automatische tests

von michael klar

mehr zum thema:
www.klar-gmbh.de

ERFAHRUNGSBERICHT AUTOMATISCHE TESTS

Mit Hilfe von generativer Programmierung (GP) lassen sich bei der Entwicklung von Software viel Zeit und Kosten sparen. Dass sich die generative Programmierung auch auf im Umfeld von Softwaretests sehr gut einsetzen lässt, verdeutlicht dieser Erfahrungsbericht.

der autor

Michael Klar (E-Mail: michael.klar@klar-gmbh.de) ist Geschäftsführer der Klar Automation GmbH und verfügt über langjährige Erfahrungen in objektorientierter Softwareentwicklung und Coaching, gerade auch im Zusammenhang mit technischen Systemen.

Moderne Wasser- und Wärmezähler sind mit Mikrokontrollern ausgestattet. Um die Zähler zu testen, wurde von der Klar Automation GmbH in Kooperation mit einem Kunden eine Prüfstandssoftware erstellt.

Die Mikrokontroller in den Zählern arbeiten mit eigenen Datentypen. Es ist sehr wichtig, dass bei den Rechenoperationen mit den eigenen Datentypen keine Fehler auftreten. Die eigenen Datentypen wirken sich zwangsläufig auch auf die Prüfstandssoftware aus. Beim Test der Prüfstandssoftware selbst musste daher nachgewiesen werden, dass diese richtig mit den mikrokontroller-spezifischen Datentypen rechnet.

GP für Testfälle

Trotz Modulen, Klassen und Komponenten gibt es häufig noch viele Teile in der Software, die nahezu ähnliche Codepassagen aufweisen. Das liegt daran, dass die „innere Gleichförmigkeit der Software" häufig übersehen wird.

Bei der generativen Programmierung ist ein Softwaresystem von vornherein darauf ausgerichtet, dass es von einem Basis-Softwaresystem unterschiedliche Ausprägungen gibt. Diese Varianten lassen sich an Hand von Merkmalen beschreiben. Man spricht in diesem Zusammenhang auch von System- oder Produktfamilien, zu denen die einzelnen Varianten gehören. Die Varianten werden mittels Generatoren automatisch erzeugt. Als Input dient dazu die Vorgabe der konkreten Merkmale, die mittels einer domänenspezifischen Sprache (*Domain Specific Language, DSL*) beschrieben werden.

Auch in Bezug auf Testfälle lassen sich Produktfamilien finden — man könnte diese als *Testfallfamilie* bezeichnen. GP lässt sich mit einfachen, standardisierten Mitteln am besten mit *XML (Extensible Markup Language)* und *XSLT (Extensible Stylesheet Language Transformation)* umsetzen. Dieses Mittel wurde auch in dem hier dargestellten Projekt gewählt.

GP bei den Testfällen der Prüfstandssoftware

Im Folgenden wird erläutert, wie sich die Grundprinzipien der GP auf die Testfälle der Prüfstandssoftware übertragen lassen.

Die Systemfamilie

Das Domänen-Problem eines Testfalls lässt sich wie folgt beschreiben: Die zu testende Funktionalität wird ausgeführt. Als Eingangsdaten werden sinnvolle Testdaten verwendet. Das resultierende Ergebnis wird mit einem zu erwartenden Ergebnis verglichen.

Um die Systemfamilie der Testfälle zu realisieren, wird demnach ein Generator-

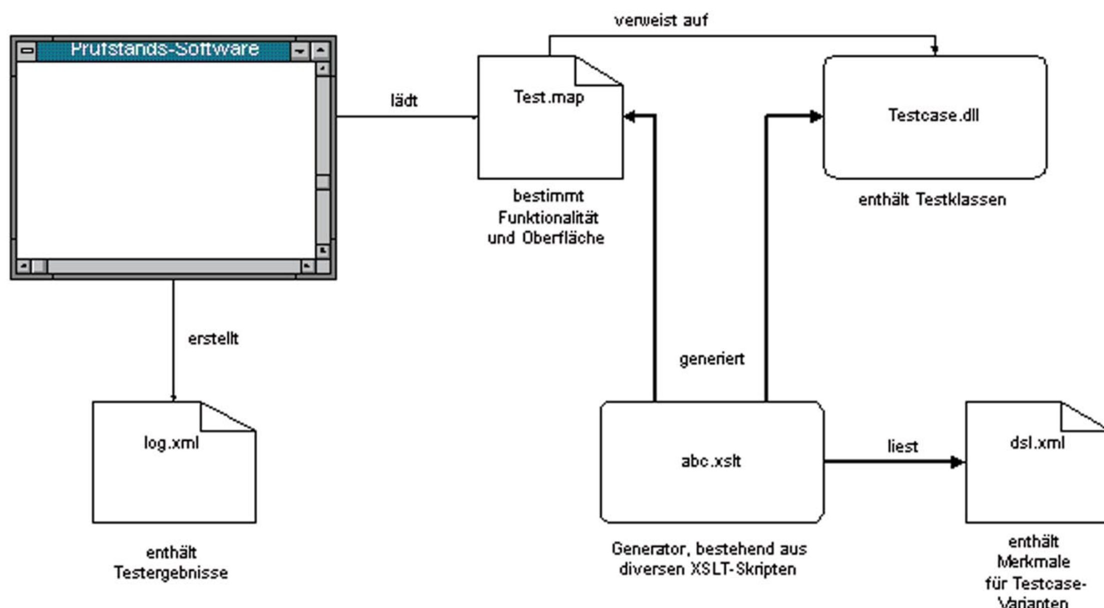

Abb. 1: Zusammenspiel zwischen Prüfstandssoftware und generierten Testfällen

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
<test name="TestCmds" number="">
<test name="HY_DATE_Test1">
<test-case op="-">
<op1 type="DATE" value="23.12.10"/>
<op2 type="HEX_BIT" value="2" msk="afafaf"/>
<res type="DATE" value="21.12.10"/>
</test-case>
<test-case op="-">
<op1 type="DATE" value="23.12.10"/>
<op2 type="HEX_BIT" value="212" msk="afafaf"/>
<res type="DATE" value="25.05.10"/>
</test-case>
</test>
</test>
```

Listing 1: *DSL-Vorgabe für Generator (Ausschnitt)*

Op1 \ Op2	HEX_BIT	DEC_BIT	SIGNED_BIT	MBCD	BCD
HEX_BIT	_int64	_int64	_int64	_int64	_int64
DEC_BIT	_int64	_int64	_int64	_int64	_int64
SIGNED_BIT	_int64	_int64	_int64	_int64	_int64
MBCD	_int64	_int64	_int64	_int64	_int64
BCD	_int64	_int64	_int64	_int64	_int64
DATE	DATE	DATE	DATE	DATE	DATE
DATE_TIME	DATE_TIME	DATE_TIME	DATE_TIME	DATE_TIME	DATE_TIME

Tabelle 1: *Zu testende Datentypen bei +/-*

system benötigt, das einen korrespondierenden Test-Quellcode in die zu testende Software integriert.

Die Prüfstandssoftware selbst ist eine Windows-Applikation, deren Funktionalität und optisches Aussehen durch eine eigens dafür konzipierte Konfigurationsdatei bestimmt wird (diese wird im Projektjargon als „Map-File" bezeichnet, daher die Endung .map **in Abb. 1**). In der Konfigurationsdatei werden die DLLs und Klassen angegeben, die zur Laufzeit geladen werden sollen, sowie die prinzipiellen *Controls*, die für das Layout auf der Oberfläche zur Verfügung stehen. Durch diesen Softwareaufbau ist es demnach auch möglich, Klassen mit Test-Quellcode zu integrieren. Dafür muss lediglich eine entsprechende Konfigurationsdatei existieren. **Abbildung 1** zeigt das Zusammenspiel zwischen der zu testenden Prüfstandssoftware und der Generierung der Testfälle.

Die generierten Testfälle (Testfallvarianten) folgen dem einheitlichen Schema (Systemfamilie):

- Anlegen des ersten Operanden,
- Anlegen des zweiten Operanden,
- Rechenoperation durchführen und im Ergebnis speichern,
- Ergebnis protokollieren.

Die DSL

Als Syntax für die DSL wird XML verwendet. In der DSL werden die einzelnen Merkmale beschrieben, die zur Generierung des jeweiligen Testfalls benötigt werden: die durchzuführende Rechenoperation, die beiden Operatoren (Datentyp und Wert) und das erwartete Ergebnis.

Listing 1 zeigt einen Ausschnitt aus der .xml-Datei, die als DSL-Input für den Generator dient.

Eine besondere Herausforderung bei den Testfällen dieses Projekts lag in der Kombinatorik von Datentypen und Rechenoperation. **Tabelle 1** zeigt, welche Kombinationen von Datentypen bezogen auf die Rechenoperatoren „+" und „-" getestet werden sollten. In der Zelle steht jeweils der resultierende Datentyp, z.B. (eingefärbt) DATE + HEX_BIT = DATE.

```
<?xml version="1.0" encoding="ISO-8859-1"
    standalone="yes"?>
<test-cases>
  <datatype name="HEX_BIT">
    <type>
      <mask>afafaf</mask>
      <value>2</value>
      <value>212</value>
      <value>131068</value>
    </type>
    <type>
      <mask>ffff</mask>
      <value>2</value>
      <value>212</value>
      <value>65033</value>
    </type>
    <operator>
      <symbol>+</symbol>
    </operator>
    <operator>
      <symbol>-</symbol>
    </operator>
  </datatype>
  <datatype name="DATE">
    <type>
      <value>23.12.10</value>
    </type>
    <type>
      <value>1.1.1</value>
    </type>
    <type>
      <value>1.3.88</value>
    </type>
  <operator type="DATE">
      <symbol>-</symbol>
      <not>DATE</not>
      <not>DATE_TIME</not>
    </operator>
    <operator type="DATE">
      <symbol>+</symbol>
      <not>DATE</not>
      <not>DATE_TIME</not>
    </operator>
  </datatype>
</test-cases
```

Listing 2: *DSL_1 beschreibt Datentypen (Ausschnitt)*

Ähnliche Tabellen existierten als Vorgabe für die Rechenoperationen von Punkt-Operatoren (*, %, /) und Vergleichsoperatoren.

Die Domäne in diesem Testprojekt konzentriert sich also aus Kundensicht zunächst auf die Datentypen. Daher wurde die DSL mehrstufig, in insgesamt fünf Schritten, aufgebaut. Die einzelnen DSL-Dateien wurden von 1 bis 5 durchnummeriert. Im ersten Schritt wurde eine DSL aus Kundensicht eingeführt (DSL_1). **Listing 2** zeigt einen Ausschnitt dieser DSL-Datei, die aus Gründen der Übersichtlichkeit nur die Angaben bezogen auf den Fall aus **Tabelle 1** enthält.

DSL_1 aus **Listing 2** ist wie folgt zu lesen: Der Datentyp DATE liefert bei Verknüpfung mit dem Symbol „-" ein Ergebnis vom Typ DATE. Dabei kann er mit allen Datentypen als zweiten Operanden (Op2) verknüpft ▸

Symbol	Anzahl Testwerte Op1 (DATE)		Anzahl Testwerte Op2 (HEX_BIT)		Testfälle gesamt
-	3	*	6	=	18
+	3	*	6	=	18
					36

Tabelle 2: *Berechnung der Anzahl der Testfälle*

```
void CDATE_Test1_1::TestCases(int RamNr)
{
  CKAutoLog log("test-results");
  log->SetLogOn(true);
  log << "<test-results>";
  {
    // case 1: testing - operator
    DATE op1(RamNr,"DATE");
    HEX_BIT op2(RamNr,"HEX_BIT2afafaf");
    DATE res(RamNr,"DATEres");

    op1="23.12.10";
    op2="2";
    res=(op1 - op2);

    log << "<result res=\"" << res << "\" target=\"21.12.10\"
    doc=\"DATE  (23.12.10) - HEX_BIT afafaf (2)  =  DATE \" />";
  }
  {
    // case 2: testing - operator
    DATE op1(RamNr,"DATE");
    HEX_BIT op2(RamNr,"HEX_BIT2afafaf");
    DATE res(RamNr,"DATEres");

    op1="23.12.10";
    op2="212";
    res=(op1 - op2);

    log << "<result res=\"" << res << "\" target=\"25.05.10\"
    doc=\"DATE  (23.12.10) - HEX_BIT afafaf (212)  =  DATE \" />";
  }

  log << "</test-results>";
  log.Update();
}
```

Listing 3: *Ausschnitt aus generierter Datei „Test1_1.cpp"*

werden, die ebenfalls das Symbol „-" aufweisen. Ausnahme dabei sind: DATE und DATE_TIME.

Durch die Werte-Angaben in DSL_1 ergeben sich 36 Testfälle (**vgl. Tabelle 2**). Hier ein Hinweis zu den Datentypen: DATE speichert ein Datum. Der Datentyp HEX_BIT ermöglicht eine Speicherung auf Bitebene im Mikrokontroller. Um eine Zuordnung zwischen der Darstellung im Mikrokontroller und der Darstellung in der Prüfstandsoftware zu erhalten, wird der Wert über einen so genannten Maskenwert normiert (*Tag* <mask>).

Die kundenspezifische DSL (**Listing 2**) wird in fünf Schritten in die testfallspezifische DSL (**Listing 1**) umgewandelt. Dabei werden alle Testfälle aus sämtlichen Kombinationen ermittelt. Zusätzlich wird das erwartete Ergebnis rechnerisch ermittelt.

Auf die Validierung der XML-Dateien durch ein XML-Schema wurde aus Zeit- und Aufwandsgründen verzichtet. Bei größeren Projekten ist eine derartige Validierung auf jeden Fall immer in Betracht zu ziehen, vor allem bei der ursprünglichen DSL-Datei. Versehentliche Eingabefehler werden auf diese Weise sofort ersichtlich.

Der Generator

Da die DSL im XML-Format vorliegt, bietet sich als Generatorsprache XSLT an.

Aus den Vorgaben der testfallspezifischen DSL (**Listing 1**) wird der Quellcode für die Testklassen generiert, die anschließend in einer Testcase.dll zusammengefasst werden. Zusätzlich wird die korrespondierende Map-Datei zum Laden der Testcase.dll und ihren Klassen erstellt. **Listing 3** zeigt einen Ausschnitt aus einer Testklasse, wie sie generiert wurde.

Wie dort zu sehen ist, wird das Ergebnisprotokoll wiederum im XML-Format gespeichert. Es enthält neben dem berechneten Wert (res) auch den erwarteten Wert (target), der aus der testfallspezifischen DSL (**Listing 1**) übernommen wurde.

In einem weiteren Evaluierungsschritt wird ebenfalls durch ein XSLT-Skript der res-Wert mit dem target-Wert verglichen. Übrig bleiben die Zeilen, in denen die beiden Werte nicht übereinstimmen, also die nicht erfolgreichen Testfälle.

Fazit

Die Anwendungsfälle von GP sind vielfältig. Überall dort, wo Software eine gewisse Gleichförmigkeit aufweist, lässt sich immens viel einsparen. Es gilt hier den Blick zu schärfen, um entsprechende Systemfamilien zu identifizieren. Bei dem vorgestellten Projekt wäre ein entsprechend umfangreicher Test in der Kürze der Zeit ohne Verwendung des generativen Ansatzes gar nicht möglich gewesen. ■