

## 5 Anwendungen mit persistenten Daten

### 5.1 Fileserver

Symbian stellt für den Zugriff auf Dateien und Verzeichnisse in einer Anwendung einen Server bereit, den Fileserver. Anwendungen richten für den Zugriff auf den Fileserver eine Session ein und verwenden vorgefertigte Funktionen. Im folgenden Abschnitt werden die wichtigsten Klassen und Abläufe für den Umgang mit dem Fileserver erläutert. Dateien bestehen dabei aus einfachen Datensätzen als Bytesequenzen. Eine höherwertige Abstraktion von Dateien sind die sogenannten Streams und ihr Format zur Speicherung, der Stream Store.

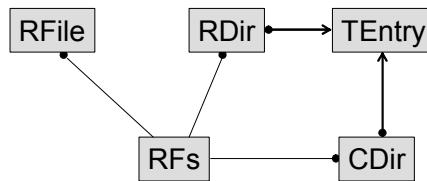
#### 5.1.1 Klassen

Der Fileserver stellt eine Anwendungsschnittstelle (Client API) zur Verfügung, welche es dem Entwickler erlaubt, Dateien und Verzeichnisse anzulegen, zu bearbeiten und zu löschen. Verzeichnisnamen werden bei Symbian durch einen Schrägstrich „\“ (Backslash) getrennt, wie bei Windows. Da das Symbian Betriebssystem auf einer Windows-Umgebung entstanden ist, finden sich einige Konventionen von Windows Dateiverzeichnissen wieder (u.a. auch die Bezeichnung des Laufwerks C: für die Festplatte). Für Dateinamenerweiterungen kann ein Punkt „.“ verwendet werden. Symbian hat keine speziellen Konventionen für Dateinamen. Es gibt jedoch Anwendungen, die diesem ihre eigene Bedeutung zuweisen. Ein Dateiname inklusive den Laufwerksbuchstaben und Verzeichnissen kann bis zu 256 Zeichen lang sein.

Abbildung 5-1 gibt einen Überblick über die Klassen des Fileservers. Die Klasse `RFs` repräsentiert eine Session eines Programms mit dem Fileserver. Diese Session wird benötigt, um Dateioperationen ausführen zu können. `RFs` ist ausserdem Voraussetzung für die Verwendung der Klassen `RFile`, `RDir` und `CDir`. Alle Server in Symbian

verwenden zur Kommunikation mit Anwendungen Sessions. Das bedeutet, dass jeder Aufruf einer Client-Funktion wie z.B. `RFs::MkDir()` oder `RFile::Write()` in eine Nachricht umgewandelt wird, die an den Server geschickt wird. Die angeforderte Funktion wird dann vom Server ausgeführt und die Ergebnisse der auf dem Server ausgeführten Operation werden an den Client zurückgeliefert. Man muss nicht verstehen, was genau im Server abläuft, um ihn verwenden zu können. Der wesentliche Punkt ist der, dass man zur Verwendung des Fileservers eine Session benötigt. Einige Hintergründe über die Kommunikation mit Servern werden später in Kapitel 6 erläutert.

**Abb. 5-1**  
Fileserver Klassen



Verzeichnisse sind in Symbian durch die Klasse `RDir` dargestellt. Ein Eintrag in einem solchen Verzeichnis ist vom Typ `TEntry`. Da es aufwändig wäre, den Fileserver nach jedem einzelnen Verzeichniseintrag zu fragen, gibt es die Klasse `CDir`, von der beim Aufruf der Methode `GetDir()` ein Exemplar als Verzeichnis zurückgeliefert wird. Man kann Attribute von Verzeichnissen ändern. Verzeichnisattribute sind beispielsweise „hidden“, „system“, „read-only“ und „archiv“. Einträge wie „hidden“ sind aus historischen Gründen vorhanden, spielen in Symbian jedoch keine besondere Rolle.

### 5.1.2 Ablauf

Was muss man tun, um mit dem Fileserver arbeiten zu können? Die wesentlichen Schritte sind die Verbindung einer Session `RFs` mit dem Fileserver, das Öffnen und Schliessen einer Datei, und das Schliessen der Session `RFs` mit Hilfe von `Close()`. Die Methode `Close()` wird verwendet, da Ressourcenobjekte (d.h. Objekte vom Typ "R") keine Destruktoren besitzen. Man kann mit einer `RFs` eine beliebige Anzahl von Dateien und Verzeichnissen öffnen. Man kann eine `RFs` auch bereits unmittelbar beim Start einer Anwendung öffnen und die Session schliessen, wenn die Anwendung beendet wird. Das CONE Framework besitzt bereits eine offene `RFs`, man muss sie also nicht einmal erst erzeugen, sondern kann sie direkt verwenden (`iCoeEnv->FsSession()`). Wenn man eine Session mit dem Fileserver been-

det, muss man darauf achten, dass man zuvor alle Ressourcen wie Dateien geschlossen hat, da man danach nicht mehr auf die Dateien zugreifen kann.

### 5.1.3 Funktionen

Die wichtigsten Funktionen, die von den Klassen in Abbildung 5-1 bereitgestellt werden, sind `SetSessionPath()`, `SetDefaultPath()`, `Open()`, `Create()`, `Replace()`, `Temp()`, `Read()`, `Write()`, `Seek()` und `Flush()`. Die Methode `SetSessionPath()` dient zum Wechseln des aktuellen Verzeichnisses. Die Festlegung des initialen Verzeichnisses für alle weiteren Sessions erfolgt mit der Methode `SetDefaultPath()`. Dabei sollte man aber darauf achten, dass man nicht das aktuelle Verzeichnis einer anderen Anwendung umbiegt, mit der man die selbe Session mit dem Fileserver teilt.

Eine geöffnete Datei wird von einem `RFile` Objekt repräsentiert. Man kann eine Datei mit einer der vier Methoden von `RFile` öffnen, von denen jede ein `RFs` Objekt als Parameter erwartet. Die Methoden zum Öffnen von Dateien sind:

- `Open()`: öffnet eine existierende Datei zum Lesen und/oder Schreiben
- `Create()`: erzeugt eine neue Datei zum Schreiben
- `Replace()`: löscht eine existierende Datei und erzeugt eine neue zum Schreiben
- `Temp()`: erzeugt eine temporäre Datei und weist ihr einen Namen zu.

Sobald man eine Datei geöffnet hat, kann man mit Hilfe der Methode `Read()` von ihr lesen und den Inhalt in einem `TDes8` Descriptor ablegen. Das Beschreiben der Datei funktioniert sinngemäss durch Daten vom Typ `TDes8` mit Hilfe von `Write()`. Mit Hilfe von `Seek()` kann man an eine bestimmte Stelle in der Datei springen. Die Methode `Flush()` schreibt serverseitige Puffer in die Datei.

Für den Zugriff auf Dateien gibt es unterschiedliche Modi: `shared read`, `exclusive write` oder `shared write`. Normalerweise legt man den Zugriffsmodus beim Öffnen der Datei fest. Der Modus lässt sich aber mit `ChangeMode()` auch bei einer bereits geöffneten Datei noch ändern. Mit `Lock()` wird der Zugriff auf die Datei für andere gesperrt. Die Sperre wird nach Beendigung der eigenen Transaktion mit `UnLock()` wieder rückgängig gemacht.

### 5.1.4 Beispiel

Den praktischen Einsatz der oben beschriebenen Funktionen zeigt folgender einfacher Programmtext:

```
RFs session = iCoeEnv->FsSession();
RFile file;
session.Connect();
file.Open(session, _L("C:\\testdatei.txt"), EFileRead|EFileWrite);
file.Write(_L8("Dies ist eine Testdatei"));
file.Close();
session.Close();
```

Wie man unschwer erkennt, zeigt das Beispiel das Schreiben in eine Datei. Zuerst wird die Session zum Fileserver geöffnet und danach die Datei geöffnet, in die geschrieben werden soll. Mit `Write()` wird ein 8 Bit Deskriptor übergeben. Dateien in Symbian arbeiten nur mit 8 Bit Deskriptoren. Nach dem Schreiben wird die Datei wieder geschlossen und danach die Session mit dem Fileserver beendet. Um das Beispiel ausführen zu können, benötigt man die Fileserver Library als Eintrag in der .MMP Datei des Projekts. Die Fileserver Library heisst `efsrv.lib`.

## 5.2 Streams und Stores

Unter einem "Stream" versteht man die serialisierte Form eines Objektes als Bytesequenz. Die Serialisierung ist in der Regel erforderlich, wenn ein Objekt übertragen oder gespeichert werden soll. Der umgekehrte Vorgang zur Serialisierung (bzw. Externalisierung) wäre die Deserialisierung (bzw. Internalisierung). Aus Sicht der internen Darstellung der Objekte gibt es also für die Serialisierung eine Schreibrichtung (Write Stream) bzw. für die Deserialisierung eine Leserichtung (Read Stream). Symbian bietet unter dem Begriff "Store" die Möglichkeit zur Verknüpfung und zur Speicherung von Streams.

### 5.2.1 Streams im Speicher

Jeder Stream in einem Store benötigt als Objektreferenz eine Kennzeichnung, d.h. eine Stream ID. Stream IDs innerhalb eines Stores sind als 32 Bit Werte realisiert. Sie können miteinander verglichen werden und in serialisierter Form auch ausgetauscht werden. Innerhalb eines Stores verhalten sich Streams im Grunde genommen wie Datensätze (Records). Mit Angabe der Stream ID lassen sie sich erzeugen, bzw. zum lesen oder schreiben öffnen.

Es sei darauf hingewiesen, dass mit einer Datei (File) hier auch einfache Bytesequenzen bezeichnet sind, also im Grunde genommen Datensätze (Records). Auch Streams sind Bytesequenzen. Ein Store enthält also als Record Store Datensätze (Records), bzw. als Stream Store Datenströme (Streams).

Ein einfacher Ablauf für den Einsatz von Stream Stores wäre beispielsweise die Erzeugung eines Objektes, die Serialisierung des Objektes als Stream und die Speicherung in einem Store. Im API Guide des SDK finden sich hierzu folgende Programmzeilen (unter Base, Store Streams):

```
TSimple thesimple;
...
... // Erzeugung eines Exemplares von TSimple
...
RStoreWriteStream outstream;
TStreamId id = outstream.CreateLC(*store);
// Serialisierung des Objektes TSimple
outstream << thesimple;
// Änderungen im Stream für gültig erklären
outstream.CommitL();
// Cleanup
CleanupStack::PopAndDestroy();
```

Mit Hilfe der Anweisung `TStreamId id = outstream.CreateLC(*store);` wurde ein Stream mit der Kennung `id` erzeugt. Der Speicher mit der Referenz `*store` existierte bereits vorher. Die Serialisierung des Objektes in den Stream erfolgt durch den Operator "`<<`". Mit der Serialisierung ist es allerdings alleine nicht getan. Die Änderung im Stream wird in der folgenden Programmzeile durch eine `Commit`-Anweisung für gültig erklärt. Diese Validierung per `Commit` ist die Basis für Transaktionen mit mehreren Objekten, die nach dem Motto "ganz oder gar nicht" entweder vollständig vollzogen werden müssen, oder vollständig unverändert bleiben müssen. In letzterem Fall kann die Änderung durch eine "Revert"-Anweisung zurückgenommen werden.

Da der Stream selber durch ein Objekt repräsentiert wird, erfolgt nach Abschluss der Transaktion die Freigabe der nicht mehr benötigter Ressourcen mit Hilfe des Cleanup Stacks in der letzten Programmzeile. Die Objektreferenz auf dem Cleanup Stack wurde bei der Erzeugung des Stream Objektes mit Hilfe der Funktion `CreateLC()` angelegt. Wenn das Objekt also nun abgeräumt wird, wo bleiben die erzeugten Daten? Wie steht es mit der Persistenz von Streams? Das hängt vom Typ des verwendeten Speichers (Stores) ab.

Stream Stores müssen nicht notwendigerweise persistent sein. Es gibt aber persistente Speicher. Für den Gebrauch der richtigen Terminologie lohnt sich ein Blick auf die beteiligten Klassen:

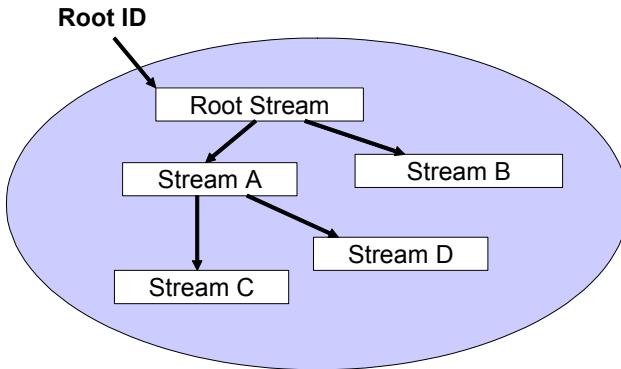
- `CStreamStore`: Das Basisinterface aller Stream Stores. Zu den Funktionen dieser abstrakten Klasse gehören die Erzeugung von Streams mit der Stream ID als Rückgabewert, das Öffnen von Streams mit Hilfe der Stream ID, das Hinzufügen und Löschen von Streams, die Validierung von Daten per Commit-Anweisung und Revert-Anweisung, sowie Funktionen zum Schreiben und Lesen von Streams.
- `CPersistentStore`: Eine Implementierung des `CStreamStore`. Wie der Name andeutet, ist dieser Speicher über die Laufzeit der Anwendung hinaus persistent. Somit muss ein Verfahren zur Konservierung der Stream IDs und der Beziehungen zwischen Streams bereitgestellt werden. Hierzu dient das im folgenden Abschnitt näher erläuterte Wurzelobjekt "Root Stream". Ausserdem stellt diese Klasse Funktionen zum Öffnen und Schliessen des Speichers bereit. Eine Spezialisierung des persistenten Speichers ist die Klasse `CEmbeddedStore`, deren Exemplare selber als Streams gespeichert werden können, sowie die Implementierungen der Klasse `CFileStore`, die es ermöglicht, Datensätze vom Typ `RFile` wie Streams zu bearbeiten.
- `CBufStore`: Eine nicht persistente Implementierung des `CStreamStore`, d.h. ein sogenannter "Memory Store" der nur im Arbeitsspeicher existiert. Eignet sich für die Bearbeitung transienter Objekte und die Bearbeitung von Puffern. Commit-Anweisungen und Revert-Anweisungen haben bei dieser Klasse keine Bedeutung.

### 5.2.2 Persistente Speicher

Ein Speicher ist persistent, wenn man die in ihm abgelegten Daten nach Beendigung der Anwendung beim erneuten Öffnen des Speichers wieder finden kann. Persistente Speicher benötigen also immer eine Referenz, auf die zurückgegriffen werden kann, um den Speicher erneut zu öffnen. Diese Referenz ist beim persistenten Speicher die Kennzeichnung (ID) des Wurzelobjektes (Root Stream), das auf die anderen Objekte bzw. Streams im Speicher verweist. Abbildung 5-2 zeigt das Prinzip.

Die Kennzeichnung (ID) des Wurzelobjekts wird beim Erzeugen des persistenten Speichers festgelegt, bevor der Speicher geschlossen wird. Beim Öffnen des Speichers wird dann mit Hilfe dieser ID erst das Wur-

zelobjekt gelesen. Die Klasse `CPersistentStore` geht aus `CStreamStore` hervor und implementiert u.a. die für den Zugriff per Root ID erforderlichen Mechanismen.

**Abb. 5-2**

Zugriff auf Streams im Speicher

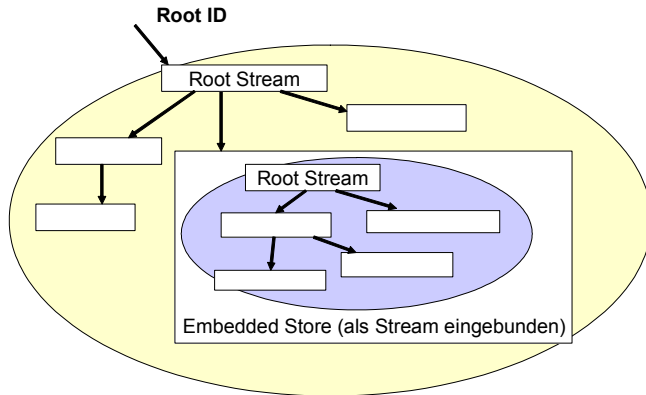
Eine weitere Art des persistenten Speichers sind File Stores. Die Klasse `CFileStore` ist eine Ableitung von `CPersistentStore`. Es wurde ja bereits festgestellt, dass Streams und Datensätze als Bytesequenzen grundsätzlich ähnlich sind. File Stores ermöglichen die Bearbeitung von Datensätzen, d.h. Exemplaren der Klasse `RFile`, als Streams. Die Klasse `CFileStore` selber ist eine abstrakte Klasse. Sie besitzt folgende Implementierungen: `CDirectFileStore` und `CPermanentFileStore`. Die Begriffe sind leider nicht besonders glücklich gewählt. Beide Typen sind permanent, da beide Klassen Ableger des `CPersistentStore` sind. Was unter "direkt" zu verstehen sein soll, bedarf ebenfalls einer Erklärung.

Ein File Store von Typ `CDirectFileStore` ist für die Bearbeitung von Datensätzen vom Typ einer Datei vorgesehen. Wie bei der Textverarbeitung üblich, lädt die Anwendung zu Beginn die Datei und speichert die bearbeiteten Daten am Ende wieder in einer Datei ab. Dabei wird in der Regel die alte Datei einfach komplett überschrieben (d.h. ersetzt). Der Typ `CPermanentFileStore` verhält sich dagegen eher wie eine Datenbank. Bei einer Datenbank werden von der Anwendung allenfalls einzelne Datensätze verändert und zurückgespeichert. Die Datenbank als solche bleibt bestehen und wird nicht etwa wie eine Datei neu angelegt und überschrieben. Der Datenbestand bleibt bei Änderungen in einzelnen Datensätzen bereits persistent. In dieser Art unterscheiden sich auch die in den File Stores angebotenen Funktionen.

Beim `CEmbeddedStore` findet sich ein Konzept wieder, das aus der "Cut & Paste" Technik mit Grafikobjekten bei der Erstellung von

Präsentationen mit einem Textverarbeitungsprogramm bekannt ist: die Gruppierung und Einbettung von Objekten in andere Objekte. Ein anderes Beispiel für solche verschachtelten Objekte wären Dokumente, die ihrerseits wiederum Textdokumente mit Bildern enthalten. Die Verschachtelung solcher Objekte wird wie in Abbildung 5-3 gezeigt auf persistente Speicher abgebildet und als "Embedded Stores" bezeichnet. Das eingebettete Objekt repräsentiert aus Sicht des übergeordneten Objekts (Host Stream) wiederum einen Stream.

**Abb. 5-3**  
Als Streams eingebettete  
Speicherobjekte



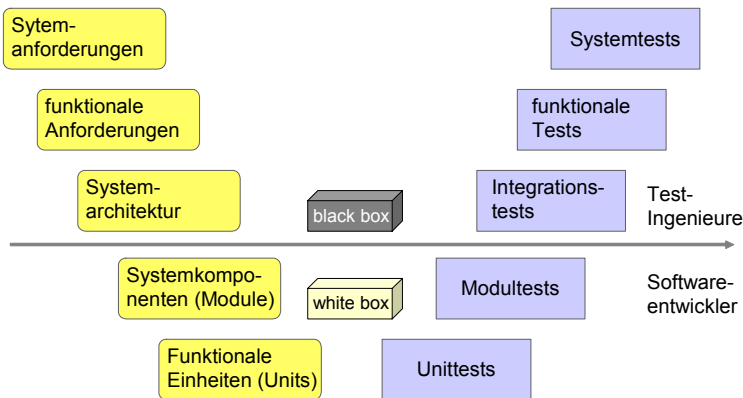
Der Vorteil einer solchen Verschachtelung liegt in der leichteren Manipulation der gruppierten Objekte. Beispielsweise lässt sich ein eingebettetes Objekt sehr leicht löschen, indem nur der eingebettete Stream gelöscht wird (anstelle der einzelnen Teilobjekte des eingebetteten Objektes). Ebenso leicht lassen sich komplexe Strukturen kopieren, indem die jeweils übergeordneten Objekte (Host Streams) kopiert werden. Sobald eine komplexe Struktur als Datei abgespeichert und geschlossen ist, lässt sich die Struktur bei einem `CEmbeddedStore` oder einem `CPersistentStore` nicht mehr verändern, d.h. nachträgliche Erweiterungen oder Löschungen einzelner Streams sind dann nicht mehr möglich. Solche datenbankähnlichen Qualitäten bietet nur der `CPermanentFileStore`.

Wie bereits beschrieben, müssen Stream Stores auch nicht unbedingt persistent sein. Die Memory Stores der Klasse `CBufStore` dienen als Pufferobjekte zur Verarbeitung komplexer Datenobjekte (z.B. Überschreiben, Ersetzen, Erweitern und Löschen von Streams). Die persistenten Speichertypen bieten auch Unterstützung für Transaktionen durch eine zweiphasige Prozedur. Änderungen im Datenstand werden erst permanent (committed), wenn alle Änderungen durchführbar sind. Andernfalls lässt sich die gesamte Transaktion rückgängig machen.

## 5.3 Testen und Debuggen

Beim Testen verwendet man das System oder eine Komponente des Systems in einem vorgegebenen Umfeld, um durch Vergleich der Systemreaktion mit den erwarteten Systemreaktionen herauszubekommen, ob das System sich gemäss dem erwünschten Systemverhalten benimmt. Abweichungen zeigen sich als Fehler im Test. Für das Testen in größeren Softwareprojekten werden hierzu Testpläne und Testfälle aufgestellt. Diese Pläne basieren auf den Anforderungen, Standards und Spezifikationen des Softwareprojekts. Fehler im Test beziehen sich auf diese Testfälle.

Das Testen von Software lässt sich grundsätzlich in zwei Kategorien aufteilen: White-Box Tests und Black-Box Tests. Bei White-Box Tests sind die inneren Details (Sourcecode) des Systems bekannt. Stellen im Sourcecode, die Fehler verursachen, lassen sich identifizieren und interpretieren. Black-Box Testen findet dann statt, wenn man nur die Spezifikationen des Systems zur Verfügung hat, aber keine Sourcecode der Software. Unit-Tests und Modul-Tests sind White-Box Tests. Funktions- und Systemtests gehören zur Kategorie des Black-Box Testens.



**Abb. 5-4**  
Systementwicklung  
und Tests

Abbildung 5-4 zeigt eine Übersicht über den Entwicklungsprozess inklusive der zugehörigen Testschritte. Während der Entwicklungsprozess bei einem sogenannten "Top-Down" Design von der Festlegung der Systemanforderungen bis zur Kodierung der funktionalen Einheiten fortschreitet, funktionieren die mit dem Entwicklungsprozess korrespondierenden Test in umgekehrter Reihenfolge. Da Unit-tests und Modultests Kenntnisse des Innenlebens der Software erfordern, werden sie in der Regel von den Software-Entwicklern durchgeführt.

Für Integrationstests, funktionale Tests und Systemtests ist eine genaue Kenntnis der Implementierung des Systems unter Umständen sogar von Nachteil. Solche Tests finden aus der Perspektive der Black-Box statt. Zu den Systemtests kann man auch Dauertests und Lasttests zählen, sowie Tests der Benutzerakzeptanz und die Bewertung der Qualität des Systems. Hier geht es vorwiegend um die sogenannten nichtfunktionalen Eigenschaften. Eine Anwendung mit einer völlig undurchschaubaren Benutzerführung kann durchaus funktionieren und somit die funktionalen Anforderungen erfüllen. Ebenso eine Anwendung, die ungefragt Verbindungen ins Internet aufnimmt, oder eine Anwendung, die sehr unwirtschaftlich mit den Betriebsmitteln haushaltet. Verändern sich einzelne Komponenten des Systems, so muss die komplette Testkette vom Unit Test bis zum Systemtest nochmals durchlaufen werden. Solche Tests nennt man Regressionstests.

### 5.3.1 Testpläne und Testfälle

*Testpläne* legen folgende Dinge fest: die Testumgebung (Rechner, Geräte usw.), die Verantwortlichkeiten (wer macht was), den zeitlichen und logischen Ablauf der Tests, den Testrahmen (was wird alles getestet). *Testfälle* geben detaillierte Instruktionen über den Ablauf jedes einzelnen Testfalls. Sie geben an, welche Dokumentation zur Überprüfung der Testergebnisse verwendet werden kann (Spezifikation etc.).

Die Testpläne und Testfälle müssen für folgende Bereiche festgelegt werden:

- Units und Module: Sourcecode und Design Dateien die getestet werden sollen
- Integration: Für die Integrationstest in den Emulator und in die echte Umgebung, sowie für Prototypen, Interaktionen und das Design der Architektur.
- Funktion: Es müssen Testpläne für die Funktionstests auf dem Emulator, dem echten Gerät, die Prototypen, die Anforderungen, das User Interface, die Spezifikation der Implementierung und Standards erstellt werden.
- Akzeptanz: Testpläne um die Akzeptanz der Benutzer zu prüfen. System: Testpläne für die endgültige Umgebung und das Gesamtsystem

Speziell für die Akzeptanz von Anwendungen lassen sich Checklisten verwenden, nach denen die Anwendung gegen vorgegebene Qualitätskriterien bewertet wird. Bewertungskriterien wären im einfachsten Fall, dass die Anwendung das Qualitätskriterium erfüllt bzw. nicht

erfüllt. Als Qualitätskriterien kommen ausser den funktionalen Anforderungen auch solche Eigenschaften in Frage, wie die saubere Installierbarkeit und Deinstallierbarkeit der Anwendung, der Schutz anderer Anwendungen und anderer Daten auf dem System, Übereinstimmung der Bedienung mit der Dokumentation, die Bedienbarkeit der wichtigsten Funktionen ohne Dokumentation, die Qualität der Benutzerführung usw. Solche Checklisten finden sich in den Entwicklerforen der Hersteller.

### 5.3.2 Unit-Tests

Bei *Unit-Tests* handelt es sich um das Testen von Software durch Software. Hier werden vor allem die folgenden Bereiche getestet:

- Konstruktion/Destruktion einer Klasse
- Spezielle Fehlersituationen: Fälle, in denen es beispielsweise zu einem Speicherengpass (out-of-memory Situation) kommen kann. Solche Fälle lassen sich gezielt provozieren, in dem z.B. das Adressbuch der Adressbuch-anwendung so voll geschrieben wird, dass kein Speicher mehr frei ist um weitere Einträge aufzunehmen.
- Bedingungen für den Ablauf: Selektionsanweisungen, Entscheidungskriterien und iterative Anweisungen (if, else, etc.) werden geprüft. Die Pfade dieser Entscheidungsanweisungen im Programm werden getestet. Solche Tests sind z.B. wichtig bei vielen verschachtelten If-Anweisungen.
- Funktionsaufrufe und Rückgabeparameter: Einzelne Module werden auf Ihre Funktion getestet.
- Überschreiten von Grenzen: Grenzwerte, Fehlerindikatoren und Null Werte werden überprüft. Was passiert beispielsweise wenn man den letzten Eintrag einer Liste löscht und anschliessend noch einen weiteren Eintrag löschen möchte?

Unit-Tests sind der effektivste Weg zur Aufdeckung möglichst vieler Fehler im Code. Studien haben gezeigt, daß Unit-Tests im Vergleich zu anderen Softwaretests das höchste Kosten-Nutzen-Verhältnis haben. Ihnen kommt daher (vor allem in sicherheits- oder missionskritischen Projekten) eine Schlüsselfunktion zu. Units sind die kleinsten Bausteine der Software. In C entsprechen die Units den Funktionen, in C++ in der Regel den Klassen (obwohl Methoden auch separat getestet werden können).

Beim Unit Test werden einzelne Softwarekomponenten (Klassen, Methoden oder Funktionen) in einer "Stand-Alone-Umgebung" getestet. Die restlichen Codeteile oder das Gesamtprojekt müssen dabei

*Bessere Codequalität  
durch Unit-Tests*

nicht vorliegen. Die Tests werden also in der Regel durchgeführt, bevor die Units in größere Module bzw. in das Gesamtsystem integriert werden. Der Vorteil dieser Methode ist, der Test "näher am Fehler". Die Fehleraufdeckung ist dadurch leichter und schneller: treten Fehler auf, müssen diese zwangsläufig in der gerade untersuchten Unit liegen.

Durch Unit-Tests werden Probleme sehr früh im Entwicklungsprozeß gefunden. Die Fehlerkorrektur ist in dieser frühen Phase noch relativ einfach und damit vergleichsweise kostengünstig. Die Aufdeckung vieler Fehler ist in späteren Entwicklungsphasen nicht oder nur sehr schwierig möglich. Auf jeden Fall gilt, daß jedes Problem um so teurer wird, je später es entdeckt wird. Ein weiterer Vorteil des Unit-Tests ist das Erreichen einer hohen Testüberdeckung.

Trotz dieser Vorteile werden Unit-Tests oft nur ungern und damit unzureichend durchgeführt, da hierfür eine spezielle Testumgebung erstellt werden muß, die oft so groß und komplex wie der zu überprüfende Originalcode ist. Da die zu testende Unit noch kein selbständig ausführbares Programm ist, wird zum Testen ein sogenannter Testharness benötigt, der zusammen mit dem zu testenden Code ein ausführbares Programm bildet. Sofern die zu testende Unit Funktionen aufruft, die noch nicht geschrieben bzw. nicht verfügbar sind, müssen diese Funktionen durch sogenannte Stubs simuliert werden. Das Schreiben des Testharnesses und der Stubs "von Hand" ist sehr mühsam und fehlerträchtig. Beim manuellen Vorgehen, muß die Testsoftware also nicht nur geschrieben, sondern selbstverständlich auch selbst getestet werden. Manuelle Unit-Tests sind kompliziert und zeitaufwendig, weshalb Unit-Tests trotz der oben beschriebenen Vorteile oft vernachlässigt werden.

### 5.3.3 Test Tools und Debugging Tools

Es gibt unterschiedliche Methoden Tests zu erleichtern. Sogenannte *Assertions* werden als Prüfschritte unmittelbar in die Software eingebaut, um die Fehler direkt sichtbar zu machen. Eine Assertion ist ein boolescher Ausdruck, der einen Zustand auf wahr oder falsch prüft. Assertions können im Debug Build verwendet werden, sie können aber auch im Code gelassen werden, um im realen Gebrauch der Anwendung zu testen oder um die Ausführung von Code in Fehlersituationen zu verhindern. Für eine Person, die Anwendungen testet, sind durch Assertions ausgelöste Abbrüche (assertion panics) einfach zu identifizieren und als Fehlermeldung zu berichten.

Durch den Einsatz eines Unit-Testtools wie CTA++ oder CTB von Testwell werden die Probleme vermieden, die beim manuellen Aufbau einer Unit-Testumgebung entstehen [Testwell]. Unit-Testtools erstellen die komplette Testumgebung bestehend aus Testharness und Stubs automatisch. Der Tester kann sich somit auf das Schreiben von Testfällen konzentrieren. Manche Entwickler erwarten von einem Testtool, daß es auch die Testfälle automatisch schreibt. Diese Aufgabe kann allerdings nicht automatisiert werden, da kein Tool selbständig feststellen kann, wie sich eine Klasse oder ein Modul verhalten soll. Jeder Testfall hat bestimmte erwartete Werte. Erst durch den Vergleich dieser erwarteten Ergebnisse mit den tatsächlichen Werten kann bestimmt werden, ob sich die jeweilige Funktion bzw. Unit in der gegebenen Situation korrekt verhält.

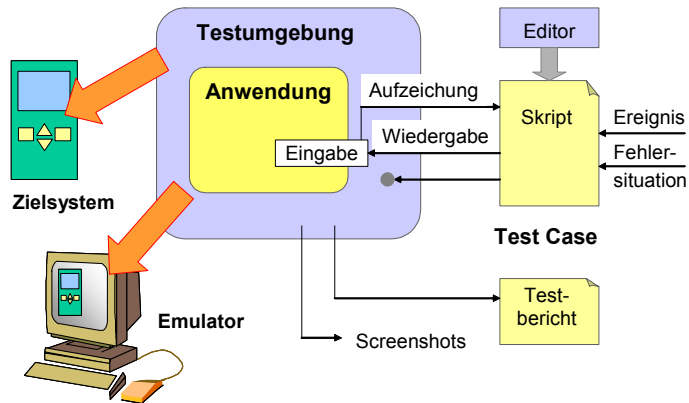
Bei jeder Modifikation einer Codekomponente müssen die bereits überprüften Tests wiederholt werden. Durch diese sogenannten Regressionstests wird sichergestellt, daß durch die Codeveränderung keine neuen Fehler eingebaut wurden. Mit Unit-Testtools kann auch diese Routine-Aktivität automatisiert werden. Die bereits existierende Testfälle vorheriger Testläufe werden durch Unit-Test-Tools automatisch dokumentiert und können für Regressionstests wiederverwendet werden.

Durch die Automatisierung wird der Testaktivität die Monotonie genommen. Der Entwickler bzw. Tester kann sich auf das Design von Testfällen konzentrieren, statt sich um mechanische und oft langweilige Koaktivitäten kümmern zu müssen. Mit dem Einsatz eines Unit-Testing-Tools wird eine höhere Codequalität in kürzerer Zeit und mit weniger Entwicklungskosten erreicht. Idealerweise sollten alle Code-teile möglichst intensiv getestet werden. Um dies sicherzustellen, sollten Unit-Testtools in Verbindung mit einem Testüberdeckungstool (Testcoveragetool) eingesetzt werden. Coveragetools zeigen ungetesteten Code an und messen die Vollständigkeit der Tests.

Die Produktivität des Testers wird erhöht, indem diesem angezeigt wird, welche Codeteile noch nicht ausreichend getestet worden sind. Das Coverageanalysetool Testwell-CTC++ unterstützt die meisten der oben beschriebenen Tests, wie den Test von Funktionen, Selektionsanweisungen und Entscheidungskriterien. CTC++ kann auch für Symbian Projekte eingesetzt werden, der Test läuft dabei im Emulator unter Windows.

Auf der Ebene der Systemtests gibt es ebenfalls Tools, wenn auch das Angebot für Symbian zum Zeitpunkt der Niederschrift noch etwas bescheiden ist. Das Grundprinzip solcher anwendungsorientierter Tools funktioniert wie in Abbildung 5-5 gezeigt. Die Anwendung wird

auf dem Zielsystem oder auf dem Emulator innerhalb einer Testumgebung gestartet. Die Testumgebung zeichnet alle Benutzereingaben bei der manuellen Bedienung der Anwendung auf. Somit ist die Testumgebung in der Lage, die Anwendung automatisch abzuspielen und einzelne Abläufe auch endlos zu wiederholen (was unter Umständen für Demos der Anwendungen ebenfalls interessant ist). Ausserdem lassen sich mit Hilfe eines Editors in den Ablauf zusätzliche Ereignisse oder Fehlersituationen einbauen, auf die die Anwendung reagieren muss. Aus den Durchläufen der Tests erstellt die Testumgebung einen Testbericht.



Ein Tool in dieser Kategorie der Systemstests ist TRY von Mobile Innovation [Mobin]. Dabei handelt es sich um ein Tool für automatisiertes Testen von Symbian OS Anwendungen. Es erlaubt mit Hilfe einer Skriptsprache Symbian Anwendungen direkt auf dem Endgerät (z.B. Nokia 7650) oder im Emulator automatisch zu testen.

## 5.4 Übung: Listen

In der letzten Übung wurde die View-Architektur gezeigt und eine Benutzeroberfläche in eine Anwendung eingebaut. Nun wollen wir diese Anwendung etwas erweitern. Dazu wird das Projekt aus der letzten Übung in Kapitel 4 wiederum geöffnet.

Zunächst wird ein Steuerelement (Control) in der Oberfläche (View) angelegt. Dazu wird eine Liste in Form einer `CEikTextListBox` eingefügt. Hierzu wird zunächst einmal in der Container Klasse im Header (hier `CViewGUIContainer3`) eine neue Member Variable angelegt:

```
private:CEikTextListBox* iListBox;
```

Für den Compiler werden noch folgende Header-Dateien hinzugefügt:

```
// Includes für die Listen
#include <aknlists.h>
#include <avkon.hrh>
```

Nun wird noch der Konstruktor der Container Klasse angepasst und darin eine Liste erzeugt. Die Liste bekommt auch gleich einen Scrollbalken, der es dem Benutzer ermöglicht, innerhalb der Liste zu "scrol-len", falls sie größer als der sichtbare Bildschirm wird:

```
void CViewGUIContainer3::ConstructL(const TRect& aRect)
{
    CreateWindowL();

    // Erzeugen einer Liste
    iListBox = new( ELeave ) CAknSingleStyleListBox();
    iListBox->ConstructL(this, EAknListBoxSelectionList);
    iListBox->CreateScrollBarFrameL( ETrue );
    iListBox->ScrollBarFrame()->SetScrollBarVisibilityL (
        CEikScrollBarFrame::EOff,
        CEikScrollBarFrame::EAuto
    );
    iListBox->SetListBoxObserver( this );
    iListBox->SetRect( Rect() );

    SetRect(aRect);
    ActivateL();
}
```

Damit die Initialisierung und das Beenden der Anwendung funktioniert, müssen noch einige weitere Methoden und der Destruktor verändert werden:

```
CViewGUIContainer3::~CViewGUIContainer3()
{
    delete iListBox;
}
TInt CViewGUIContainer3::CountComponentControls() const
{
    return 1;
}

CCoeControl* CViewGUIContainer3::ComponentControl(TInt aIndex)
const
{
    switch ( aIndex )
    {
```

```

        case 0:
            return iListBox;
        default:
            return NULL;
    }
}
void CViewGUIContainer3::SizeChanged()
{
    iListBox->SetRect( Rect() );
}

```

Abb. 5-6

Ein Fenster mit Aussicht



No data

Options      Back

Nach dem Durchlauf durch den Compiler ergibt das Starten des Programms das in Abbildung 5-6 gezeigte Ergebnis. Der Hinweis "no data" hat als Ursache, dass für die Liste noch kein Inhalt definiert ist. Für den Inhalt definieren wir ein Array und füllen dieses mit einigen Elementen aus einer Schleife. Mit dieser Ergänzung sieht der Container Konstruktor folgendermaßen aus:

```

void CViewGUIContainer3::ConstructL(const TRect& aRect)
{
    CreateWindowL();

    // Erzeugen einer Liste
    iListBox = new( ELeave ) CAknSingleStyleListBox();
    iListBox->ConstructL(this, EAknListBoxSelectionList);
    iListBox->CreateScrollBarFrameL( ETrue );
    iListBox->ScrollBarFrame()->SetScrollBarVisibilityL (
                                                CEikScrollBarFrame::EOff,
                                                CEikScrollBarFrame::EAuto
    );
    iListBox->SetListBoxObserver( this );
    iListBox->SetRect( Rect() );
    // Erzeugen und Füllen des Arrays
    CDesCArray* array = STATIC_CAST( CDesCArray*,
                                    iListBox->Model()->ItemTextArray() );

    TBuf<255> listItemFormat ( _L(" \tElement Nr. %d\t" ) );

```

```

for (TInt i = 0; i < 8; i++)
{
    TBuf<255> listItem;
    listItem.Format( listItemFormat, i );
    array->AppendL( listItem );
}

iListBox->HandleItemAdditionL();

iListBox->ActivateL();
iListBox->DrawNow();

SetRect(aRect);
ActivateL();
}

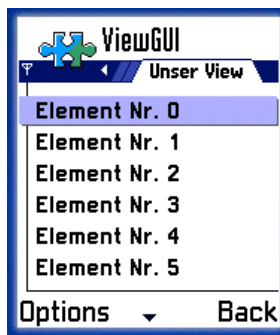
```

Damit der Compiler keine Fehler meldet, wird für das Array noch eine Library namens `bafl.lib` benötigt, die in die `.mmp` Datei eingetragen wird:

```
LIBRARY euser.lib apparc.lib cone.lib eikcore.lib
```

```
LIBRARY eikcoct1.lib avkon.lib bafl.lib
```

Anschliessend wird die Projektdatei neu erzeugt durch die Anweisung `abld makefile vc6`. Nach dem Durchlauf durch den Compiler sollte sich bei der Ausführung auf dem Emulator nun das in Abbildung 5-7 gezeigte Ergebnis zeigen.



**Abb. 5-7**

Fenster mit Auswahlliste

Wenn man nun die Pfeiltasten des Emulators verwendet, um in den Listen Elementen zu wechseln, stellt man fest, dass nichts passiert. Es fehlt offensichtlich noch die Ereignisbehandlung (Event-Handling Routinen). Die Pfeiltasten des Emulators lösen als Ereignisse Key-Events aus. Damit der Container diese Events auch bekommt, muss die

Methode `OfferKeyEventL()` implementiert werden, die der Container von `CCoeControl` erbt. Das Framework ruft dann beim Drücken einer Taste auf der Klasse die `OfferKeyEventL()` Methode auf. In dieser Methode werden die Key-Events für die Pfeiltasten für "Aufwärts" und "Abwärts" an die `ListBox` weiter gereicht.

```

TKeyResponse CViewGUIContainer3::OfferKeyEventL(
                                                    const TKeyEvent& aKeyEvent,
                                                    TEventCode aType )
{
    if ( aType != EEventKey )
    {
        return EKeyWasNotConsumed;
    }
    switch ( aKeyEvent.iCode )
    {
        case EKeyUpArrow:
        case EKeyDownArrow:
            if ( iListBox )
            {
                return iListBox->OfferKeyEventL( aKeyEvent, aType );
            }
            break;
        case EKeyOK:
            iEikonEnv->InfoMsg(_L("KeyEvent Ok"));
            break;
        default:
            break;
    }
    return EKeyWasNotConsumed;
}

```

Nun reagiert die Liste auch auf Tastendrücke der Pfeiltasten für "Aufwärts" und "Abwärts" und "scrollt" auch auf und ab. Die KeyCodes

sind übrigens in den Header `uikon.hrh` und `e32keys.h` definiert. Man sollte möglichst `KeyCodes` und keine `ScanCodes` verwenden, da bei der Nokia Implementierung eventuell mehrere `ScanCodes` einen `KeyCode` ergeben. Hier ein paar Beispiele für `KeyCode` Definitionen bei Nokia. Welche Knöpfe damit gemeint sind, lässt sich durch ausprobieren testen:

```
#define EKeyOK EKeyDevice3
#define EKeyCBA1 EKeyDevice0
#define EKeyCBA2 EKeyDevice1
#define EKeyPhoneSend EKeyYes
#define EKeyPhoneEnd EKeyNo
#define EKeyApplication EKeyApplication0
#define EKeyPowerOff EKeyDevice2
#define EKeyGripOpen EKeyDevice4
#define EKeyGripClose EKeyDevice5
#define EKeySide EKeyDevice6
```

Wenn man direkt Events von der Liste bekommen möchte, beispielsweise wenn mit dem Stift auf ein Listenelement geklickt wird, so verwendet man die Methode `SetListBoxObserver()` um den Container bei der `ListBox` als `Observer` zu registrieren. Damit das allerdings auch funktioniert, sollte unser Container noch von der Klasse `MEikListBoxObserver` erben und die Methode `HandleListBoxEventL()` überschreiben. `HandleListBoxEventL()` wird dann bei Auftreten eines Event auf unsere `ListBox` vom Framework aufgerufen.

```
void CViewGUIContainer3::HandleListBoxEventL(
    CEikListBox* /*aListBox*/,
    TListBoxEvent aEventType )
{
    iEikonEnv->InfoMsg(_L("Handle ListBox Event"));
}
```

## Literatur

[Harris03] Richard Harrison, Symbian OS C++ for Mobile Phones, John Wiley & Sons, Chichester, ISBN 0-470-85611-4, 2003

[JUnit] Vincent Massol, Ted Husted, JUnit in Action, Manning Publications Co., ISBN 1920110995, Nov. 2003

[Testwell] Testtool für die Anwendungsentwicklung, <http://www.testwell.fi>

[Mobin] Testtool für die Anwendungsentwicklung, <http://www.mobinnovation.co.uk>

