankModel.setLevel(getValue());
      }
  }
}
```

Auch der *TankController* erhält die Referenz auf das *TankModel* im Konstruktor übergeben, um sich auf dessen Werte zu initialisieren. Zusätzlich muss er sich die Referenz merken, um später, wenn der *Slider* vom Benutzer bewegt wird, seine Bewegung an das Modell zu übertragen. Dazu horcht er sozusagen auf sich selbst, indem er sich eine Instanz der inneren *MySliderListener*-Klasse registriert.

4.1.5 MVC – Alles zusammen

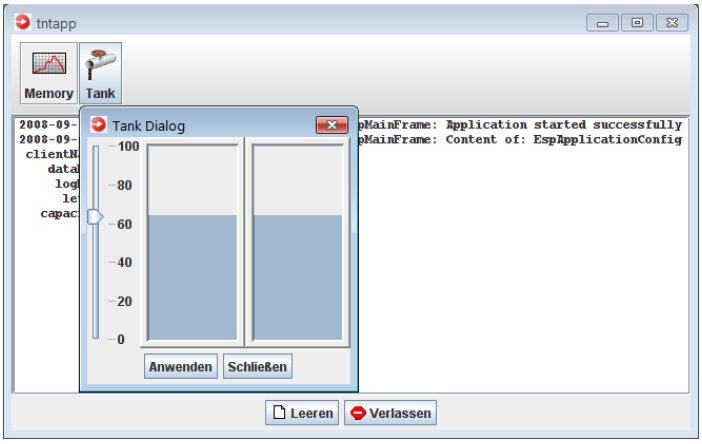
Jetzt bauen wir in unserem *TankDialog* alles zusammen:

```
  ...
  public TankDialog(EwbApplicationContext ctx, Component caller) {
      super(ctx, new RawNlsKey("Tank Dialog"), caller);

      int intitCapacity = ctx.getApplicationConfig().getCapacity();
      int intitLevel = ctx.getApplicationConfig().getLevel();

      TankModel tankModel = new DefaultTankModel(intitCapacity, intitLevel);
      add(new TankController(tankModel), WEST);
      add(new TankView(tankModel), CENTER);
      add(new TankView(tankModel), EAST);
  }
...
```

Der *TankController* wurde im West-Bereich des Frames platziert. Zusätzlich haben wir noch einen zweiten *TankView* im Ost-Bereich eingebaut, nur um zu zeigen, dass wir beliebig viele Views (natürlich auch Controller) einhängen können. Wenn sie den Controller betätigen, dann sehen sie, wie schön synchron die Views auf Änderungen reagieren. Dies funktioniert äußerst robust und mit einer hervorragenden Performance!



5 Fazit

In diesem Dokument wurde beispielhaft das zusätzliche Werkzeug *TankDialog* in eine existierende Esprit-Applikation eingebaut. Das *Tank-Tool* arbeitet nach dem bewährten Model-View-Controller Prinzip und benutzt die Event-Unterstützung des Esprit-Frameworks. Es wurden auch einfache Infrastruktur-Dienste, wie GUI-Translations, Logging und Launching verwendet. Das *EspritWorkbench* Projekt enthält eine ganze Reihe von Beispielen, die dem Leser zum Studium zur Verfügung stehen, nach dem Motto: ein funktionierendes Beispiel ist mehr wert, als eine noch so dicke Dokumentation.

Ganz allgemein gesagt: wenn Sie eine Lösung suchen, von der sie denken, dass sie so allgemein ist, dass sie bereits existieren müsste, dann ist die Wahrscheinlichkeit groß, sie im Esprit-Framework tatsächlich zu finden.