Dr. Hartmut Schorrig, www.vishia.org

Wiederholbarkeit der Softwaregenerierung, Source-Zuordnung, Prüfcodes der Binaries

Ablaufende Software wird von Binaries bestimmt, sie sind die eigentlichen Arbeitsobjekte. Was dort abläuft, ist schwer bis gar nicht einzusehen, man sieht nur das Verhalten von außen.

Es gibt diesbezüglich mehrere Themen. Eines davon ist die Beeinflussung der Ablauf-Binaries mit Schadcode, also unerkannte Änderung der Binaries. Ein weiteres Thema sind Softwarefehler und Lücken, die entweder von Schadcode ausgenutzt werden können (‚*vulnerability*') oder unentdeckte und selten auftretene Softwarefehler, die aber Schaden anrichten können.

Die Binaries selbst werden häufig von Check-Codes begleitet (MD5 usw), und sind damit in ihrer Integrität geschützt. Die anderen Themen sind aber weitgehend offen.

Inhaltsverzeichnis

1. Sources und Binaries

Wie einleitend beschrieben, den Binaries selbst sieht man nicht an, wie sie eigentlich, im Inneren funktionieren. Lediglich das Verhalten nach außen ist aus isolierter Betrachtung eines Binaries bekannt.

Nun ist die Einsatzzeit von Binaries oft sehr hoch, Jahre und Jahrzehnte. Entsprechend groß ist das Vertrauen auf ihre Funktionsfähigkeit, mit langjähriger Erfahrung belegt.

Gibt es nun Rückfragen an die innere Funktionalität, sei es aus Gründen der Aussage zu Lücken der Sicherheit, sei es für eine Fehleranalyse, dann werden die zugehörigen Sourcen benötigt.

Welche Sources zu den Binaries gehören, ist nun mit einem ordentlichen Prozess der Binary-Erstellung, den man voraussetzen möge, über diverse Versionsmanagement-Systeme und deren Checkpoints geklärt. So eine gewisse Erwartungshaltung.

Ein Rückschluss aus den Binaries auf die faktische Richtigkeit der Source-Zuordnung kann zwar teilweise erbracht werden, man ist aber hier eher auf dem Gebiet der Tiefenanalyse der Aufklärung unterwegs. Die probaten Mittel sind beispielsweise:

* Listingfiles bei C/++-Assembler-Compilierung. Die Listingfiles enthalten genau den Binärcode. Über Mapfiles können die verschiedenen Teilmodule im Binary zugeordnet werden. Zugehörig dazu die Assemblerfiles,die temporär bei der Compilierung entstehen.
* Rückübersetzung aus Java-Bytecode und Abgleich mit den Quellen.

Jedoch wird in der Praxis eher wenig Listings und Assembleroutputs gespeichert, eher noch Map-Files, da diese oft für Debug-Datenzugriffe herangezogen werden.

Die Aussage, „*diese und jene Sources sind genau die Sources, die den Binaries zugrunde liegen*“ sind möglicherweise eine Behauptung, deren Nachweis ebenfalls eine Behauptung ist: „D*er Checkpoint stimme ja, man habe nach Prozess gearbeitet.*“ Ein Nachweis ist dies nicht.

2. Was beeinflusst die Binaries bei der Generierung

Primäre Bedingungen:

* Die Sources selbst.
* Optionen der Generierung
* Die Generiertools (Compiler, Codegenerierung der Grafischen Programmierung)
* Möglicherweise auch das eingesetzte Betriebssystem.

Die Sources selbst sind dabei noch einmal mehrfach geteilt:

* Die Sources selbst:
  * Die Sources der jenigen Hauptkomponente.
  * Benötigte abhängige Komponenten, die möglicherweise als fertige Library, also wieder als Binary eingezogen wurden, dann nach Auflösung deren primäre Bedingungen. Einfacher ist es, wenn alle abhängigen Module ebenfalls als Sources vorliegen und in einem Ablauf mit dem Hauptmodul übersetzt wurden.

    * Sources die dem Generiertool zugeordnet sind, also für den C/++-Bereich Systemheader, System-Libraries.

Die Optionen der Generierung sind ebenfalls mehrfach zu betrachten. Optionen der Generierung sind für den C/++-Bereich beispielsweise Compileroptimierungseinstellungen. Im Zusammenhang mit den Systemsources zählen dazu auch eingestellte Includepfade, eingezogene Libraries beim Linken etc.

* Optionen der Generierung:
    * Mit den Sources eingecheckte Optionen, die häufig in Makefiles stehen.
    * Optionen, die als „Defaulteinstellung“ genutzt werden.
    * Optionen die möglicherweise von Umgebungsvariablen des Systems bestimmt sind.

Es dürfte klar abgezeichnet sein, dass es trotz Einhaltung eines genauen Prozesses der Softwareerstellung in dem einen oder anderen Fall nicht alle Bedingungen der Generierung der Binaries aufgezeichnet wurden. Dazu kommt noch ein möglicher manueller Handlungsfehler. Beispielsweise werden häufig Details noch einmal nachgebessert, und nicht in allen Fällen in einer schnell-lebigen Umgebung (nennt man Hektik, oder „*im Eifer des Gefechts*“) richtig eingecheckt.

Zur Erinnerung: Es geht häufig um Bestand-Sources, die vor Jahren von anderen Kollgen eingecheckt wurden.

3. Prinzip der Nachgenerierbarkeit von Binaries

Eigentlich wäre zu erwarten, dass die Generierung von Binaries aus den gleichen Sources bei gleichen Generiertools zu den gleichen Ergebnisfiles führt. Das heißt, die neu generierten Binaries sollten bytekompatibel sein.

Leider wird diesem Prinzip in der Praxis keine genügende Bedeutung beigemessen, obwohl es essentiell ist in der Frage der Softwarepflege. Es gibt prinzipiell die Möglichkeit der bytekompatiblen Nachgenerierung. Diese Möglichkeit wird aber teils durch eigentlich unnötige Dinge erschwert, teils sind etwas aufwändigere aber nur einmalig zu programmierende Aufwände notwendig:

* Bei der Generierung von Quellen aus Grafischen Programmierungen wird häufig das Generierdatum, der Login-Name am Generierrechner oder weitere Metainformationen hinzugesetzt, aus dem Ansinnen heraus dass es geloggt werden sollte, wann wer generiert hat. Der Prozess der Generierung ist aber unwesentlich für das Binary, wesentlich sind nur die verwendeten Sources. Solange diese Informationen nur in Kommentare in den Sources fließen, wird gegebenenfalls ‚nur‘ eine unnötige Neu-Generierung angestoßen.
* Selbiges Verfahren der Protokollierung des Generierdatums und des Ausführenden finden sich auch in einigen Kopfdaten der Binaries.
* In den Objectfiles werden häufig die Informationen zum Auffinden der Quellen mit absolutem Pfad eingeschrieben. Das ist für das Debuggen notwendig. Wenn von einem anderen Verzeichnisort generiert wird, dann sind teils auch die Binaries unterschiedlich, weil durch die eigentlich als Metainformationen in den Objects eingespeicherten Pfade Optimierungsergebnisse abhängen. Hier hilft beispielsweise die Generierung immer von gleichen Verzeichnis aus. Dies kann durch einen symbolischen Link im Verzeichnissystem erreicht werden, unter Windows sozusagen ab dessen Root kann das auch ein umgeleitetes Laufwerk sein (subst-Befehl).
* Die Timestamps der erzeugten Files beeinflussen teilweise das Binary. Das ist jedenfalls und beispielsweise so bei den jar-Files der Java-Generierung.

4. Vorteile der Nachgenerierbarkeit von Binaries für ein bekanntes Produkt vor Änderungen

Bei einem gegebenen Binary möchte man ggf. einen Fehler beseitigen. Dabei soll möglichst wenig Aufwand entstehen. Aber die Anforderungen und die Erwartung an die korrekte Funktionsfähigkeit sind möglicherweise hoch.

Wenn man davon ausgeht, dass die Sources zwar ordnungsgemäß gespeichert wurden, insbesondere aber bei einer Neugenerierung eines älteren Softwarestandes mit Miss-Ständen gerechnet werden muss, dann ist a priori nicht davon auszugehen, dass bei einer Nachgenerierung ein Binary mit genau der selben bekannten Funktionalität entstehen. Man kann zwar damit rechnen, aber sicher ist dies nicht.

Als Beispiel soll nur das Thema „`volatile`-Kennzeichnung von Daten“ herangezogen werden. Häufig wird die notwendige Verwendung einer `volatile`-Kennzeichnung insbesondere in Alt-Software vergessen. Man stellt diesen eigentlichen Bug beim Testen nicht fest, weil in den kritischen Abläufen der Wert einer Variablen vom Speicher aus anderen Gründen, auch ohne `volatile`-Kennzeichnung erneut geladen wird. Wird nun mit einer neuen Compilierung versehentlich mit einem anderen Optimierungslevel gearbeitet, weil diese Compilereinstellung doch nicht so klar abgespeichert wurde, dann tritt ein bisher vorhandener Fehler jetzt erst zu Tage und führt zu Fehlreaktionen des Binaries. Gleiches kann passieren, wenn Module in der Reihenfolge anders zusammengesetzt werden und es dadurch zu anderen Optimierungen kommt.

Wird bei der Nachgenerierung genau das gleiche Binärergebnis erzeugt, dann ist genau die gleiche Funktionalität gegeben. Das steht außer Frage, wenn auch die Abarbeitungsmechanismen identisch sind. Letzteres kann sich aber auch ändern, beispielsweise mit einer neuen Hardwareversion eines Prozessors mit einem korrigierten Fehlverhalten, dies nur als Anmerkung.

Wird auf der Basis der in der Nachgenerierung sich identisch gezeigten Sources eine kleine Änderung eingebracht, beispielsweise ein Parameterwert anders gesetzt, eine Bedingung hinzugefügt oder nur eine Negation oder andere Skalierung programmiert, dann kann man davon ausgehen, dass die Funktionalität der Binaries auch nur in diesem einen Punkt geändert ist. Hilfreich dabei ist es, wenn bestimmte Bereiche der Binaries, die den Software-Quell-Modulen zuordenbar sind, also Object-Files oder class-Files in Java, bei unveränderten Sources tatsächlich unverändert sind, und nur diejenigen Bereiche in Binaries geändert sind, die auch in den Sources geändert sind. Weiterhin hat man bei der Nachgenerierung vor der Änderung die Zwischenfiles erhalten, die in der Vergangenheit zumeist nicht mit abgespeichert worden sind. Man kann über die Zwischenfiles (Assemblerfile, Listing, Mapfile) genau zuordnen, an welcher Stelle und selbst ***wie*** die Änderung in das Binary eingebracht worden ist.

Diese Informationen sind äußert hilfreich, um beispielsweise dem Kunden nachzuweisen, dass das bisher eingesetzte und bekannte Produkt nur an genau den erwarteten Stellen geändert worden ist. Daraufhin kann auf teils sehr umfangreiche, zeitintensive und damit teure Gesamt-Tests verzichtet werden.

5. Vorteile der Nachgenerierbarkeit von Binaries für die Source-Zuordenbarkeit

Soll beispielsweise im Nachhinein ein Verhalten bewertet werden, dann kann über eine Nachgenerierung zweifelsfrei nachgewiesen werden, dass die vorhandenen Sources auch tatsächlich dem eingesetzten Binary mit seinem bekannten Verhalten zuordenbar sind. Es kann beispielsweise sein, dass die Möglichkeit eines Hacker-Angriffs auf bestehende eingesetzte Software bewertet werden soll. Wenn man davon ausgeht, dass die Richtigkeit der Zuordnung der Sources eingentlich immer nur eine Behauptung ist, und kein wirklicher Nachweis, dann bleibt die Möglichkeit der Fehlvermutung offen. Nur die Nachgenerierung die das identische Binary abliefert, ist der tatsächliche Beweis.

6. Aufbewahren einer Generierumgebung – oder Nachgenerierung mit neuer Generierumgebung

Es wird teils ein beträchtlicher Aufwand getrieben, um eine einstmals vorhandene Generierumgebung für die „Altlastenpflege“ aufzubewahren. Dazu gehört der damalig eingesetzte Compiler, mit dem verwendeten Betriebssystem, aber auch Testumgebungen.

Es ist nun möglich, vom Compilerhersteller abzuverlangen, dass eine neue Toolversion, lauffähig unter einer neuen Betriebssystemumgebung, sich mit bestimmten Einstellungen genauso wie die alte Toolversion verhält, also gleiche Binaries erzeugt. Es geht hierbei um Cross-Compiling, das Betriebssystem und das Tool sind also nur Werkzeug-Umgebungen. Wenn die neue Compilerversion besser optimieren kann, neuere Prozessoren unterstützt und dergleichen, dann kann sie ohne weiteres auch die älteren Prozessoren mit nicht so sehr optimierten Code genauso unterstützten wie Vorversionen des Tools. Das widerspricht keineswegs einem Fortschrittsgedanken. Man muss aber im Einzelfall dieses Verhalten vom Toolhersteller verlangen.

Damit kann jeweils beim Wechsel der Toolumgebung (Betriebssystem, Tool als solches) über die Nachgenerierung aller bekannter und bei Kunden zu pflegenden Binaries nachgewiesen werden, dass dies kompatibel und byteidentisch auch mit der neuen Version möglich ist. Möglicherweise muss dazu etwas an den Generierfiles nachgebessert werden (andere Aufrufe, andere Formulierung der Options), alles in allem dürfte sich der Aufwand jeweils aber in Grenzen halten.

Damit braucht nicht die alte Umgebung aufbewahrt werden, sondern beim Wechsel ist jeweils der Nachweis zu erbringen, dass die Generierbarkeit auch in der neuen Umgebung ablaufen kann und bytekompatible Binaries erzeugt. Mit dem neuen Tool sind gegebenenfalls Vorteile verbunden, schnellerer Ablauf, bessere Log-Möglichkeiten und dergleichen, alles in allem also nur Vorteile.

Ein Wechsel einer Tool- und Betriebssytemumgebung ist häufig aller paar Jahre zutreffend. Aber auch „Development and Operation“ im Toolbereich ist möglich, wenn bei den kleinen erwarteten Tooländerungen der Nachweis der identischen Generierbarkeit automatisiert erfolgt, Nachtlauf.

7. Thema der identischen Generierbarkeit am Beispiel eines jar-Files (Java-Binary)

Bei der Generierung eines Java-Binaries (jar) werden zunächst alle Java-Files neu generiert zu class-Files. Diese enthalten den Java-Bytecode. Es ist aufgrund der Standardisierung des Bytecodes zu erwarten, dass dieser auch bei verschiedenen Java-Versionen identisch ist. Der Java-Compiler ist also relativ verträglich was die byteidentische Nachgenerierbarkeit betrifft.

Das jar-File ist ein zip-Archiv der class-Files. Neben den class-Files selbst ist ein sogenanntes „Manifest“ enthalten, das Zusatzinformationen enthält. Man kann das jar-File als Zip-file aufklappen, und die dort enthaltenen class-Files und das Manifestfile binär vergleichen – und wird Identität bei der Nachgenerierung feststellen. Das ist zunächst einmal sehr gut.

Dem Endanweder interessiert aber nicht der aufklappbare Inhalt (dies wäre für einen Nachweis interessant) sondern das Binary, das ist der jar-File, als solcher. Diese jar-Files sind häufig für den Bezug aus dem Internet mit Prüfcodes gesichert (MD5, SHA1, SHA256). Der Prüfcode bezieht sich auf die Binärdarstellung des jar-Files, nicht auf dessen aufklappbaren Inhalt.

Es ist nun interessant, über die gleichen Quellen einen jar-File nachzugenerieren, dessen Prüfcode ebenfalls identisch ist. Somit ist der Nachweis erbracht, dass ohne Änderung der Bezugssysteme, also der Zugriffe auf eine Internet-Datenbasis (jar-Archive) mit Prüfcode die Sources tatsächlich identisch sind. Wird dem Nutzer neben dem prüfcodegesicherten jar-File zusätzlich ein Source-zipfile geliefert, dann kann er selbst ohne große Aufwändungen durch Nachgenerierung prüfen, dass die Sources dem eingesetzeten jar-File genau entsprechen.

Problem dabei sind eigentlich nur die Timestamps der Files. Der Inhalt ist identisch. Aber ein Zipfile speichert neben dem Inhalt auch den Zeitstempel der Files. Damit ist bei identischem Inhalt der Einzelfiles der Inhalt des zip-Archivs (das jar-File ist ein zip-Archiv) leider nicht mehr identisch.

Nutzt man die üblichen Java-Generiersysteme, also beispielsweise eine Generierumgebung in Gradle (www.gradle.org), dann hat man auf diese Dinge keinen Einfluss. Wie einleitend erwähnt wird der binärkompatiblen Generierung offiziell und kommerziell keine Bedeutung beigemessen. Man kann aber auch in Gradle das folgende Generierscript aufrufen. Dessen essentieller Befehl für Binary-Kompatiblilität ist `touch`. Der zweite notwendige Punkt ist die nicht automatische Erzeugung des Manifest-Files, sondern dessen manuelles kopieren.

Das Script ist ein Unix-Shell-Script:

```
#shell script to generate jar file
#it can be run under Windows using MinGW: sh.exe - thisScript.sh
#MinGW is part of git, it should be known for Gcc compile too.

echo compile java and generate jar with binary-compatible content.
echo Java-Tools at $JAVAC_HOME
echo time stamp and version = $VERSION
echo primary sources to compile in file $PRIMARYSRCFILE
echo sourcepath: $SRCPATH
echo classpath : $CLASSPATH
echo temparary files at $TMPJAVAC
echo jar-file to generate = $JARFILE
echo MD5-file to generate = $MD5FILE

if test "$JAVAC_HOME" = ""; then
  echo you must set JAVAC_HOME in your system to the installed JDK
  exit 5
```

```
fi
# clean the binjar because maybe old faulty content:
if test -d $TMPJAVAC/binjar; then rm -f -r -d $TMPJAVAC/binjar; fi
mkdir -p $TMPJAVAC/binjar

if ! test "$SRC_ALL" = ""; then
  echo gather all sources, at $SRC_ALL
  find $SRC_ALL -name "*.java" > $TMPJAVAC/sources.txt
  export FILE1SRC=@$TMPJAVAC/sources.txt
fi
echo compile javac
$JAVAC_HOME/bin/javac -d $TMPJAVAC/binjar -cp $CLASSPATH -sourcepath $SRCPATH $FILE1SRC
mkdir $TMPJAVAC/binjar/META-INF
##Note: create the manifest file manually, not with jar, because of time stamp
cp $MANIFEST $TMPJAVAC/binjar/META-INF/MANIFEST.MF
echo touch timestams to $VERSION
find $TMPJAVAC/binjar -exec touch -d $VERSION {} \;
echo build jar
$JAVAC_HOME/bin/jar -cvfM $JARFILE -C $TMPJAVAC/binjar . > $TMPJAVAC/jar.txt
if ! test "$MD5FILE" = ""; then echo output MD5 checksum
  md5sum -b $JARFILE > $MD5FILE
fi
echo ok $JARFILE
```

Dieses Script ist generell gültig, kann erweitert werden bezüglich javac-Aufruf und dergleichen.

Das Script wird aus einem shell-Script gerufen, dass die Umgebungsvariable übersichtlich setzt. Dieses Script ist angepasst auf die jeweilige Source/Generierumgebung:

```
#this file is the user-adapt-able frame for makejar_vishiaBase.sh
#edit some settings if there are different form default.

export TMPJAVAC=$TMP/javac_vishiaBase/build/javac
export JAVAC_HOME=c:/Programs/Java/jdk1.8.0_211

# SWT for Windows-64
##export CLASSPATH=../../../libs/org.eclipse.swt.win32.win32.x86_64_3.110.0.v20190305-0602.jar;…
export CLASSPATH=xx
export VERSION=2020-03-17
export JARFILE=../vishiaBase-$VERSION.jar.zip
export MD5FILE=../vishiaBase-$VERSION.jar.MD5

# located from this workingdir as currdir for shell execution:
export SRCPATH=..
export MANIFEST=MANIFEST.MF
# FILE1SRC=../org/vishia/jztxtcmd/JZtxtcmd.java
export SRC_ALL=..

#now run the common script:
./makejar.sh
```

In diesem Fall handelt es sich um die Generierung aus srcJava_vishiaBase-2020...zip. Man kann dieses Zip-File entpacken und auf dieser Stelle compilieren. Umgebungsangepasst muss `JAVAC_HOME` gesetzt werden. Das ist die notwendige Anpassung an die Systemumgebung. Man sieht aber in der Vorgabe, welche Java-Version für das zugehörige Jar-File verwendet wurde. Die kommentierte Zeile für `CLASSPATH` zeigt, wie mit abhängigen jar-Files umgegangen werden kann. Sie müssen benannt werden und passend möglichst mit relativem Pfad auffindbar sein. Die aktuelle Angabe `xx` ist notwendig, da die Argumentangabe in der `javac`-Aufrufzeile irgendwie besetzt werden muss, auch wenn es in diesem Fall keinen classpath gibt.

Der `FILE1SRC` könnte, nicht hier, die unmittelbaren relativen Pfade zu den primären Sources der javac-Compilierung enthalten: Ein javac-Compiler holt sich die notwendigen abhängigen Sources (mit `import` in den Java-Quellen bezeichnet) on demand aus dem angegebenen `SRCPATH`. Man braucht also nur bestimmte `path/to/MyClass.java` anzugeben. In diesem Fall wird aber ähnlich wie beim *Gradle*-Standardverhalten vorgegangen: Es wird mit `SRC_ALL` bestimmt, wo alle `*.java` aufzufinden sind, die primär compiliert werden. Im `makejar.sh`-Script mit dem `find`-command alle vorhandenen `*.java` zusammengestellt und dann `FILE1SRC` als „Input aus File“ angesagt.

Insgesamt sollte die Generierumgebung trotz möglicher komplexer Abhängigkeiten noch überschaubar sein. Diese hier gezeigten Files sind mit der Version eingecheckt und auch im Source.zip enthalten. Die manuelle Aufrufumgebung gilt in diesem Fall für die Generierung aus den gegebenen Sources im zipfile ohne jede weitere Aufwändungen. Lediglich ein JDK (Java Development Kit) muss vorhanden sein.

Ein Aufruf aus gradle, der neben diesem etwa zusätzlich Javadoc, Asciidoctor, Tests, Deployment organisiert, sieht mit diesem Anspruch wie folgt aus:

```
task jcc_main(type: Exec) {
 workingDir 'src/main/java/_make'

 environment('TMPJAVAC', '../../../../build/javac')
 environment('CLASSPATH', 'xx')
 environment('VERSION', version)   //should have the form of a date, yyyy-mm-dd
 environment('JARFILE', '../../../../build/libs/vishiaBase-'+version+'.jar')
 environment('MD5FILE', '../../../../build/libs/vishiaBase-'+version+'.md5')
 environment('SRCPATH', '..')            //located in the workingDir
 environment('MANIFEST', 'MANIFEST.MF')  //located in the workingDir
 environment('SRC_ALL', '..')            //located in the workingDir

 executable 'sh'
 args '-c', './makejar.sh'
}
```

Dies ist im `build.gradle` eine Task, die gerufen wird, beispielsweise um ein Deployment zusammenzukopieren:

```
task copyJar(type:Exec) {
  dependsOn jcc_main
  dependsOn srcZip
```

In dieser `jcc_main`-Task werden genau die Umgebungsvariable gesetzt, die auch in dem oben vorgestelltem manuellen Aufrufscript versorgt wurden. Nur hier wird der gradle-Worktree-Aufbau benutzt. Die eigentlichen Sources, die mit folgender task gezipt werden:

```
task srcZip(type: Zip) {
    archiveFileName = 'vishiaBase-'+version+'-source.zip'
    destinationDirectory = file("libs")

    from "src/main/java"
    include "_make"
    include "org"
}
```

stehen unter `src/main/java`. Dort steht auch das `_make`-Subdirectory, das ebenfalls, speziell für die Nachgenerierung, mit gezipt wird. Für die gleiche Aufrufsituation des `makejar.sh` wird das `src/main/java/make` als `workingDir` gesetzt.

Alle angegebenen Paths in den Environment-Variablen müssen dann relativ zu diesem working directory angegeben werden. Die Ergebnisfiles werden wie in gradle üblich in `build` abgelegt, was also als `../../../../build` angegeben werden muss. Anmerkung: Das `build` zeigt beim Verfasser

über `$TMP` auf einen RAM-Disk-Bereich, die build-Files sind also wirklich temporär. Sie werden während des gradle-Build in entsprechende Deployment-Verzeichnisse kopiert.

8. Thema der identischen Generierbarkeit bei C/++-Compilierung

Dieses Thema wird in Zukunft hier genauer erläutert werden, am Beispiel eines bestimmten Compilers einer Embedded Plattform. Der Verfasser hatte in einem firmeninternen Thema mit dem ADSP-Tigersharc-Prozessor und dem damit verbundenen Compiler jahrelang erfolgreich identisch generiert. Der Trick dabei war, den Arbeitsbereich (Working directory) während der Compilierung auf ein substituiertes B:-Laufwerk zu legen. Damit wurde dieser Path der Sources in die Objects eingeschrieben. Diese waren damit unabhängig von der PC-Umgebung und dem Datum identisch. Das Datum wurde nicht in die Object-Files eingeschrieben, es spielte im build-Prozess keine Rolle.

Allerdings wurde kein übliches make-File benutzt, sondern über einen Generator (JZtxtcmd) wurde ein Batch-Script (Windows) erzeugt mit dediziertem Compileraufruf. Dazu gehört ein Dependency-Checker, der feststellt, ob eine Neugenerierung notwendig ist.

In dieses Batch- oder Shell-script können dann bei Bedarf entsprechende touch-Befehle mit integriert werden falls notwendig.

Dieses Gesamtthema würde diesen Artikel sprengen und muss auch neu aufgesetzt werden, da die Vorgängerarbeiten firmenintern waren.

9. Ein Wort zu Dependencies

Häufig werden Tools eingesetzt, die Dependencies automatisch auflösen. Wenn eine Library eine weitere braucht, dann steht das dort drin. Das Laden der notwendigen Libraries erfolgt aus dem Internet, über bestimmte Repositories, die sich als zentrale Zugriffsorte etabliert haben.

Der Anwender wird damit entlastet. Er braucht sich um nichts zu kümmern, scheinbar. Ihm wird die Verantworutng für Dependencies scheinbar abgenommen, es scheint ja alles gut zu sein.

Als Folge entstehen immer mehr Abhängigkeiten, die eigentlich in Verantwortung bekannt sein sollten aber dann nicht mehr überschaubar sind. Ein Nebeneffekt: Module werden automatisch geladen, man muss am Internet sein. Sind die geladen, kann man offline arbeiten, die Files sind beispielsweise unter

```
c:\Users\hartmut\.gradle\caches\modules-2\files-2.1\org.asciidoctor\asciidoctorj-groovy-dsl\
1.0.0.Alpha2\f15d0828dc9f12c241020beafd540aefa9bad956\*.jar
```

gespeichert. Zum einen schaut man dort nicht hin, zum anderen ist das derart viel, dass man auch gar nicht mehr hinschauen möge, und zum dritten funktioniert nichts mehr, wenn man den User wechselt und nicht am Internet sein kann. Ist man am Internet, so werden Files doppelt kopiert, was allerdings selbst bei der Menge kein Problem ist, da SSD-Festplatten immer noch größer sind als die Files. Tendenz bei beiden steigend.

Das ist die Situation im Java-Tool-Bereich auf dem PC. Beim Softwareeinsatz in C/C++ in embedded Systemen ist die Überschaubarkeit noch ein Thema, aber nur weil der Speicherplatz auf den Geräten oft etwas beschränkt ist. Folgt man der Tendenz, immer mehr Leistung, dann wird möglicherweise auch hier der Durchbruch in die Nichtüberschaubarkeit folgen. Im C/++-Bereich kann etwa Libraries toolgestützt dazu laden, dessen Inhalt man gar nicht mehr kennen will.

Oft ist es so, dass für die eigentliche Arbeit nur Weniges benötigt wird. Folgt man der Device „Es gibt schon alles im Internet“, dann erkauft man sich mit der Nutzung von kleinen Dingen komplexe Abhängigkeiten. Es ist also im Konkretfall abzuwägen:

* Ein System includieren, weil man vieles daraus gebrauchen kann und dessen Abhängigkeiten nicht ausufern
* Ein System eben nicht includieren wegen ausufernden Abhängigkeiten und statt dessen gegebenenfalls
* von den Quellen ausgehen, möglicherweise nicht notwendiges reduzieren
* oder einfache überschaubare Dinge in Eigenverantwortung ggf. selbst schreiben oder aus offenen Quellen kopieren (dabei die Nutzungsrechte beachten) und anpassen.

Der letztere Punkt kann und sollte gegebenenfalls dazu führen, dass die vereinfachten Algorithmen (Beschränkung auf das Notwendige) in die Open-Source-Gemeinschaft zurückgegeben werden und unter dem Motto „*Weniger ist Mehr*“ ein von anderen nutzbares System bildet.

*****