Masterarbeit

CODEGENERIERUNG FÜR RUNDTAKTANLAGEN

ausgeführt am

Fachhochschul-Masterstudiengang
Automatisierungstechnik-Wirtschaft

von
Florian Spitzer, BSc
1610322013

betreut und begutachtet von
FH-Prof. Dipl.-Ing. Dieter Lutzmayr

Graz, im Jänner 2018 .....

Unterschrift

EHRENWÖRTLICHE ERKLÄRUNG

Ich erkläre ehrenwörtlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die benutzten Quellen wörtlich zitiert sowie inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

.....

Unterschrift

DANKSAGUNG

Ich widme diese Arbeit meiner Mutter Susanne, welche mich immer zu Weiterbildungen ermutigt hat aber leider das Ende meines Studiums nicht mehr miterleben durfte. Du bleibst unvergessen.

Vielen Dank an meine Frau Victoria dafür, dass sie mir in den letzten Jahren den Rücken frei gehalten hat damit ich mich auf mein Studium konzentrieren konnte. Danke auch an meine Tochter Lea für ihr Verständnis, dass ich an den Wochenenden nicht immer Zeit für sie hatte.

Einen herzlichen Dank auch an meine ganze Familie, welche mich seit dem Beginn meines Bildungsweges unterstützt hat.

Ebenfalls danke ich meinem Freund Hannes für das Korrekturlesen der Masterarbeit.

Darüber hinaus bedanke ich mich bei meinen ehemaligen Arbeitskollegen der Programmierabteilung von Schunk Hoffmann Carbon Technology für ihr Mitwirken bei der Entwicklung des Codegenerators.

Abschließend bedanke ich mich bei meinem Betreuer Herrn FH-Prof. Dipl.-Ing. Dieter Lutzmayr, für die hilfreichen Anregungen und die konstruktive Kritik bei der Erstellung dieser Arbeit.

KURZFASSUNG

Das Unternehmen Schunk Hoffmann Carbon Technology ist Weltmarktführer in der Produktion von elektrisch leitenden Kohlebürsten für Gleichstrommotoren. Die Finalfertigung des Produkts findet auf selbstgebauten Maschinen statt, sogenannten Rundtaktanlagen. Für ein neues Projekt kopiert die Programmierabteilung des Maschinenbaus ein vorhandenes Programm einer Rundtaktanlage und ändert es entsprechend den Bedürfnissen der herzustellenden Anlage ab. Dieses Vorgehen birgt die Gefahr Fehler zu übersehen und sie erst bei der Inbetriebnahme der Maschine zu entdecken. Darüber hinaus ist die Prozedur monoton und beansprucht Zeit, welche der Entwickler anderweitig für das Projekt nutzen könnte.

Das Ziel dieser Masterarbeit war es einen Codegenerator zu entwickeln, der automatisiert ein benutzerdefiniertes Grundprogramm für eine Rundtaktanlage erstellt. Ein Generator hat den Vorteil, dass keine Fehler aufgrund des Kopierens alter Anwendungen auftreten und er die zeitaufwendige manuelle Nacharbeit eliminiert.

Um eine parametrierbare und wiederverwendbare Vorlage für den Codegenerator zu erhalten, fand eine Analyse der bisherigen Rundtaktanlagen statt. Nachdem die Programmierabteilung die Anwendungen der Anlagen in einer Entwicklungsumgebung der Firma Beckhoff erstellt, wurden Methoden zur automatisierten Erzeugung von Programmcodes für diese Applikation evaluiert.

Das Resultat der Masterarbeit ist der Codegenerator Code Monkey. Ein Mitarbeiter der Maschinenbauabteilung erstellt einmalig eine Vorlage für eine Rundtaktanlage. Der Generator bereitet anschließend das Template für den Bediener auf und dieser kann es entsprechend seinen Anforderungen konfigurieren. Aufgrund der frei wählbaren Benutzereingaben ist es dem Generator möglich aus einer Vorlage das Grundprogramm für eine Vielzahl an unterschiedlichen Rundtaktanlage zu erzeugen.

Mit der Applikation Code Monkey ist der Maschinenbau der Firma Schunk Hoffmann Carbon Technology zukünftig im Stande Programme für Rundtaktanlagen in nur wenigen Schritten generieren zu lassen und gleichzeitig die Entwicklungszeiten für neue Anlagen zu reduzieren.

ABSTRACT

The company Schunk Hoffmann Carbon Technology is the world leader in producing electrically conductive carbon brushes for DC motors. The final manufacturing of the product is done by a so called round table machine. These machines are built by the company's engineering department. Until now the programming department copied an existing round table machine program and altered it according to the new machine's requirements. This method carries the danger of overlooking mistakes and not detecting them until the commissioning of the machine. Moreover, the procedure is monotonous and time consuming for the programmer.

The aim of this master thesis was to develop a code generator which automatically creates a user defined basic program for a round table machine. As a result, mistakes should not occur due to the copying of existing programs anymore and the time consuming manual rework should be outsourced to the generator.

In order to obtain a reusable and configurable template for the code generator an analysis of the previous round table machines was carried out. As the programming department uses a development environment of the company Beckhoff to write the machine application, methods to automatically generate codes for the development environment were evaluated.

The result of the master thesis on hand is the code generator Code Monkey. With the help of this code generator an employee of the engineering department creates a template for a round table machine once. The generator prepares the template for the user and he is able to parameterize it according to his needs. Because of the freely selectable user input, the generator can produce the basic program for a variety of round table machines from just one template.

In the future, the engineering department will be able to generate the program for a round table machine in a few steps with the application Code Monkey, while at the same time the development time for a new machine can be reduced.

INHALTSVERZEICHNIS

1 EINLEITUNG

Ziel dieser Masterarbeit ist es einen Codegenerator für Rundtaktanlagen (RTA) der Firma Schunk Hoffmann Carbon Technology (SHCT) zu erstellen. Dazu soll zunächst das Unternehmen sowie ein RTA vorgestellt werden, um die Beweggründe zur Umsetzung eines Codegenerators und seine möglichen Vorteile darzulegen.

1.1 Schunk Hoffmann Carbon Technology

Schunk Hoffmann Carbon Technology ist auf die Herstellung von elektrisch leitenden Kohlenstoffprodukten spezialisiert. Die Beschreibung des Unternehmens und ihrer Erzeugnisse ist angepasst der Firmenhomepage entnommen:[1]

Das Unternehmen wurde 1946 von Dr. Franz Hoffmann gegründet und ist seit 1999 Teil der Schunk Gruppe. Schunk ist weltweit führend in den Bereichen Ultraschallschweißen, Kohlenstoff-, Klima- und Sintermetalltechnik. Durch die Kooperation mit einer Vielzahl von Schunk-Standorten weltweit hat sich Hoffmann zu einer Qualitätsmarke in den Bereichen Automobil und Bahn etabliert.

Der Geschäftsbereich Automotive ist zuständig für die Produktion von Kohlebürsten, welche in Elektromotoren für Fahrzeuge verbaut werden. Sie stellen den Kontakt zum Kommutator her. Das Produktportfolio umfasst Elektrokohlen für Kraftstoffpumpen (siehe Abb. 1.1), Starter-, ABS- und Zusatzmotoren für den Komfortbereich z. B. Fensterheber oder Sitzversteller. Um den hohen Anforderungen der Automobilbranche gewachsen zu sein, setzt das Unternehmen auf eine firmeneigene Materialentwicklung, Produktionsprozessdefinition und Maschinenkonstruktion.

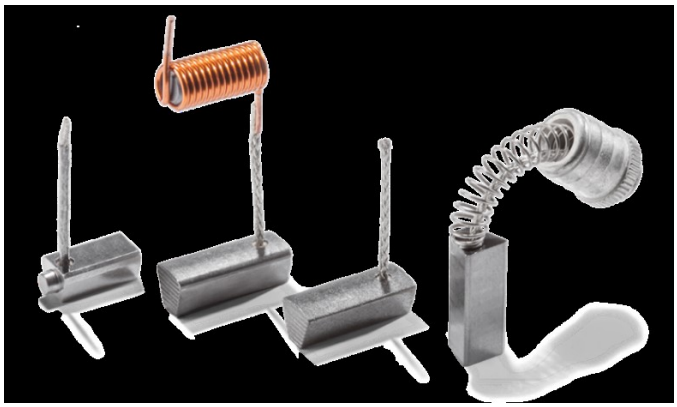


Abb. 1.1: Kohlebürsten für Kraftstoffpumpen, Quelle: Schunk Carbon Technology (o.J.), Online-Quelle [1.12.2017].

Der zweite Geschäftsbereich, Current Transmissions – Railways, konzentriert sich auf die sichere und zuverlässige Stromversorgung für den Nah- und Fernverkehr. Für eine einwandfreie Übertragung von der Oberleitung bzw. Stromschiene zum Straßen- oder Schienenfahrzeug sind Kohlenstoff-Schleifstücke notwendig (siehe Abb. 1.2). Je nach Bauart des Stromabnehmers bietet SHCT das passende Kohlenstoff-Schleifstück an.

[1] Vgl. Schunk Hoffmann Carbon Technology AG (o.J.), Online-Quelle [1.12.2017].

Abb. 1.2: Kohlenstoff-Schleifstück für den Schienenverkehr, Quelle: Schunk Carbon Technology (o.J.), Online-Quelle [1.12.2017].

Neben den beiden oben genannten Bereichen produziert das Unternehmen die Aluminiumgraphite ALG 1808 und ALG 2208. Es handelt sich dabei um einen Verbundwerkstoff auf Basis von Elektrographit und Aluminium. Diese Materialien zeichnen sich durch geringe Dichte, hohe Wärmeleitfähigkeit und einem niedrigen thermischen Ausdehnungskoeffizienten aus. Der Werkstoff lässt sich einfach mechanisch bearbeiten und ermöglicht dadurch die Herstellung von Produkten nach Kundenwunsch. Dazu gehören Kühlplatten bzw. Kühlkörper (siehe Abb. 1.3) sowie Ronden oder Ringe für Dioden und Thyristormodule.

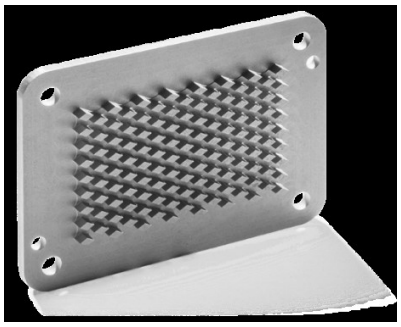


Abb. 1.3: Kühlkörper aus Aluminiumgraphit, Quelle: Schunk Carbon Technology (o.J.), Online-Quelle [1.12.2017].

1.2 Rundtaktanlagen in der Produktion bei SHCT

Die Produktion von Elektrokohlen für den Automotive-Bereich (siehe Abb. 1.1) durchläuft vier Schritte. Zuerst wird in Mühlanlagen Graphitpulver mit verschiedenen Zusatzstoffen hergestellt. Eine Presse bringt unter hoher Krafteinwirkung das Pulver in Form. Als Nächstes härten die Kohlenrohlinge in einem Ofen aus, dabei gehen die unterschiedlichen Materialien im Graphitpulver eine Verbindung ein. Abschließend erfolgt die Finalbearbeitung der Kohlebürsten auf einer Rundtaktanlage.

Eine RTA besteht aus 6 bis 16 Stationen. Der Arbeitsablauf beginnt immer bei der Zuführstation, welche die Kohlen über einen Fördertopf vereinzelt und die Werkstückträger des zentralen Rundtakttellers bestückt (in Abb. 1.4 rot umrandet). Die nachfolgenden Bearbeitungsschritte sind produktspezifisch. Hier sollen anhand einer Anlage typische Vorgänge vorgestellt werden. Nach der Zufuhr erfolgt ein Ausstreifen und Abschneiden der Kupferlitze (in Abb. 1.4 gelb umrandet). Anschließend fräst eine CNC-Station (Computerized Numeric Control) die Elektrokohle auf Maß und versieht die Lauffläche mit Rillen um einen besseren Bürsteneinlauf zu erreichen (in Abb. 1.4 grün umrandet).

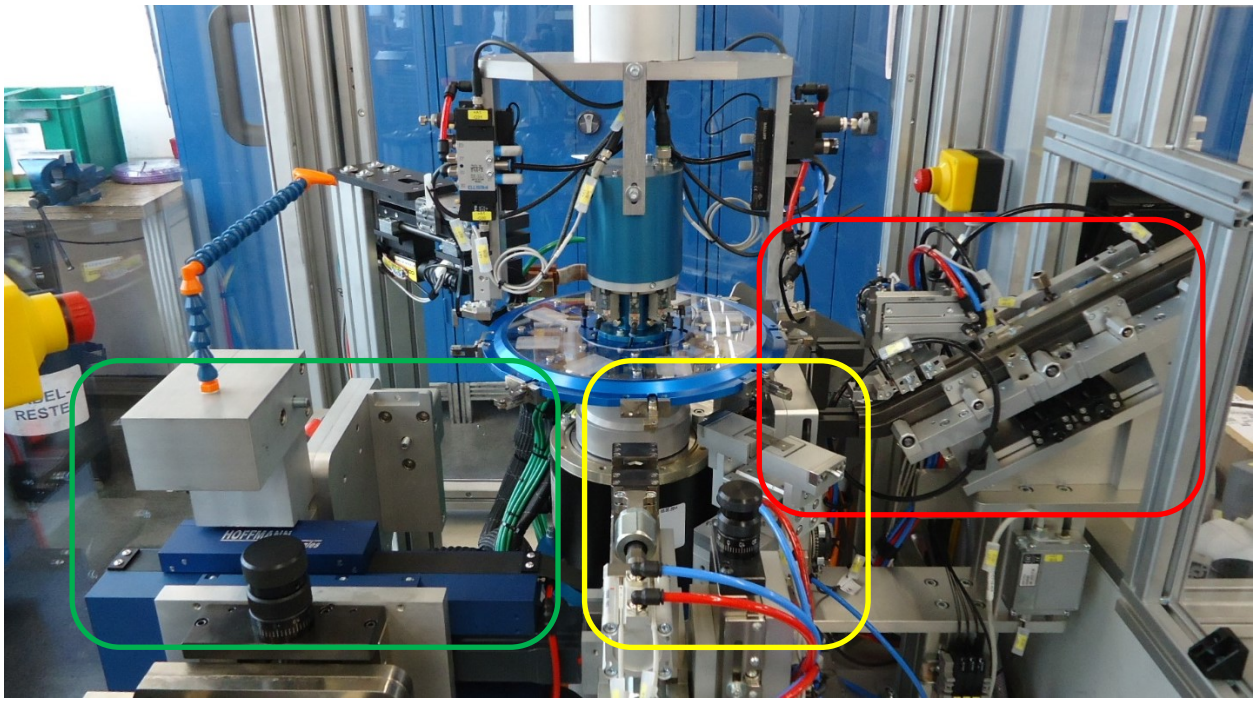


Abb. 1.4: Rundtaktanlage mit vier Stationen, Quelle: Eigene Darstellung.

Auf die mechanische Bearbeitung folgt eine optische Kontrolle der Lauffläche (in Abb. 1.5 rot umrandet). Das Litzenende wird von einer Schweißzange verdichtet um ein Ausfransen zu verhindern und dem Kunden den Einbau der Kohlebürste in seine Produkte zu erleichtern (in Abb. 1.5 gelb umrandet). Anschließend trennt die Auswurfstation Gut- sowie Schlechtteile in verschiedene Behältnisse auf.

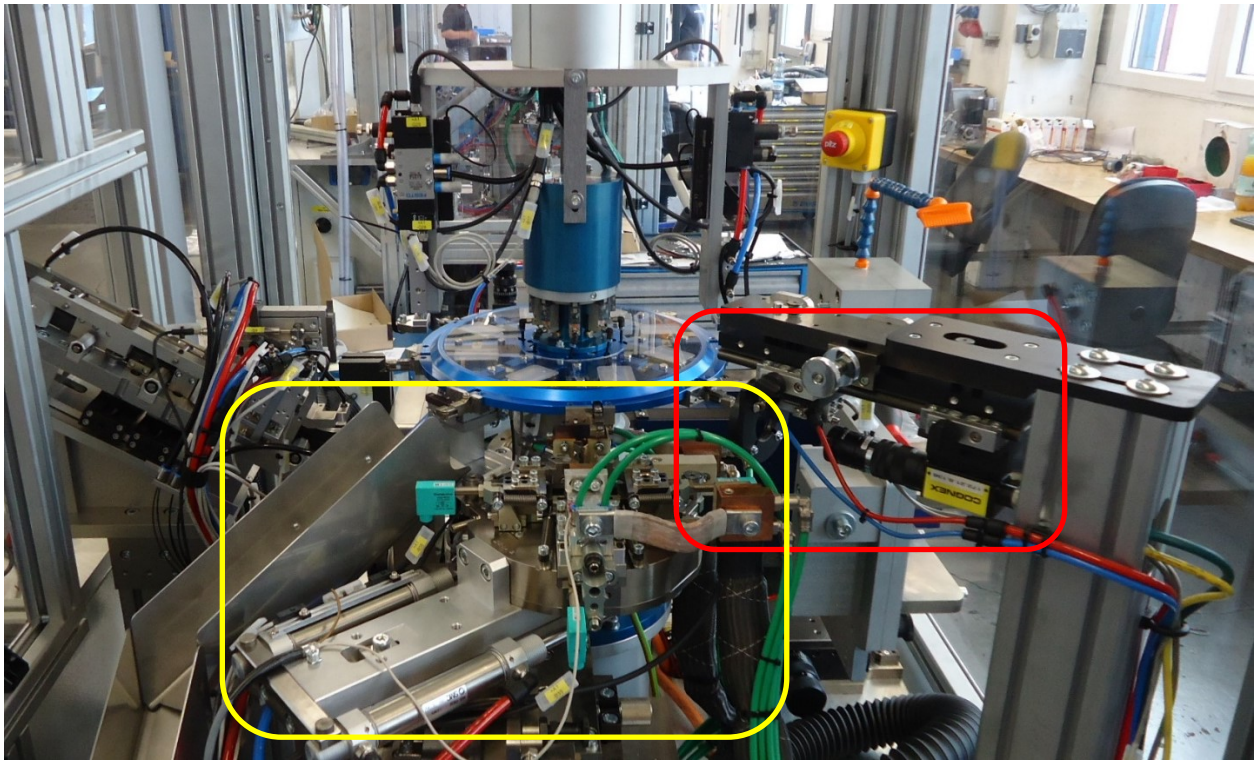


Abb. 1.5: Rundtaktanlage mit drei Stationen, Quelle: Eigene Darstellung.

1.3 Ausgangsituation und Zielsetzung

Die firmeneigene Maschinenbauabteilung von SHCT stellt Rundtaktanlagen für den eigenen Standort aber auch für die weltweiten Niederlassungen der Schunk-Gruppe her, z. B. Deutschland, Mexiko, Brasilien und China. Die Abteilung ist zuständig für die mechanische, elektrische und softwaretechnische Planung sowie Umsetzung. Ziel dieser Masterarbeit ist es, den Ablauf der Programmerstellung zu erleichtern bzw. zu verbessern.

Bei einer neugebauten RTA kommt eine Speicherprogrammierbare Steuerung (SPS) der Firma Beckhoff zum Einsatz. Obwohl bereits einzelne Komponenten, beispielsweise ein Pneumatikzylinder, und ganze Stationen, wie die Schweißstation, programmtechnisch standardisiert sind, ist es üblich bei Neubauten bestehende SPS-Projekte zu kopieren und sie entsprechend abzuändern. Dieser Vorgang birgt die Gefahr, dass notwendige Änderungen übersehen werden und es bei der Inbetriebnahme der Maschine zur vermehrten Fehlersuche kommt.

Um den Produktionsprozess von SPS-Applikationen für Rundtaktanlagen zu optimieren, ist es notwendig das Programm aus standardisierten Bestandteilen und Vorlagen, auch Templates genannt, zu generieren. Aus diesem Grund soll eine Anwendung entwickelt werden, welche es dem Softwareingenieur ermöglicht, die gewünschte RTA in wenigen Schritten zu konfigurieren und automatisch das erforderliche SPS-Projekt für eine Beckhoff-Steuerung zu erstellen. Dadurch steht dem Programmierer eine Basis zur Verfügung, die er je nach Anforderung an die Anlage erweitern kann. Die Applikation hat ein generisches Verhalten zum Ziel um sich auf ändernde Standards im Maschinenbau zu adaptieren können. Aufgrund der automatisierten Herstellung sind geringere Entwicklungszeiten und eine Verminderung von Fehlern, welche bisher durch das Kopieren bzw. Überarbeiten bestehender SPS-Anwendungen entstanden sind, zu erwarten. Das nachstehende Kapitel widmet sich ausführlich den Vorteilen der Codegenerierung und dem Generator selbst.

2 GRUNDLAGEN DER CODEGENERIERUNG

2.1 Vorteile der Codegenerierung

Entwicklungsingenieure stehen vor der Herausforderung, dass an Produkte immer höhere Anforderungen gestellt werden. Zur gleichen Zeit verkürzt sich auf Grund des globalen Wettbewerbs und dem Aufscheinen neuer Technologien der Produktlebenszyklus fortwährend. All diese Faktoren steigern die Komplexität von Projekten und erhöhen gleichzeitig den Druck, den Entwicklungsaufwand möglichst gering zu halten. Im Softwarebereich bietet Codegenerierung aus den von Herrington angeführten Gründen eine Entlastung im Entwicklungsprozess:[2]

- Qualität: Handgeschriebener Code ist von inkonsistenter Qualität, da er meist während der Verwendung von Ingenieuren auf Grund von neuen oder besseren Problemlösungsansätzen überarbeitet wird. Templatebasierte Codegenerierung liefert eine konstante Basis, weil ein Generator jede Änderung der Vorlage sofort übernimmt und damit die Qualität bei allen Projekten gleich hoch bleibt.
- Architektur: Ein Codegenerator ermöglicht nicht nur gleichbleibende Qualität, sondern veranlasst Mitarbeiter durch sein Generat innerhalb der vorgegebenen Softwarearchitektur zu arbeiten.
- Zentrale Wissensbasis: Wie bereits in den vorherigen Punkten erwähnt eignet sich ein Codegenerator als Sammelpunkt für Entscheidungen über die Softwarequalität und Softwarearchitektur. Jede Verbesserung führt zu einer Änderung an einer zentralen Stelle, dem Generator selbst bzw. seinen Templates oder Konfigurationsdateien.
- Zeitersparnis: Neben einer konstanten Grundlage für Projekte ist die Zeitersparnis ein weiterer Vorteil der Codegenerierung. Der Generator übernimmt Routinearbeiten und erzeugt ein Basisprojekt, welches den Entwicklungsingenieuren ermöglicht, ihre Zeit auf neue bzw. komplexe Problemstellungen zu konzentrieren.
- Steigerung der Motivation: Die mit der Codegenerierung einhergehenden Fokussierung der Angestellten auf herausfordernde Aufgaben steigert die Motivation. Besonders bei Projekten, die aus einer Vielzahl sich wiederholenden Programmteilen bestehen, sinkt die Aufmerksamkeit sowie die Konzentration des Programmierers und das Risiko eines Fehlers steigt.

2.2 Formen von Codegeneratoren

Codegeneratoren unterteilen sich in zwei Hauptgruppen: dem aktiven und dem passiven. Eine passive Form generiert einmalig Programmcode, welchen der Benutzer nach Belieben ändern kann. Im Gegensatz dazu ist es dem aktiven Generator möglich, mehrmals mit ein und demselben Generat zu arbeiten. Änderungen am Ergebnis erfolgen durch das Einstellen von Parametern des Codegenerators und führen nach erneuter Generierung zu einem aktualisierten Resultat. Diese Methode ist gegenüber der passiven zu bevorzugen, da eine nachträgliche Parametrierung und erneute Erzeugung in der Praxis gängig ist.[3]

[2] Vgl. Herrington (2003), S.15 ff.
[3] Vgl. Herrington (2003), S. 28.

Bei den nachfolgenden Formen handelt es sich ausschließlich um aktive Codegeneratoren. Dabei unterscheidet Herrington auf Grund ihrer Ein- und Ausgaben sechs unterschiedliche Typen:[4]

Code Munger: Munging ist ein Begriff aus der Informatik und steht für die dauerhafte Umwandlung einer Datei oder Ähnlichem. Der Generator durchsucht seine Quelle nach wichtigen Eigenschaften und nutzt diese, um eine oder mehrere Dateien verschiedener Formate zu erstellen (siehe Abb. 2.1). Bei diesem Vorgang kommen häufig reguläre Ausdrücke, Parser und interne oder externe Templates zum Einsatz (eine Erläuterung der genannten Begrifflichkeiten erfolgt in Kapitel 2.3).

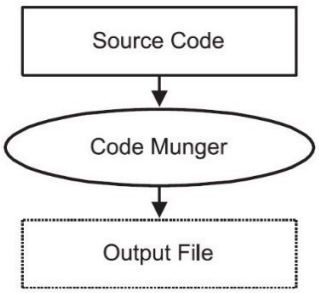


Abb. 2.1: Arbeitsschritte Code Munger, Quelle: Herrington (2003), S. 29.

Inline-Code Expander: Ein Inline-Code Expander hält im zugeführten Quellcode nach speziellen Auszeichnungen (markup) Ausschau, um sie durch ausführbaren Code zu ersetzen (siehe Abb. 2.2). Das Ergebnis ist ein einsatzfähiges Programm.

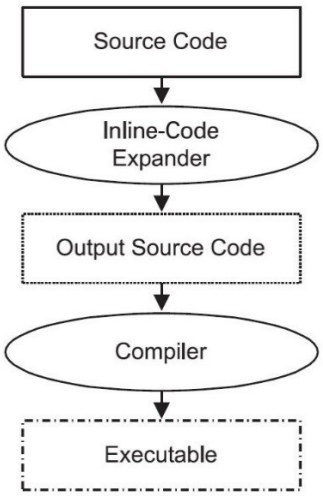


Abb. 2.2: Arbeitsschritte Inline-Code Expander, Quelle: Herrington (2003), S. 30.

Mixed-Code Generator: Ähnlich dem Inline-Code Expander verwendet der Mixed-Code Generator besondere Kommentarbereiche im Sourcecode um sie mit benutzerdefinierten Inhalten zu füllen. Im Gegensatz zum Inline-Code Expander verarbeitet dieser Typ ein und dieselbe Datei zu einer lauffähigen Anwendung (siehe Abb. 2.3).

[4] Vgl. Herrington (2003), S. 29-34.

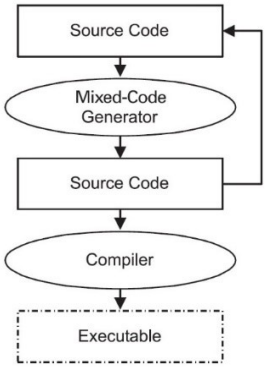


Abb. 2.3: Arbeitsschritte Mixed-Code Generator, Quelle: Herrington (2003), S. 30.

Partial-Class Generator: Er erhält seine Informationen aus einer Konfigurationsdatei und nutzt Templates um eine Bibliothek von Basisklassen zu erzeugen. Kapitel 3.1 widmet sich der Objektorientierung (OO) und erläutert unter anderem den Begriff Klasse. Der Entwicklungsingenieur vervollständigt das Generat durch abgeleitete (derived) Klassen zu einem Programm (siehe Abb. 2.4). Besitzt die Software eine Drei-Schicht-Architektur (Three Tier Architecture), so eignet sich ein Partial-Class Generator als Ausganspunkt zur Erstellung einer Ebene (siehe Abb. 2.4).

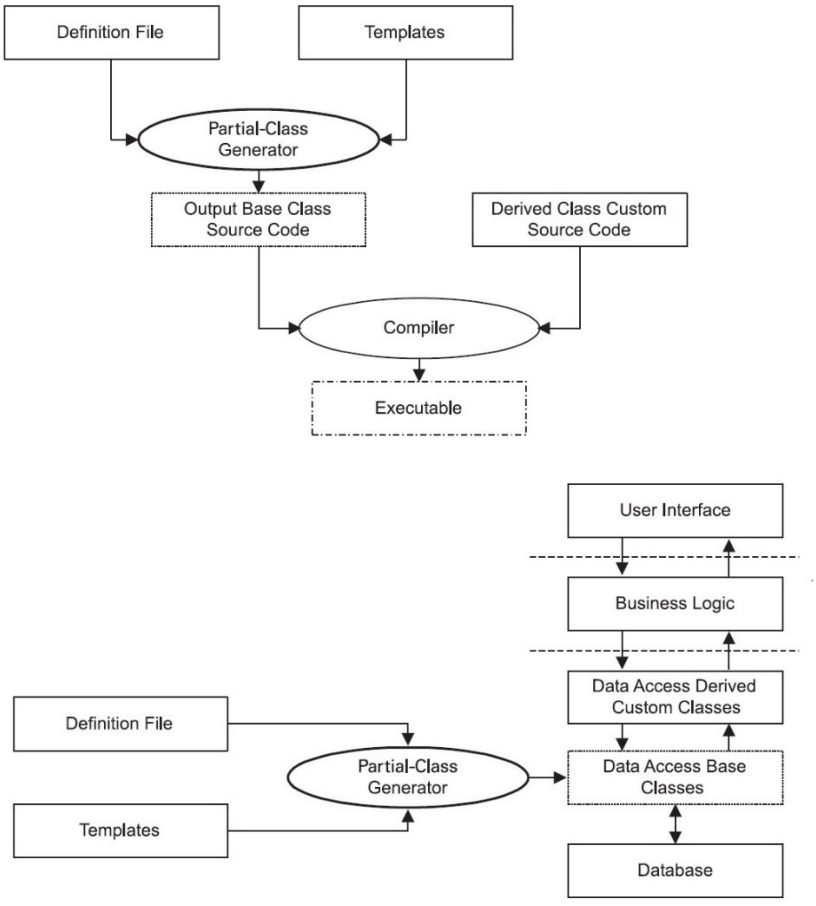


Abb. 2.4: Arbeitsschritte Partial-Class Generator (oben) sowie Einsatz in Drei-Schicht-Architektur (unten), Quelle: Herrington (2003), S. 31 f (leicht modifiziert).

Tier Generator: Ist die Weiterführung des Partial-Class Generators, da er im Stande ist eine komplette Schicht einer Architektur ohne manueller Nacharbeit zu generieren (siehe Abb. 2.5). Der geringere Aufwand für den Anwender spricht für den Tier Generator. Jedoch lässt sich ein Partial-Class Generator einfacher sowie schneller implementieren und ist universell einsetzbar, weil er nicht speziell für einen Verwendungszweck zugeschnitten ist.

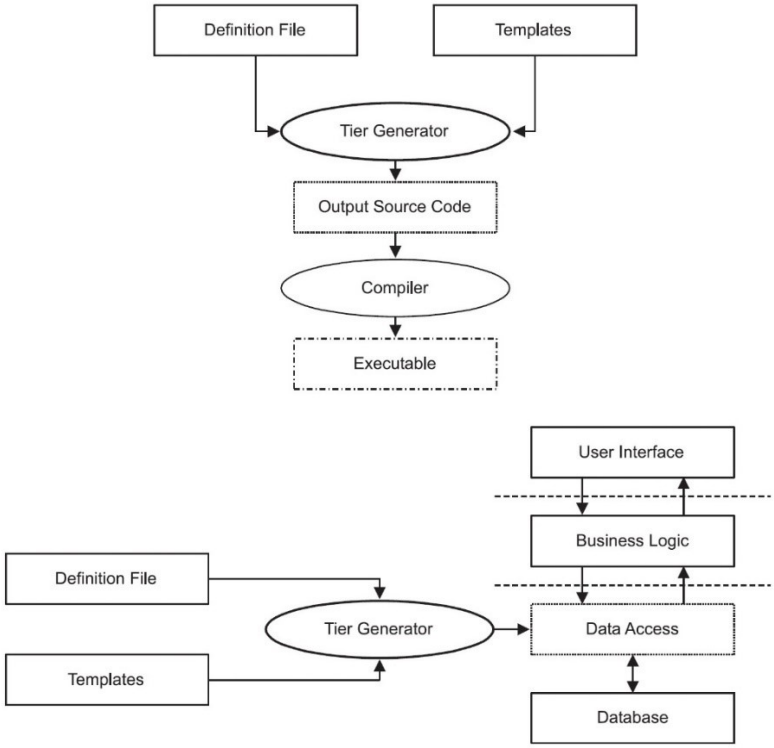


Abb. 2.5: Arbeitsschritte Tier Generator (oben) sowie Einsatz in Drei-Schicht-Architektur (unten), Quelle: Herrington (2003), S. 32 f (leicht modifiziert).

Domänenspezifische Sprache: Dabei handelt es sich um eine Sprache, welche speziell für eine Anwendung oder zur Handhabung einer Problemstellung entwickelt wird. Ein Beispiel hierfür ist das Softwarepaket Mathematica (basierend auf der Programmiersprache Wolfram Language), das es dem Benutzer unter anderem ermöglicht auf einfache Weise Matrixberechnungen durchzuführen, ähnlich der Software Matrix Laboratory (MATLAB) von MathWorks. Mit traditionalen Sprachen wie C, C++ oder Java lassen sich solche Rechenvorgänge nur mit hohem Aufwand realisieren. Ein weiteres Beispiel ist die Sprache HALCON der Firma MVTec, welche einen Standard in der industriellen Bildverarbeitung bildet und wie viele andere domänenspezifischen Sprachen eine dazugehörige Entwicklungsumgebung zur Verfügung stellt (in diesem Fall HDevelop).[5] Die bisher vorgestellten Codegeneratoren eigenen sich zur Erzeugung großer Mengen an ausführbarem Code anhand von Beschreibungen. Eine domänenspezifische Sprache führt dieses Konzept weiter und steht am Ende des Spektrums der Codegenerierung. Neben dem Vorteil, ein auf das Problem zugeschnittenes Werkzeug zu sein, hat die Erstellung einer eigenen Sprache den Nachteil vom Benutzer neu erlernt werden zu müssen und der Erfinder steht vor einem hohen Entwicklungs- sowie Wartungsaufwand.

[5] Vgl. MVTec Software GmbH (o.J.), Online-Quelle [1.12.2017].

2.3 Aufbau und Werkzeuge eines Codegenerators

Eine Software soll so im Idealfall aufgebaut sein, dass sie aus unabhängigen Komponenten besteht und diese nach Belieben erweitert oder ausgetauscht werden können. Diese Anforderungen treffen ebenfalls auf einen Codegenerator zu. Er muss ein generisches Verhalten aufweisen um nicht bei Änderungen der Inhalte das Programm vollständig überarbeiten zu müssen und den Wartungsaufwand möglichst gering zu halten.

Eine entscheidende Rolle hierfür spielt die Architektur der Software, denn sie beschreibt die Struktur eines Systems und beeinflusst Faktoren wie Leistungsfähigkeit oder Verfügbarkeit. Über die Jahre haben sich verschiedene Architekturmuster entwickelt z. B. Model-View-Controller (MVC), Schichten-, Repository- oder Client-Server-Architektur. Diese Modelle enthalten Informationen über deren Aufbau und Eigenschaften sowie Vor- bzw. Nachteile.[6]

Um die eingangs angeführten Forderungen zu erfüllen, bietet sich eine Schichtarchitektur für den Codegenerator an. Sie unterstützt die Erreichung von Trennung und Unabhängigkeit durch eine Aufteilung der Systemfunktionen in unterschiedliche Ebenen (siehe Abb. 2.6). Jede Schicht ist in sich gekapselt und kommuniziert über definierte Schnittstellen mit der darüber- bzw. darunterliegenden Ebene. Dieses Modell hat den Vorteil, dass sich Komponenten einfach ändern oder austauschen lassen solange die Schnittstellen gleich bleiben. Der Nachteil ist, dass sich die Schichten in der praktischen Umsetzung oft nicht sauber trennen lassen und sie über mehrere Ebenen hinweg interagieren müssen.[7]

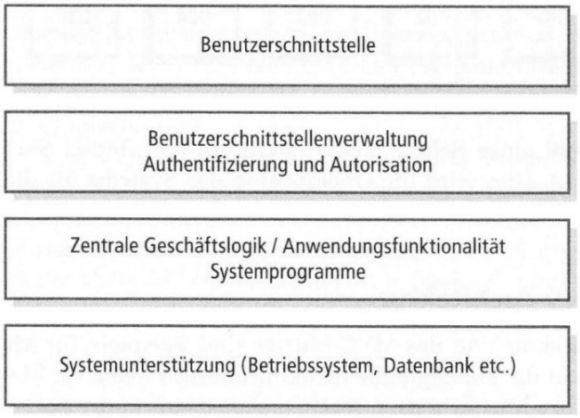


Abb. 2.6: Eine allgemeine Schichtenarchitektur mit vier Ebenen, Quelle: Sommerville (2012), S. 195.

Einen weiteren Ansatz bildet die Model Driven Architecture (MDA), die im Jahr 2001 als Standard von der Object Management Group (OMG) veröffentlicht wurde. Die MDA hat zum Ziel, die Schichten einer Software und deren Entwicklungsprozess als UML-Modelle (Unified Modeling Language) abzubilden. Als Erstes entsteht ein plattformunabhängiges Modell (Platform Independent Model, PIM). Das Modell für die Zielplattform (Platform Specific Model, PSM) lässt sich aus vorhergehenden Modellen ableiten (Kapitel 4.4 widmet sich der modellbasierten Entwicklung im Detail).[8]

[6] Vgl. Sommerville (2012), S. 211.
[7] Vgl. Sommerville (2012), S.194 f.
[8] Vgl. ITWissen (o.J.), Online-Quelle [1.12.2017].

Neben der Architektur sind die Werkzeuge eines Codegenerators entscheidend um ein möglichst flexibles Verhalten zu erreichen. Wie in Kapitel 2.2 erläutert, zählen dazu Templates, reguläre Ausdrücke sowie Parser.

Codegenerierung beinhaltet immer das Erstellen von einer oder mehreren strukturierten Textdateien. Vorlagen (auch Templates genannt) helfen die Formatierung des Codes von der zu verarbeitenden Logik zu trennen.[9]

Hier soll ein simples Beispiel die Funktionsweise von Templates veranschaulichen. Die Entwicklungsumgebung Visual Studio (VS) von Microsoft bietet mit T4-Textvorlagen ein Werkzeug zur Generierung von Textdateien. Das Template besteht aus Textblöcken z. B. den HTML-Tags (Hypertext Markup Language) *<html>* und *<body>* sowie einer Logik, welche in der Programmiersprache C# verfasst ist, hier *DateTime.Now.ToString(…)*.[10] Der Codeausschnitt dient als Platzhalter für das aktuelle Datum mit Uhrzeit und wird nach dem Kompilieren ersetzt. Ein Webbrowser ist im Stande die generierte HTML-Datei darzustellen (siehe Abb. 2.7).

```
<html>
    <body>
        Aktuelles Datum mit Uhrzeit: <#= DateTime.Now.ToString("dd/MM/yyyy HH:mm:ss") #>
    </body>
</html>
```

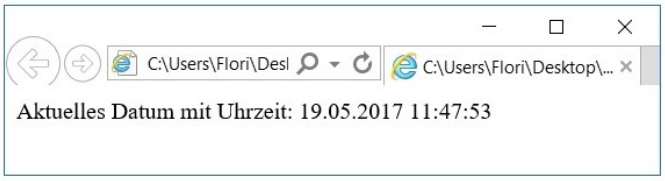


Abb. 2.7: T4-Textvorlage vor dem Kompilieren (oben) und die Darstellung im Webbrowser (unten), Quelle: Eigene Darstellung.

Für einen Codegenerator sind ebenfalls reguläre Ausdrücke von Bedeutung. Sie bieten eine effiziente und flexible Vorgehensweise um Textpassagen zu verarbeiten. Die Haupteinsatzgebiete sind:[11]

- Texte mit einem vorgegebenen Muster abgleichen, siehe nachfolgendes Beispiel.
- Eingaben überprüfen z. B. Passwörter.
- Texte in Bestandteile zerlegen und in strukturierte Form bringen, beispielsweise beim Einlesen einer Konfigurationsdatei.
- Textausschnitte identifizieren und ersetzen.

All diese Funktionen erweisen sich für einen Generator als nützlich, da sie es erlauben, ein Dokument auf simple Weise nach Informationen zu durchsuchen sowie ein bestimmtes Format vorzugeben. Das folgende Beispiel zeigt, wie reguläre Ausdrücke verwendet werden können, um Eingaben mit einem vorgegebenen Muster zu vergleichen. Reguläre Ausdrücke sind Bestandteil vieler Programmiersprachen, hier kommt C# zum Einsatz.

[9] Vgl. Herrington (2003), S. 38.
[10] Vgl. Microsoft (2016), Online-Quelle [1.12.2017].
[11] Vgl. Walter/Thomas (2013), S. 259.

Es erfolgt eine Überprüfung einer Verzeichniseingabe, welche dem Format *Verzeichnis/xyz.test* entsprechen soll. *([a-z]+)* bedeutet, dass der Dateiname ausschließlich aus Kleinbuchstaben bestehen darf und mindestens einen Buchstaben enthalten muss, ansonsten ist der Ausdruck ungültig. Nach der Kontrolle der drei Pfadangaben erscheint ein Fenster mit den Ergebnissen (siehe Abb. 2.8).

```
// Eingaben
string e1 = "Verzeichnis/dateitest";
string e2 = "Verzeichnis/Datei.test";
string e3 = "Verzeichnis/datei.test";

// Mustervorgabe
string v = @"Verzeichnis/([a-z]+)\.test";

// Vergleich mit Muster
Match m1 = Regex.Match(e1, v);
Match m2 = Regex.Match(e2, v);
Match m3 = Regex.Match(e3, v);

// Ausgabe der Ergebnisse
Console.WriteLine(Program.stringRegex(e1, m1, e2, m2, e3, m3));
```

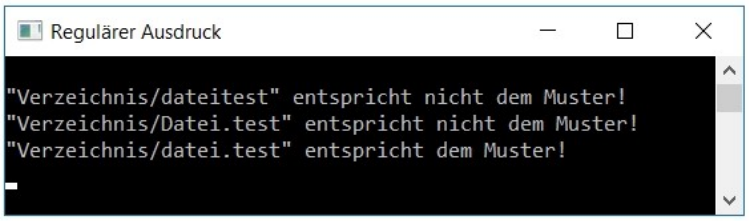


Abb. 2.8: Eingabekontrolle durch regulären Ausdruck (oben) und Anzeige der Ergebnisse (unten), Quelle: Eigene Darstellung.

Den Abschluss bildet der Parser, welcher häufig Teil eines Compilers ist. Ein Compiler übersetzt einen Quelltext in ausführbaren Maschinencode. Erster Schritt ist die lexikalische Analyse, d. h. die Schriftzeichen der Quelle werden Symbolen des Vokabulars der Programmiersprache zugeteilt. Anschließend zerlegt ein Parser die Symbolreihe in weiter verarbeitbare Blöcke oder Einzelzeichen, auch Syntaxanalyse genannt.[12]

Bei einem Parser handelt es sich um ein Programm, welches einen Text in seine Bestandteile auseinandernimmt, um ihn für eine höhere Ebene aufzubereiten. Neben dem Kompilieren finden Parser häufig Anwendung beim Auslesen von Dateien, besonders bei XML-Dateien (Extensible Markup Language). Kapitel 3.4 widmet sich ausführlich der XML und ihren Möglichkeiten in der Codegenerierung. Aufgrund des weitverbreitenden Einsatzes von XML stellen viele Sprachen Parser zur Verfügung, unter anderem C#.

Im nachfolgenden Beispiel werden die Inhalte der Elemente *Parameter1* bis *Parameter3* einer XML-Quelle (siehe Abb. 2.9) ausgelesen und angezeigt. Zunächst lädt die Anwendung das Dokument anhand des Befehls *xml.Load(…)* (siehe Abb. 2.10). Der Parser trennt den Text in seine Bestandteile auf und ermöglicht es somit, die Inhalt der Elemente über eine *for*-Schleife mit der Anweisung *xml.SelectSingleNode(…)* darzustellen (siehe Abb. 2.10).

[12] Vgl. Wirth (2011), S. 1 f.

```
<Konfiguration>
    <Parameter1>4</Parameter1>
    <Parameter2>65.7</Parameter2>
    <Parameter3>C#</Parameter3>
</Konfiguration>
```

Abb. 2.9: XML-Konfigurationsdatei mit drei Parametern, Quelle: Eigene Darstellung.

```
// XML-Datei laden
XmlDocument xml = new XmlDocument();
xml.Load(@"C:\...\test.xml");

// Ausgabe der Parameter 1 bis 3
Console.Title = "XML-Parser";
Console.WriteLine("\nDie Konfigurationsdatei enthält folgende Informationen:");
for (int i = 1; i < 4; i++)
{
    XmlNode node = xml.SelectSingleNode("/Konfiguration/Parameter" + i);
    Console.WriteLine("Parameter{0} = {1}", i, node.InnerText);
}
```

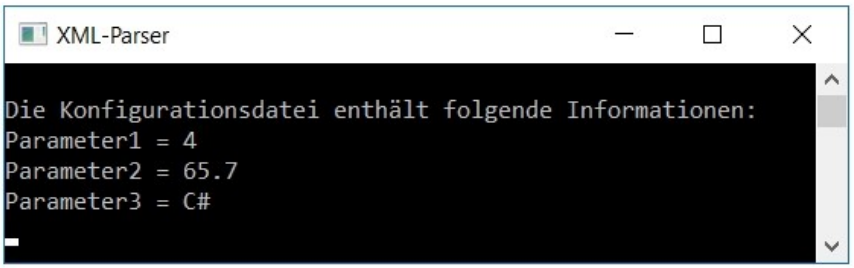


Abb. 2.10: Auslesen von Elementinhalten einer XML-Datei (oben) und deren Ausgabe (unten), Quelle: Eigene Darstellung.

3 CODEGENERIERUNG IN DER INFORMATIK

In der Informatik hat sich aufgrund zunehmender Komplexität der Software die Objektorientierung seit Jahrzehnten als Standard etabliert. Die Objektorientierung umfasst die Analyse, das Design und die Implementierung eines Systems, wobei sich die Unified Modeling Language als Werkzeug zur textuellen und graphischen Darstellung für die einzelnen Entwicklungsphasen herauskristallisiert hat.

Zunächst widmet sich das Kapitel den Grundlagen der OO und der UML, um dann die Möglichkeiten der Codegenerierung unter Einsatz der UML zu zeigen. Abschließend behandelt dieser Abschnitt die Extensible Markup Language, eine Auszeichnungssprache, welche aufgrund ihrer Plattformunabhängigkeit in der Informatik sowie Automatisierungstechnik weit verbreitet ist und Möglichkeiten der Transformation besitzt (ähnlich den Templates aus Kapitel 2.3).

3.1 Konzepte und Techniken der Objektorientierung

Die Objektorientierung hat zum Ziel die reale Welt abzubilden. Dabei sieht sie Personen Gegenstände oder abstrakte Gebilde als Objekte an.

Ein Objekt besitzt gewisse Eigenschaften, auch Attribute genannt. Die Attribute selbst sind konstant, im Gegensatz zu deren Inhalt. Die Summe der Attribute und ihrer Werte bilden den Zustand eines Objekts. Der Zustand kann über Operationen, sogenannten Methoden, verändert werden. Die Menge an Operationen spiegelt das Verhalten des Objekts wider.[13]

Abb. 3.1 zeigt ein Objekt eines Mitarbeiters mit den Attributen *personalnr*, *name* und *gehalt* sowie den Methoden *einstellen*, *erhöheGehalt* und *druckeAusweis*. Die Eigenschaft *gehalt* ist inhärent, jedoch ruft die Operation *erhöheGehalt* eine Änderung ihres Inhalts hervor.

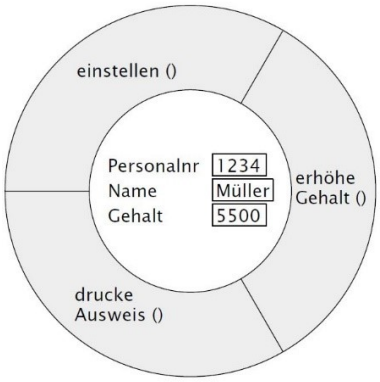


Abb. 3.1: Objekt eines Mitarbeiters, Quelle: Balzert (2011), S. 20.

Daraus geht hervor, dass der Zustand und das Verhalten eines Objekts eine Einheit bilden. Das Objekt kapselt die Informationen in sich und erlaubt einen Zugriff von außen nur über die vorgegebenen Schnittstellen, den Operationen. Man spricht vom Geheimnisprinzip. Abb. 3.2 verdeutlicht dieses Prinzip und die Kommunikation zwischen Objekten. Ein Objekt erhält eine Botschaft eines anderen Objekts durch den Aufruf seiner Methoden mit den entsprechenden Parametern und gibt gegebenenfalls Daten zurück.[14]

[13] Vgl. Balzert (2011), S. 20.
[14] Vgl. Balzert (2011), S. 21.

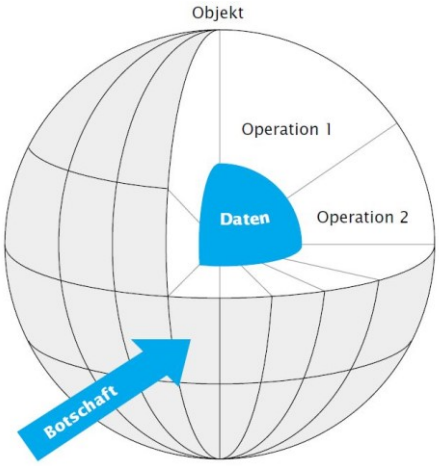


Abb. 3.2: Realisierung des Geheimnisprinzips, Quelle: Balzert (2011), S. .20

Neben dem Zustand und dem Verhalten besitzen Objekte eine Identität, die sie eindeutig ausweist, selbst wenn ein anderes Objekt dieselben Attribute und Methoden aufweist. Diese Beschaffenheit spielt für die Klassifizierung eine entscheidende Rolle.

Klassen bilden den Bauplan eines Objekts und beinhalten die Eigenschaften sowie Operationen, welche eine Gruppe von Objekten vereinen. Klassen lassen sich hierarchisch ordnen und darunterliegende Klassen erben automatisch alle Attribute sowie Methoden ihres Vorgängers. Beispielsweise ist es möglich die Mitarbeiter einer Universität als Personen mit den gemeinsamen Merkmalen *name* und *geburtsdatum* zusammenzufassen (siehe Abb. 3.3). Von der abstrakten Klasse *Person* leiten sich die Klassen *Student* und *Angestellter* ab. Sie besitzen die Inhalte der Elternklasse und erweitern diese um die Eigenschaften *matrikelnr* oder *einstellungsdatum*. Die Vererbung setzt sich über eine weitere Ebene fort, wie die Klassen *wiss. Angestellte* und *Verwaltungsangestellte* zeigen. Von all den abgebildeten Klassen lässt sich eine unbegrenzte Anzahl von Objekten mit unterschiedlichen Identitäten erstellen.[15]

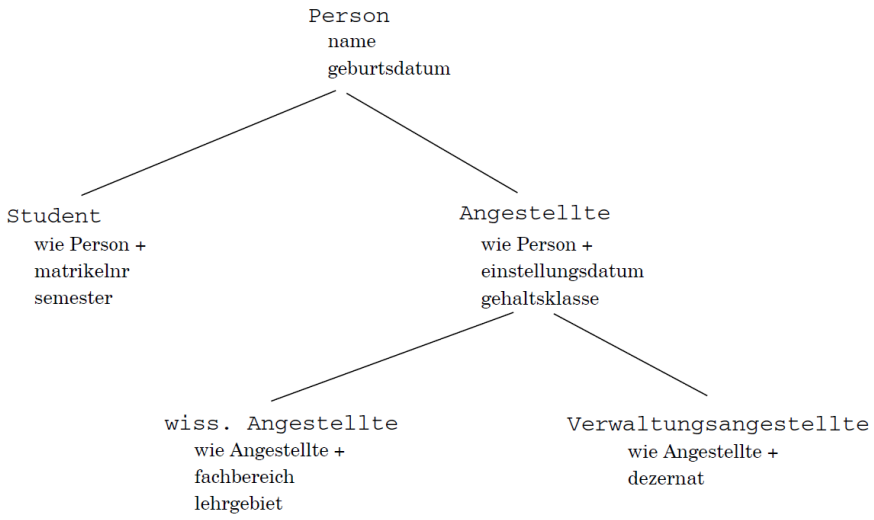


Abb. 3.3: Klassifikation der Mitarbeiter einer Universität, Quelle: Poetzsch-Heffter (2009), S. 19.

[15] Vgl. Poetzsch-Heffter (2009), S. 18 f.

3.2 Unified Modeling Language

Die Unified Modeling Language wurde in den neunziger Jahren des 20. Jahrhunderts entwickelt um einen Standard für die Erstellung objektorientierte Software zu schaffen. Die Object Management Group, ein Zusammenschluss von mehr als 800 Mitgliedern (z. B. Microsoft, Hewlett-Packard oder IBM), wahrt den Standard. Die UML umfasst Notationselemente, welche Analyse, Design und Architektur unterstützen. Dabei ist ihr Einsatz nicht auf die Softwareentwicklung beschränkt, sondern dient im Allgemeinen zur Beschreibung bzw. Visualisierung komplexer Systeme.[16]

Aus dem Einsatz der UML ergeben sich folgende Vorteile:[17]

- Eindeutigkeit: Die Notationselemente besitzen eine präzise, von Experten definierte und entwickelte Semantik.
- Verständlichkeit: Die einfach gehaltene grafische Notation visualisiert das System und erleichtert somit das Verständnis. Unterschiedliche Diagrammtypen bringen differenzierte Sichtweisen auf das System zum Vorschein und ermöglichen somit in jeder Entwicklungsphase das Hervorheben bestimmter Aspekte.
- Ausdrucksstärke: Die wichtigsten Details eines Softwaresystems können durch die verfügbaren Notationselemente beschrieben werden.
- Plattform- und Sprachunabhängigkeit: Die Stärken der UML liegen in der objektorientierten Welt, lassen sich aber auch auf die prozedurale anwenden. Sie kann Softwaresysteme für jede Plattform und Programmiersprache modellieren.

Die UML besitzt in ihrer zweiten Version insgesamt 14 Diagrammtypen (siehe Tab. 3.1). Die Strukturdiagramme dienen zur Beschreibung der Statik eines Systems und die Verhaltensdiagramme zur Darstellung der Dynamik, wobei sich vier Typen der Interaktion von Objekten widmen.[18]

Strukturdiagramme	Verhaltensdiagramme	
		Interaktionsdiagramme
Klassendiagramm	Use-Case-Diagramm	Sequenzdiagramm
Paketdiagramm	Aktivitätsdiagramm	Kommunikationsdiagramm
Objektdiagramm	Zustandsautomat	Timingdiagramm
Kompositionsstrukturdiagramm		Interaktionsübersichtsdiagramm
Komponentendiagramm		
Verteilungsdiagramm		
Profildiagramm		

Tab. 3.1: Diagramme der UML 2, Quelle: Rupp/Queins/SOPHISTen (2012), S. 7.

Jedes Diagramm spiegelt unterschiedliche Sichtweisen auf eine Problemstellung wider und eignet sich besonders, um einen Sachverhalt darzustellen. Im Nachfolgenden werden die drei wichtigsten Arten vorgestellt, welche ebenfalls Einsatz in der Unified Modeling Language Programmier-geeignet (UML/P) Einsatz finden (siehe Kapitel 3.3).

[16] Vgl. Rupp/Queins/SOPHISTen (2012), S. 4.
[17] Vgl. Kecher (2005), S. 16.
[18] Vgl. Rupp/Queins/SOPHISTen (2012), S. 7.

Das Klassendiagramm hält den Aufbau einer Klasse fest (siehe Abb. 3.4) und bildet die Hierarchie mehrerer Klassen ab (vgl. Abb. 3.3). Klassen bestehen aus einem Namensfeld, einer Attribut- und Operationsliste. Jedes Attribut wird durch einen Typ spezifiziert. Der Detailgrad der Angaben zu Attributen und Operationen ist von der Unified Modeling Language nicht vorgegeben. Es obliegt dem Systemanalysten, ob er neben dem Namen weitere Information über Attribute und Operationen festlegt. Jede Klasse erfüllt einen bestimmten Zweck im System. Durch eine Kurzbeschreibung für jede Klasse wird der jeweilige Aufgabenbereich deutlich hervorgehoben.[19]

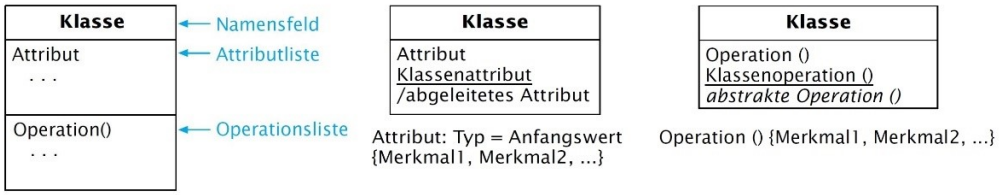


Abb. 3.4: Notation von Klassen, Quelle: Balzert (2011), S. 24 (leicht modifiziert).

Ein Objekt besitzt immer einen Anfangszustand und kann mehrere Zustände bis hin zum Endzustand einnehmen. Der Zustandsautomat stellt diese dar (siehe Abb. 3.5). Zu jedem Augenblick befindet sich das Objekt in nur einem bestimmten Zustand und Ereignisse lösen einen Wechsel (auch Transition genannt) aus. Es ist möglich, dass für die Transition gewisse Bedingungen (*Wächter*) erfüllt sein müssen. Erreicht ein Objekt einen Zustand führt es, falls vorhanden, *entry*-Aktivitäten aus. Analog sind beim Verlassen exit-Aktivitäten anwendbar. Die dritte Variante ist eine do-Aktivität, die das Objekt vom Eintritt bis zum Ende des Zustands verrichtet.[20]

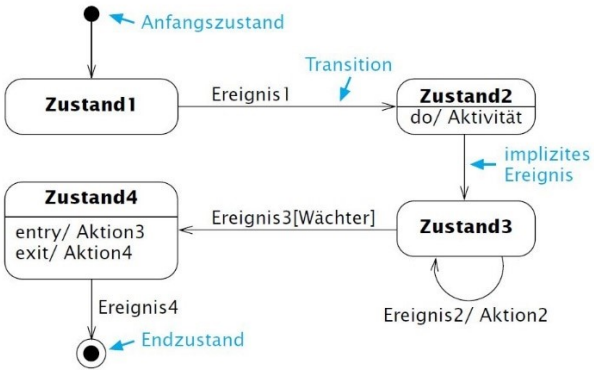


Abb. 3.5: Notation eines Zustandsautomaten, Quelle: Balzert (2011), S. 90.

Die Interaktion zwischen Kommunikationspartnern zeigt das Sequenzdiagramm auf (siehe Abb. 3.6). Es kann sich dabei um Objekte oder Benutzer (*Akteur*) handeln, welche mit dem System interagieren. Jede Botschaft ruft eine Operation des jeweiligen Objekts auf und endet nach dem Abarbeiten der Aktionssequenz in einer Rückantwort. Der Vorteil dieses Diagramms ist, dass es die zeitliche Reihenfolge dokumentiert und alle zur Aufgabenerfüllung notwendigen Operationen auflistet.[21]

[19] Vgl. Balzert (2011), S. 28 ff.
[20] Vgl. Balzert (2011), S. 87 ff.
[21] Vgl. Balzert (2011), S. 81 ff.

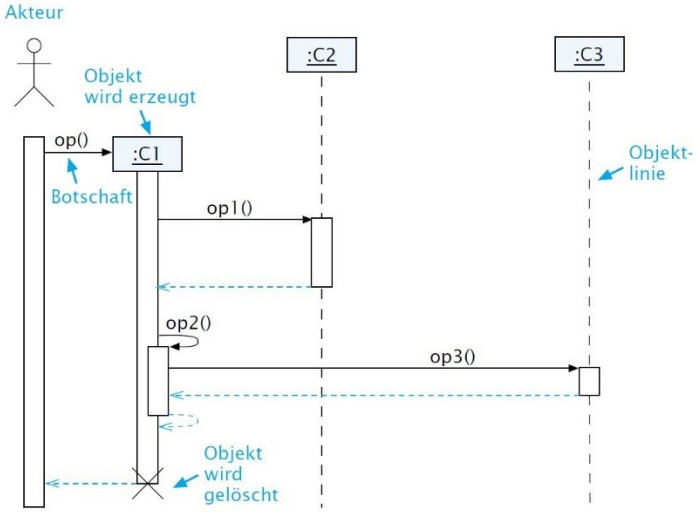


Abb. 3.6: Notation Sequenzdiagramm, Quelle: Balzert (2011), S. 81.

Die UML hat sich unter anderem als Standard in der Softwareindustrie etabliert, weil sie einen Übergang von der objektorientierten Analyse (OOA) über das objektorientierte Design (OOD) zur Umsetzung durch die objektorientierte Programmierung (OOP) ohne Paradigmenwechsel erlaubt (siehe Abb. 3.7). Die Entwurfsphase und die Implementierung sind stark miteinander verbunden. Durch die UML lassen sich Klassen aus dem OOD-Modell direkt in eine Programmiersprache umwandeln.[22]

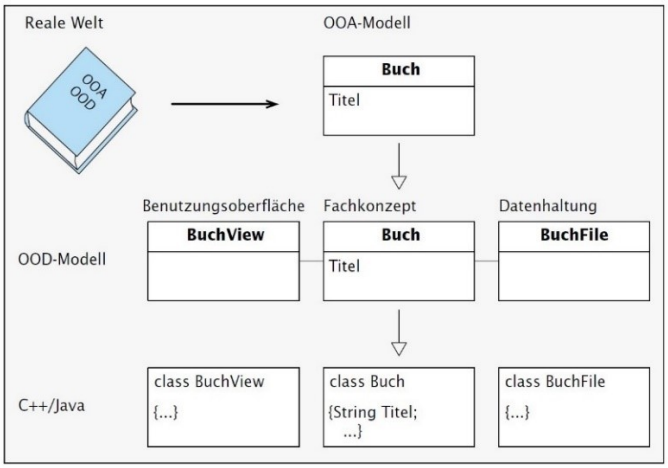


Abb. 3.7: Drei-Schicht-Architektur in den einzelnen Entwicklungsstufen, Quelle: Balzert (2011), S. 13.

Neben der Codegenerierung aus Klassendiagrammen unterstützen gängige Entwicklungsumgebungen wie Microsofts Visual Studio oder Eclipse von Eclipse Foundation weitere Diagrammtypen, unter anderem den Zustandsautomaten. Darüber hinaus sind Applikationen speziell für die Softwareentwicklung mit UML am Markt erhältlich, beispielsweise Modelio SD C++[23] von Modeliosoft und UModel 2017[24] von Altova. Sie bieten neben der Generierung von Programmcode aus UML-Modellen auch eine Synchronisierung von Code und Modell mittels Round-Trip-Engineering an (siehe Abb. 3.8).

[22] Vgl. Balzert (2011), S. 13.
[23] Vgl. Modeliosoft (o.J.), Online-Quelle [1.12.2017].
[24] Vgl. Altova (o.J.), Online-Quelle [1.12.2017].

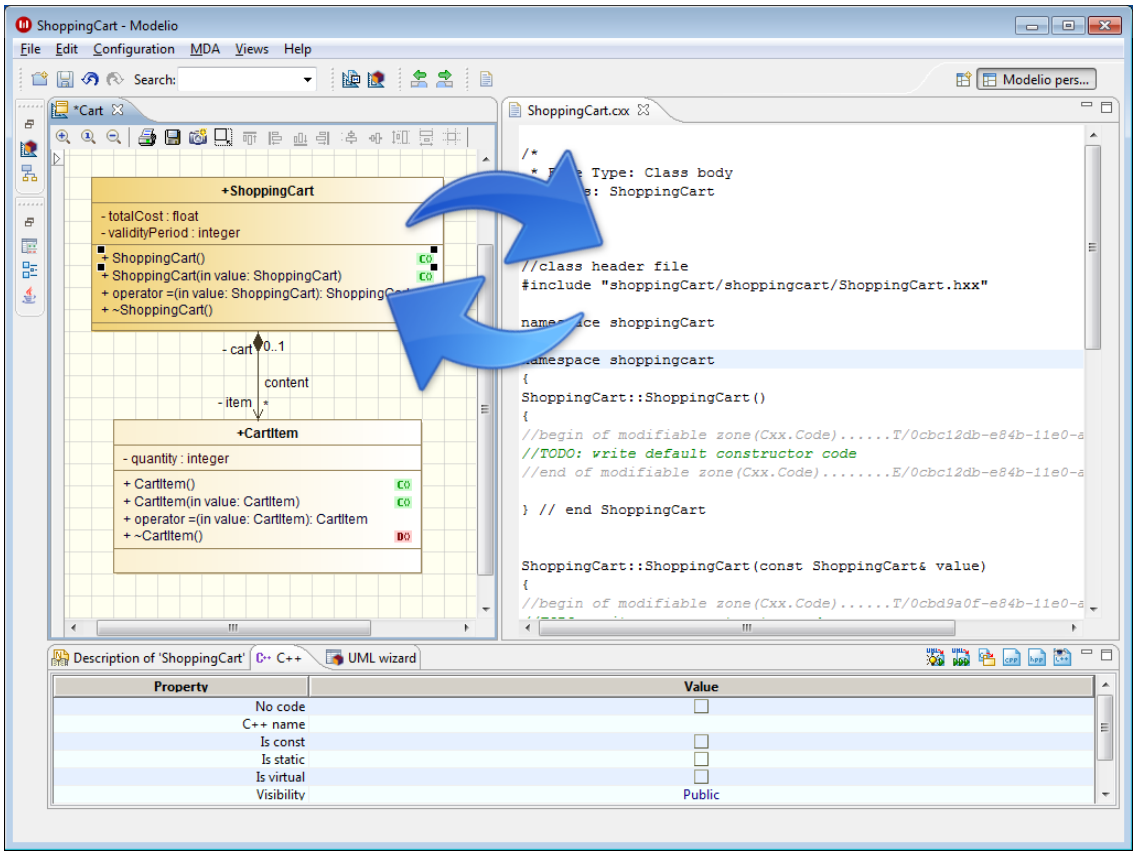


Abb. 3.8: Synchronisierung von Modell und Code in Modelio SD C++, Quelle: Modeliosoft (o.J.), Online-Quelle [1.12.2017].

Eine Weiterentwicklung der UML ist die System Modeling Language (SysML) der OMG, welche als Standardsprache zur Systemmodellierung gilt. Mit der SysML ist es möglich ein System zu beschreiben, das aus Software, Hardware, Personal, Anlagen und den dazugehörigen Prozeduren besteht. Darüber hinaus nutzt SysML XML Metadata Interchange (XMI), um einen standardisierten Datenaustausch zwischen den verschiedenen Ingenieursdisziplinen zu ermöglichen und Drittanbietern eine Schnittstelle für entsprechende Werkzeuge zur Verfügung zu stellen.[25]

Die UML eignet sich zwar zur Codegenerierung, ist aber oftmals zu schwerfällig und zu allgemein. Aus diesem Grund entwickelte sich die Unified Modeling Language Programmier-geeignet, welche im nachstehenden Kapitel behandelt wird.

3.3 Unified Modeling Language Programmier-geeignet

Wie bereits in Kapitel 3.2 erläutert hat sich die UML zu einem Standard in der Softwareindustrie etabliert. Durch die Vielfalt an Diagrammen (siehe Tab. 3.1) erscheint sie nicht für eine agile Entwicklung geeignet. Die UML/P setzt sich zum Ziel die modellbasierten Ansätze der UML mit einer agilen Vorgehensweise zu vereinen und die Ansätze der UML zur Codegenerierung stetig weiterzuführen. Dazu wurde die Anzahl der Diagramme reduziert und die Programmiersprache Java integriert. Durch die Erweiterungen unterstützt die UML/P besonders den Entwurf, die Implementierung sowie Weiterentwicklung von Software und ist als eigenständige Programmiersprache einsetzbar, daher der Suffix Prorammier-geignet. [26]

[25] Vgl. Object Management Group (o.J.), Online-Quelle [13.1.2018].
[26] Vgl. Schindler (2012), S. 19.

Abb. 3.10 zeigt die einzelnen Bestandteile der UML/P:[27]

- Klassendiagramm (Class diagram, CD): Der UML entsprechend dient dieses Diagramm dazu, den Aufbau eines Systems zu beschreiben und stellt das Grundgerüst des Modells dar.
- Objektdiagramm (Object diagram, OD): Bildet Objekte in ihrem aktuellen Zustand ab und eignet sich um Testfälle zu spezifizieren.
- Zustandsautomat (Statechart): Zeigt, wie bei der UML, das Verhalten eines Systems und zusätzlich lassen sich mit der Object Constraint Language (OCL) Einschränkungen bzw. Bedingungen formulieren.
- Sequenzdiagramm (Sequence diagram, SD): Modelliert konkrete Abläufe und definiert in Kombination mit dem Objektdiagramm vollständige Testfälle. Auch hier kommt die OCL zum Einsatz.
- Object Constraint Language: Ist eine Modellierungssprache um Invarianten, also Vor- und Nachbedingungen von Methoden zu definieren. In der UML/P wird der Syntax des OCL-Standards an Java angepasst und bildet die OCL/P.[28] In Abb. 3.9 ist die Überprüfung der Methode *abs()* zu sehen, welche den Absolutwert einer Zahl (*val*) berechnet. Eine Vorbedingung (*pre*) ist nicht notwendig aber die Nachbedingung (*post*) überprüft das Resultat (*result*) auf seine Korrektheit.

```
context int abs(int val)                                    OCL
pre:  true
post: if (val>=0) then result==val else result==-val
```

Abb. 3.9: Überprüfung einer Methode durch OCL, Quelle: Rumpe (2012), S. 82.

- Java: Findet in verschiedenen Diagrammen Einsatz, z. B. um Methodenrümpfe in Klassendiagrammen zu spezifizieren. Java/P ermöglicht durch das Schlüsselwort *ocl* die Integration der OCL/P.

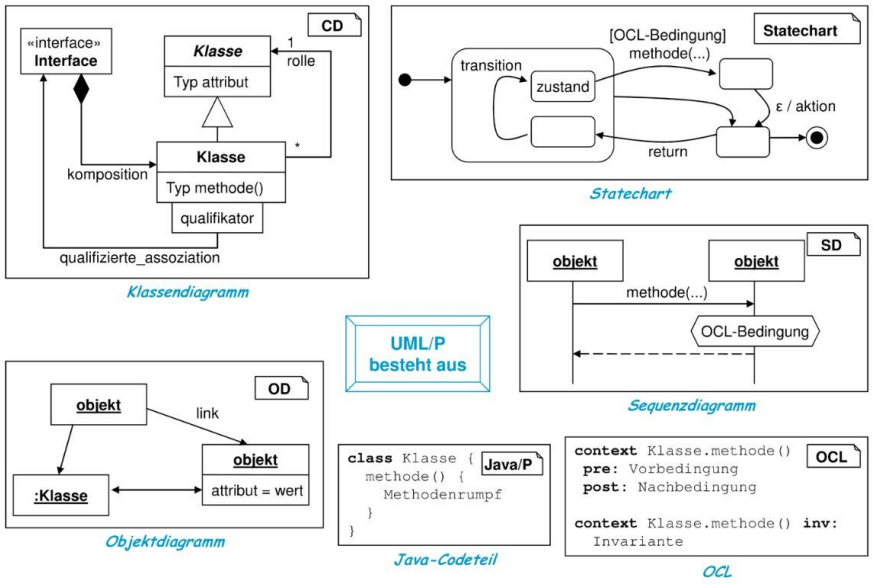


Abb. 3.10: Bestandteile der UML/P, Quelle: Rumpe (2004), S. 35.

[27] Vgl. Schindler (2012), S. 19 f.
[28] Vgl. Rumpe (2012), S. 39 f.

Aufgrund ihrer Bestandteile ermöglicht es die UML/P Modelle zu erstellen und aus ihnen Code für das System zu generieren sowie diesen zu testen (siehe Abb. 3.11). Für den Systemcode kommt das Klassendiagramm sowie Statechart zum Einsatz und für den Testgenerator das Objekt- sowie Sequenzdiagramm. Neben der OCL kann in der UML/P theoretisch auch eine andere Programmiersprache als Java verwendet werden, beispielsweise C++.[29]

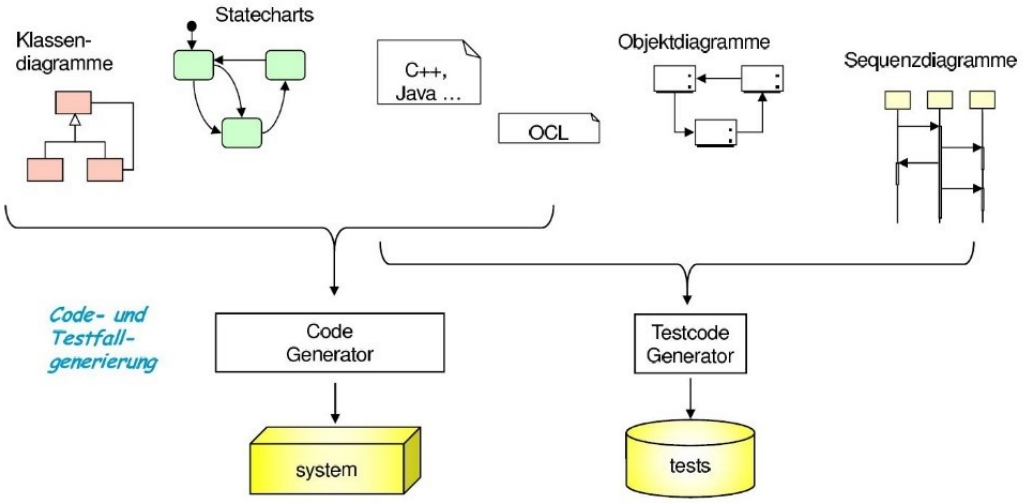


Abb. 3.11: Generierung von Code und Tests aus der UML/P, Quelle: Rumpe (2012), S. 83.

Um die Codegenerierung gegenüber der UML besser zu spezifizieren, ist es notwendig Mechanismen für die Erzeugung plattformabhängiger Codes zu schaffen. Jede Plattform unterliegt einem stetigen Wechsel, der sich durch das Erscheinen neuer Hardware oder Softwarebibliotheken zeigt. Da der Generator nicht im Stande ist vorauszusehen, welche Codestücke einzusetzen sind, ist es notwendig, die Codegenerierung möglichst flexibel zu gestalten. Dabei unterscheidet Rumpe zwei wesentlichen Ansätze:

- Codegenerierung für abstrakte Schnittstellen (siehe Abb. 3.12)
- Parametrierung der Codegenerierung (siehe Abb. 3.13)[30]

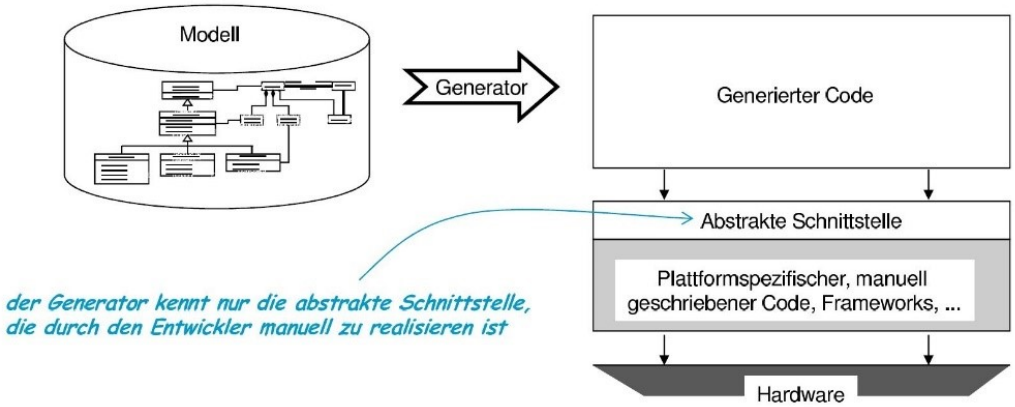


Abb. 3.12: Generierung von Code gegen eine abstrakte Schnittstelle, Quelle: Rumpe (2012), S. 85.

[29] Vgl. Rumpe (2012), S. 82 f.
[30] Vgl. Rumpe (2012), S. 85.

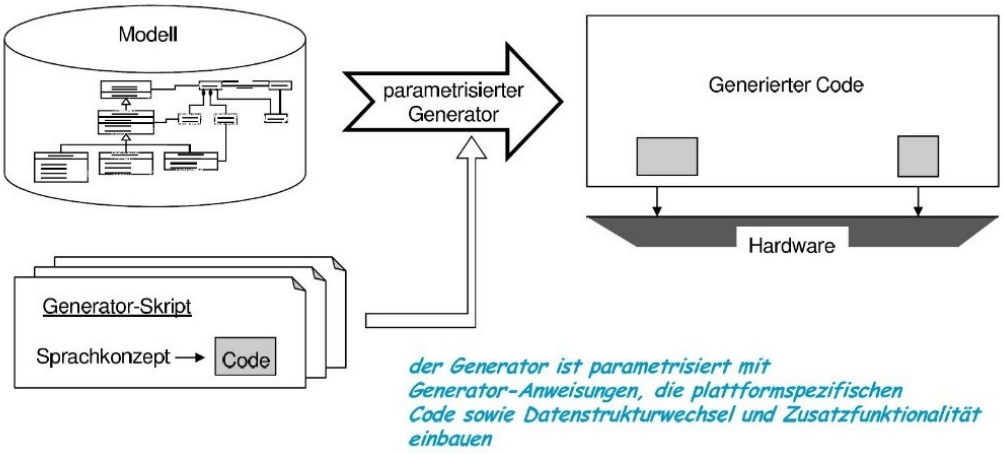


Abb. 3.13: Parametrisierte Codegenerierung, Quelle: Rumpe (2012), S. 86.

Die abstrakte Schnittstelle trennt den generierten Code von der plattformabhängigen Hardware. Der parametrisierte Generator nutzt Skripte und Templates um das Resultat an die gegebenen Bedingungen anzupassen. In der Praxis führt eine Mischform der beiden Typen zum besten Ergebnis. Des Weiteren ist eine Laufzeitumgebung notwendig, um Funktionalitäten zur Verfügung zu stellen, welche von Java nicht abgebildet werden können (z. B. die Behandlung von Mengen und Listen durch die OCL). Abb. 3.14 zeigt somit den prinzipiellen Aufbau des Codegenerators.[31]

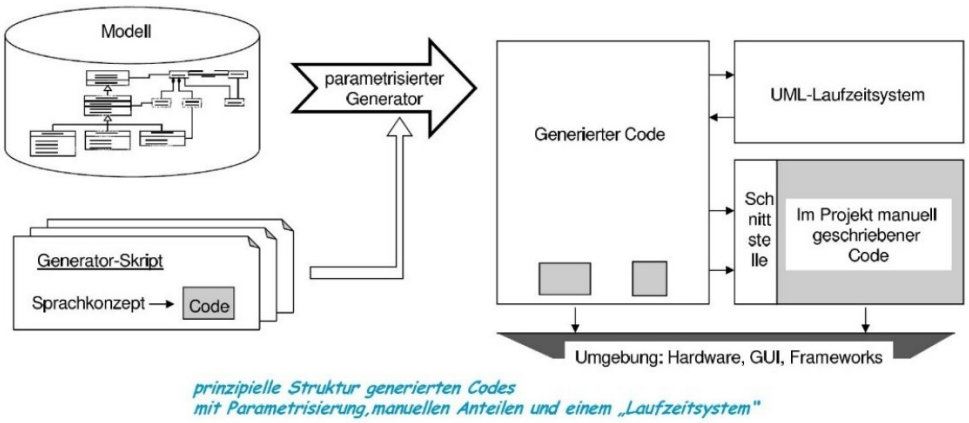


Abb. 3.14: Struktur eines Codegenerators, Quelle: Rumpe (2012), S. 87.

Da die zu erzeugenden Codestrukturen von vielen Faktoren abhängen, ist es meist unerlässlich Bedingungen für die Anwendbarkeit, Generierung oder Analyse festzulegen und zu überprüfen. Skripten und Templates ist es über eine entsprechende Schnittstelle (*PlugIn-API*) möglich, die Transformation hin zum Implementierungscode zu beeinflussen (siehe Abb. 3.15). Für die UML/P kommen verschiedene Sprachen in Frage unter anderem XML/XSLT (Extensible Markup Language/Extensible Stylesheet Language), welche in Kapitel 3.4 behandelt wird. Rumpe entscheidet sich aufgrund der Vielfalt an Bibliotheken und Werkzeuge für Java, damit Skript- und Zielprogrammiersprache ähnlich sind.[32]

[31] Vgl. Rumpe (2012), S. 86 f.
[32] Vgl. Rumpe (2012), S. 96 f.

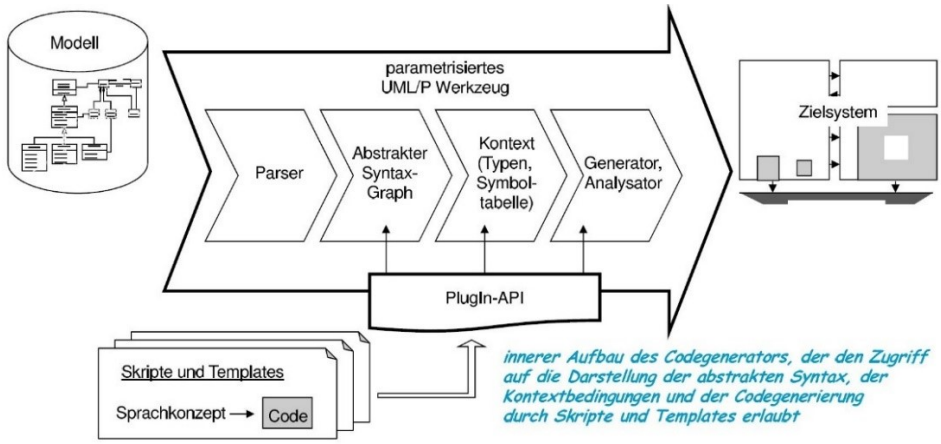


Abb. 3.15: Innere Struktur eines Codegenerators, Quell: Rumpe (2012), S. 96.

Abb. 3.16 stellt ein Beispiel für eine Transformation einer Klasse mit einem Attribut aus dem Klassendiagramme in ausführbaren Java-Code dar. Die Platzhalter sind dabei kursiv gehalten.

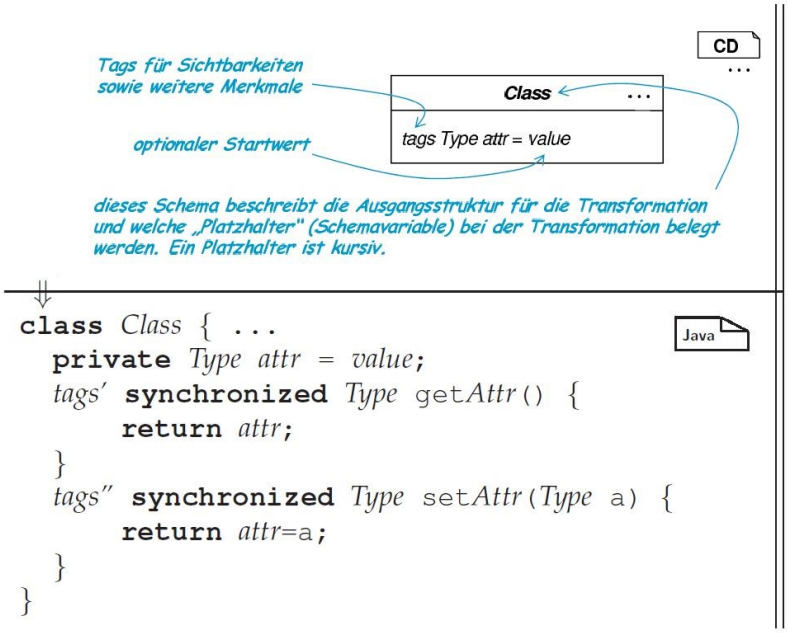


Abb. 3.16: Transformation von Attributen, Quelle: Rumpe (2012), S. 100 (leicht modifiziert).

Abschließend ist auf die Dissertation von Martin Schindler mit dem Titel *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P* zu verweisen. In dieser setzt er das MontiCore Generatorframework und die Java Template Engine FreeMarker ein, um ein Werkzeug zur agilen Entwicklung mit UML/P zu erstellen, das sich erweitern und an Technologien sowie Domänen anpassen lässt.[33]

3.4 Extensible Markup Language

Bereits in Kapitel 2.3 und 3.3 findet XML Einsatz im Codegenerierungsprozess und soll in diesem Abschnitt ausführlich behandelt werden, besonders in der Kombination mit XSLT.

[33] Vgl. Schindler (2012), S. 237 f.

Bei XML handelt sich um eine vom World Wide Web Consortium (W3C) entwickelte und standardisierte Metasprache, welche eine beliebige Auszeichnungssprache definiert. XML ist universell einsetzbar, da sie nur die Struktur und den Aufbau der Daten beschreibt. Der Aufbau eines XML-Dokuments unterliegt gewissen Regeln, um als wohlgeformtes Dokument zu gelten und somit für eine Vielzahl an Werkzeugen verarbeitbar zu sein.[34]

Eine wohlgeformte XML-Datei besteht aus einem Prolog und den Daten selbst, die zumindest ein Element besitzen müssen (siehe Abb. 3.17). Der Prolog enthält Informationen über das Dokument selbst z. B. die XML-Version oder Zeichenkodierung.[35]

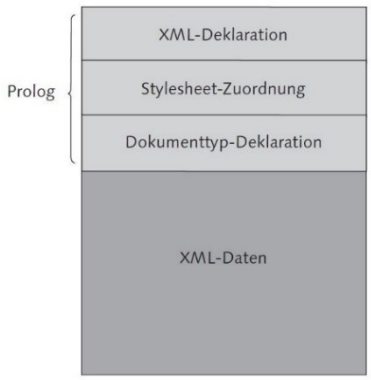


Abb. 3.17: Aufbauschema eines XML-Dokuments, Quelle: Vonhoegen (2015), S. 51 (leicht modifiziert).

Die Daten sind hierarchisch in Form eines Baums angeordnet und bestehen aus Elementen sowie Attributen (siehe Abb. 3.18). Ein Element muss das gesamte Dokument umschließen, das sogenannte Wurzelelement (hier *kontakte*). Innerhalb des Wurzelelements sind der Tiefe der Verschachtelung keine Grenzen gesetzt, entscheidend ist nur die richtige Reihenfolge der Tags.[36]

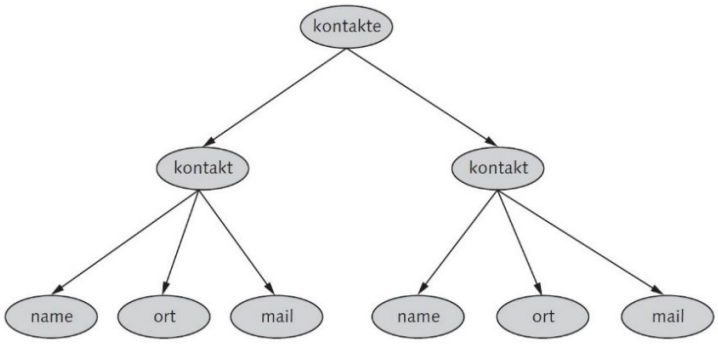


Abb. 3.18: Baumstruktur eines Dokuments, Quelle: Vonhoegen (2015), S. 54.

Jedes Element besitzt einen Start- und End-Tag, welche in spitzen Klammern stehen (siehe Abb. 3.19). Die Tags umschließen den Elementinhalt, der aus einfachem Text besteht. Ein öffnendes Tag muss auch wieder geschlossen werden. Zusätzlich kann ein Tag über sogenannte Attribute verfügen, die weitere Spezifikationen enthalten z. B. *<title language = "de">*.[37]

[34] Vgl. Kersken (2013), S. 821 f.
[35] Vgl. Vonhoegen (2015), S. 50.
[36] Vgl. Vonhoegen (2015), S: 53 f.
[37] Vgl. Vonhoegen (2015), S. 54 f.

Abb. 3.19: Ein Element in XML, Quelle: Vonhoegen (2015), S. 54.

Bei XSLT handelt es sich um eine in XML definierte Sprache: Sie beschreibt die Transformation eines XML-Dokuments in ein XML-, HTML- oder Textdokument. Der XSLT-Stylesheet legt fest, welche Elemente und Attribute des eingehenden Dokuments im ausgehenden abgebildet werden. Um in der Baumstruktur einer XML-Datei zu navigieren und die einzelnen Bestandteile identifizieren zu können, nutzt XSLT die Sprache XML Path Language (XPath). Die Transformation übernimmt eine spezielle Anwendung, der XSLT-Prozessor. Er verarbeitet das XML-Dokument gemäß den Regeln des Stylesheets und erzeugt das gewünschte Ergebnisdokument (siehe Abb. 3.20).[38]

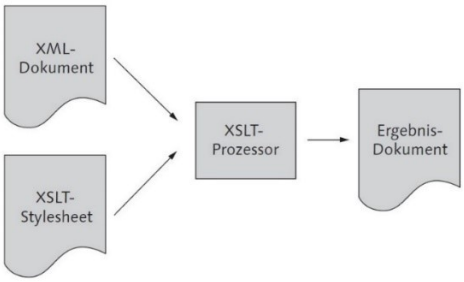


Abb. 3.20: Generelles Schema der Transformation durch XSLT, Quelle: Vonhoegen (2015), S. 257.

Den Abschluss soll ein Beispiel für eine Transformation einer XML- in eine HTML-Datei durch XSLT bilden. In Abb. 3.21 ist eine CD-Sammlung zu sehen. Das Wurzelelement <*catalog*> umschließt mehrere CDs (<*cd*>) mit ihren jeweiligen Informationen (z. B. <*title*> oder <*artist*>).

```
<?xml version="1.0" encoding="UTF-8"?>
<catalog>
  <cd>
    <title>Empire Burlesque</title>
    <artist>Bob Dylan</artist>
    <country>USA</country>
    <company>Columbia</company>
    <price>10.90</price>
    <year>1985</year>
  </cd>
  <cd>
    <title>Hide your heart</title>
    <artist>Bonnie Tyler</artist>
    <country>UK</country>
    <company>CBS Records</company>
    <price>9.90</price>
    <year>1988</year>
  </cd>
  <cd>
    <title>Greatest Hits</title>
    <artist>Dolly Parton</artist>
    <country>USA</country>
    <company>RCA</company>
    <price>9.90</price>
    <year>1982</year>
  </cd>
</catalog>
```

Abb. 3.21: Eine CD-Sammlung als XML-Dokument, Quelle: w3schools (o.J.), Online-Quelle [1.12.2017] (leicht modifiziert).

[38] Vgl. Kersken (2013), S. 850 f.

Der dazugehörige Stylesheet enthält eine Reihe von Templates, welche durch *<xsl:template>* gekennzeichnet sind (siehe Abb. 3.22). Das Attribute *match* beinhaltet jeweils eine XPath-Anweisung und gibt an, auf welches Element des eingehenden XML-Dokuments das Template angewendet werden soll z. B. *"cd"* für *<cd>-Elemente*. Das erste Template fügt dem Wurzelelement die Überschrift *My CD Collection* hinzu. Das zweite wählt aus jeder CD die Kindelemente *<title>* und *<artist>*. Schlussendlich extrahieren Template drei und vier jeweils den aktuellen Elementinhalt (*<xsl:value-of select = ".">*) und fügen ihn hinter dem Text *Title:* bzw. *Artist:* an.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
  <html>
  <body>
  <h2>My CD Collection</h2>
  <xsl:apply-templates/>
  </body>
  </html>
</xsl:template>

<xsl:template match="cd">
  <p>
    <xsl:apply-templates select="title"/>
    <xsl:apply-templates select="artist"/>
  </p>
</xsl:template>

<xsl:template match="title">
  Title: <xsl:value-of select="."/><br/>
</xsl:template>

<xsl:template match="artist">
  Artist: <xsl:value-of select="."/><br/>
</xsl:template>
</xsl:stylesheet>
```

Abb. 3.22: XSLT-Stylesheet mit Templates, Quelle: w3schools (o.J.), Online-Quelle [1.12.2017] (leicht modifiziert).

Das Resultat der Transformation ist eine Auflistung der CD-Sammlung mit einer Überschrift und den Daten zu Titel bzw. Künstler. Das Ergebnisdokument ist in einem Webbrowser darstellbar (siehe Abb. 3.23).

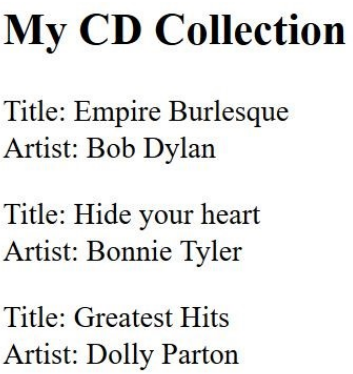


Abb. 3.23: Ergebnis der Transformation in einem Webbrowser, Quelle: w3schools (o.J.), Online-Quelle [1.12.2017] (leicht modifiziert).

4 CODEGENERIERUNG IN DER AUTOMATISIERUNGSTECHNIK

In der Automatisierungstechnik ist der objektorientierte Entwurf von Anwendungen und deren Umsetzung im Gegensatz zur Informatik keine Selbstverständlichkeit und findet erst seit einigen Jahren Anklang.[39] Die Branche zeichnet sich durch ihre interdisziplinäre Arbeitsweise aus. Üblicherweise stellt die mechanische Konstruktion die erste Phase des Entwicklungsprozesses dar und führt über die elektrische Planung zur Softwareentwicklung. Um Personen, welche nicht mit der Steuerungstechnik vertraut sind, bereits früh eine Schnittstelle zu bieten, gibt es Bestrebungen, Anwendungen, die weitverbreitet sind (z. B. Microsoft Excel) oder grafische Modellierung zu nutzen (wie Automaten oder Petri-Netze) um daraus Code zu generieren. Nachdem Programme immer mehr Aufgaben in einer Maschine übernehmen und durch die vierte industrielle Revolution (Industrie 4.0) eine steigende Komplexität zu erwarten ist, wird die durchgängige modellbasierte Entwicklung forciert, um von der altgedienten Vorgehensweise der Projektabwicklung loszukommen.[40] Das Kapitel zeigt Umsetzungen zu allen angesprochenen Themengebiete, widmet sich aber zunächst den Grundlagen der Steuerungstechnik.

4.1 Speicherprogrammierbare Steuerung

Eine SPS zeichnet sich besonders durch ihren langlebigen Einsatz in industriellen Umgebungen aus. Sie besteht aus einer Stromversorgung, einem Steuerungsprozessor sowie digitalen oder analogen Ein- und Ausgangsgruppen. Die einzelnen Module sind meist steckbar ausgeführt und können individuell kombiniert werden. Neben der Hardware-SPS findet die PC-basierte Steuerung zunehmend Verwendung in Automatisierungslösungen. Eine sogenannte Soft-SPS besteht aus einer Applikation, welche auf einem Betriebssystem basiert und das Verhalten einer Hardware-SPS emuliert. Beide Typen nutzen Feldbusse um mit externer Peripherie zu kommunizieren.[41]

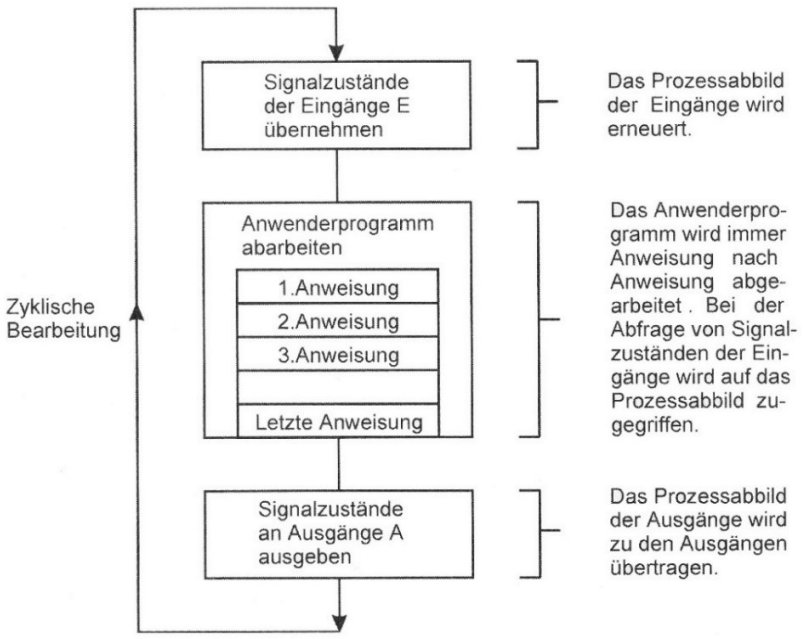


Abb. 4.1: Funktionsmodell einer SPS, Quelle: Wellenreuther/Zastrow (2011), S. 14.

[39] Vgl. SPS Magazin (2016), Online-Quelle [1.12.2017].
[40] Vgl. Hofmann/Menager/Schweig/Mikelsons (2015), S. 252.
[41] Vgl. Wellenreuther/Zastrow (2011), S. 9 f.

Ein weiteres Merkmal der SPS ist ihre zyklische Abarbeitung des Anwenderprogramms (siehe Abb. 4.1). Zunächst liest die Steuerung die Signalzustände der Eingangsbaugruppen ein und bildet daraus das Prozessabbild der Eingänge (PAE). Mit dem PAE durchläuft die Steuerung jede Codezeile und erstellt anschließend das Prozessabbild der Ausgänge (PAA). Das PAA wird an die Ausgänge der Ausgabebaugruppen weitergegeben. Die für einen einmaligen Durchlauf benötigte Zeit nennt man Zykluszeit. Sie muss so gering sein, dass die SPS notwendige Änderungen an den Eingängen erkennt und entsprechende Ausgänge rechtzeitig schaltet. Diese Anforderung ist in der Steuerungstechnik als Echtzeitbedingung bekannt.[42]

Seit dem Jahr 1992 pflegt die PLCopen die standardisierten Programmiersprachen der IEC 61131-3. Ziel der Norm ist es, Anwendern eine Möglichkeit zu geben, herstellerunabhängige SPS-Applikationen zu verfassen und sie zwischen unterschiedlichen Plattformen zu transferieren. Insgesamt sind fünf Sprachen definiert, wobei Steuerungen von Siemens andere Bezeichnungen verwenden (siehe Tab. 4.1) und der Syntax ebenfalls von der IEC 61131-3 abweicht.[43]

Darstellung	**IEC 61131-3**	**Siemens**
grafisch	Kontaktplan (KOP)	Kontaktplan (KOP)
grafisch	Funktionsbausteinsprache (FBS)	Funktionsplan (FUP)
textuell	Anweisungsliste (AWL)	Anweisungsliste (AWL)
textuell	Strukturierter Text (ST)	Structured Control Language (SCL)
grafisch und textuell	Ablaufsprache (AS)	S7-Graph

Tab. 4.1: Vergleich der Benennung der Programmiersprachen, Quelle: Vogel-Heuser/Wannagat (2009), S. 28 (leicht modifiziert).

Die Wurzeln der Steuerungstechnik liegen im Abbilden von Relaissteuerungen. Aus diesem Grund ist die grafische Programmiersprache Kontaktplan entstanden. Die Darstellung ist an die Stromlaufpläne der Elektrotechnik angelehnt. Der Funktionsplan, entwickelt von Siemens, ist in Europa weit verbreitet. Er stellt boolesche Elemente und Funktionen gekapselt in Kästchenform dar. Diese grafische Fachsprache ist in der IEC 61131-3 als Funktionsbausteinsprache enthalten. Die historische Weiterentwicklung führt zu der assemblernahen textuellen Sprache Anweisungsliste. Der Strukturierte Text ergänzt die Norm um eine Hochsprache. Die Ablaufsprache enthält textuelle sowie grafische Elemente und dient zur Darstellung von Vorgängen.[44]

Ein Steuerungsprogramm besteht laut Norm aus einer Kombination der drei Programmorganisations-einheiten (POE): Programm (PROG), Funktionsbaustein (FB) und Funktion (FC). Auch hier weicht Siemens in zwei Punkten von der IEC 61131-3 ab: Organisationsbausteine (OB) ersetzen das Programm und zusätzlich gibt es Datenbausteine (DB), die im Stande sind Daten persistent zu speichern. Die einzelnen POEs unterscheiden sich wie folgt:[45]

[42] Vgl. Wellenreuther/Zastrow (2011), S.14.
[43] Vgl. Vogel-Heuser/Wannagat (2009), S.27.
[44] Vgl. Vogel-Heuser/Wannagat (2009), S. 28.
[45] Vgl. Vogel-Heuser/Wannagat (2009), S. 36.

- Eine Funktion ist die einfachste Form der drei Bausteine. Sie verarbeitet mehrere Eingänge zu einem Ausgang und gibt diesen als Rückgabewert aus. Bei gleichbleibenden Eingängen liefert die Funktion dasselbe Ergebnis, da sie über keine Gedächtnisfunktion verfügt.
- Im Gegensatz zur Funktion besitzt der Funktionsbaustein ein Gedächtnis. Aus mehreren Eingängen werden unter der Berücksichtigung der intern gespeicherten Zustände, ein oder mehrere Ausgänge an den aufrufenden POE zurückgegeben. Das bedeutet, er arbeitet mit einem eigenen Datensatz und gleichbleibende Eingangsparameter können bei einem nochmaligen Durchlauf zu einem neuen Resultat führen. Um FBs mehrmalig in einem Programm zu nutzen, müssen von ihnen Instanzen gebildet werden.
- Der oberste POE-Typ in der Hierarchie ist das Programm. Es besitzt als einziges die Möglichkeit auf die Peripherie zuzugreifen und stellt somit anderen Organisationseinheiten den Zugriff auf Eingangs- und Ausgangsvariablen der SPS zur Verfügung. Meist wird ein Programmbaustein als zentrale Verwaltungsstelle benutzt, um von dort aus die für den Ablauf benötigten POEs aufzurufen.

Allen drei Typen ist derselbe Aufbau gemein. Der Deklarationsteil beinhaltet einerseits lokale Variablen und andererseits jene, die eine Schnittstelle nach außen bilden. Der Anweisungsteil besteht aus dem auszuführenden Programmcode (siehe Abb. 4.2).

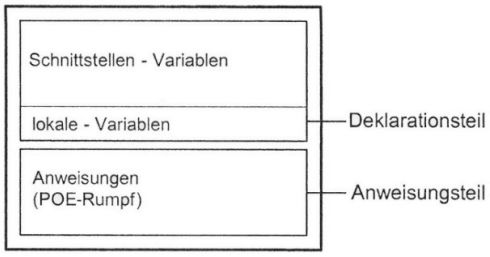


Abb. 4.2: Aufbau einer Programmorganisationseinheit, Quelle: Vogel-Heuser/Wannagat (2009), S. 37 (leicht modifiziert).

Nachdem sich die Konzepte der Objektorientierung in der Informatik seit Jahren etabliert haben (siehe Kapitel 3.1) finden sie durch die objektorientierte Erweiterung der IEC 61131-3 Einzug in die Steuerungstechnik. Die Norm erhält zehn zusätzliche Schlüsselwörter wie *METHOD*, *PROPERTY*, *EXTENDS* oder *INTERFACE* und erhebt den Funktionsbaustein zur Klasse. Dies hat zur Folge, dass Funktionsbausteine voneinander erben können und Methoden sowie Attribute besitzen.[46]

4.2 Excel-Listen

Eine wirtschaftlich rentable Programmierung von Automatisierungslösungen für Großanlagen zeichnet sich durch eine Vielzahl von wiederverwendbaren und adaptierbaren Elementen aus. Es liegt nahe, die einzelnen Bestandteile als Liste zu führen, um aus ihr den entsprechenden Code zu generieren und somit die Entwicklungszeit sowie Kosten zu reduzieren. Als Anwendung zur Erstellung und Bearbeitung von Listen bietet sich Microsoft Excel an, da es einer breiten Masse bekannt und einfach zu bedienen ist. Die beiden nachstehenden Masterarbeiten von Hannes Schweigler und Nikola Mastilovich erläutern den Aufbau eines Codegenerators unter Zuhilfenahme von Excel-Listen.

[46] Vgl. Schmitt (2011), S. 119.

Schweigler verfasste seine Arbeit für das Unternehmen ematric gmbh, welches im Bereich Förder- und Verfahrenstechnik tätig ist, mit dem Schwerpunkt auf die Automatisierung von Lackieranlagen für Automobile. Die SPS-Programmierung einer solchen Maschine unterliegt einer standardisierten Vorgehensweise, die unter anderem das Kopieren und Parametrieren einer Vielzahl von Bausteinen vorsieht. Ziel war es, diese monotone Arbeiten für den Handbetrieb an den Codegenerator auszulagern, damit sich der Entwickler auf das Wesentliche konzentrieren kann: Den Automatikbetrieb. Durch die Umsetzung wurden 12,26 % der Kosten eines Projekts für die Porsche AG eingespart.[47]

Die Softwareplanung beginnt mit der Erstellung einer Bauteilliste in Excel, welche sich aus dem Stromlaufplan, Rohrleitungs- und Instrumentenschema sowie Anlagenlayout ergibt (siehe Tab. 4.2). Sie enthält neben den Bauteilbezeichnungen (*DIM*, *FG*, *Bauteil* und *Bezeichnung*) die entsprechende Nummerierung der Funktionsbausteine (*FB*) und deren Instanzdatenbausteine (*DB*). Bei den Templates handelt es sich um Funktionen oder Funktionsbausteine für Siemens-Steuerungen mit Platzhaltervariablen.[48]

Modul	FB	DB	DIM	FG	Bauteil	Bezeichnung	Template
Abwasseranlage							
Neutrapuffer 1 (nickelhaltig)	5000	5000	AB01	BA715	M11	Rührwerk	Motor
	5001	5001	AB01	BA715	P21	Pumpe	Motor
	5002	5002	AB01	BA715	P31	Pumpe	Motor
	5003	5003	AB01	BA715	V17	Pneumatikklappe PS nickelhaltig	Ventil E
	5004	5004	AB01	BA715	V27	Pneumatikklappe Säurespülung	Ventil E
	5005	5005	AB01	BA715	V37	Pneumatikklappe Zone 9	Ventil E
	5006	5006	AB01	BA715	V47	Pneumatikklappe nickelhaltig	Ventil E
	5007	5007	AB01	BA715	V57	Pneumatikklappe nickelarm	Ventil E
	5008	5008	AB01	BA715	V67	Pneumatikklappe Kondensat	Ventil E
	5009	5009	AB01	BA715	V77	Pneumatikklappe Abluftwäscher	Ventil E
	5010	5010	AB01	BA715	V10	Pneumatikklappe Druckseite	Ventil
	5011	5011	AB01	BA715	V20	Pneumatikklappe Druckseite	Ventil
	5012	5012	AB01	BA715	V124	Klappe Filtrat FPR	Handklappe
	5013	5013	AB01	BA715	F12	Durchflussmessung	Analog IN
	5014	5014	AB01	BA715	L13	Füllstandsmessung	Analog IN
	5015	5015	AB01	BA715	L23	Übervollschutz	Digital IN

Tab. 4.2: Bauteilliste der ematric gmbh, Quelle: Schweigler (2012), S. 31.

Die Templates sind in AWL verfasst. Aus ihnen wird der Quellcode sowie die dazugehörige Symboltabelle als Textdatei exportiert (siehe Abb. 4.3). Der Codegenerator, programmiert in C#, importiert die Template-Informationen sowie die Excel-Tabelle und ersetzt Schlüsselwörter durch entsprechende Bauteilbezeichnungen (siehe Abb. 4.4). Das Ergebnis sind AWL-Quellen mit den dazugehörigen Symboltabellen, welche Siemens Entwicklungsumgebung SIMATIC STEP 7 einlesen kann. Zusätzlich zeichnet eine Log-Datei die Ereignisse des Generierungsprozesses auf.[49]

[47] Vgl. Schweigler (2012), S. 1 f u. S. 35 ff.
[48] Vgl. Schweigler (2012), S. 32 ff.
[49] Vgl. Schweigler (2012), S. 41 ff.

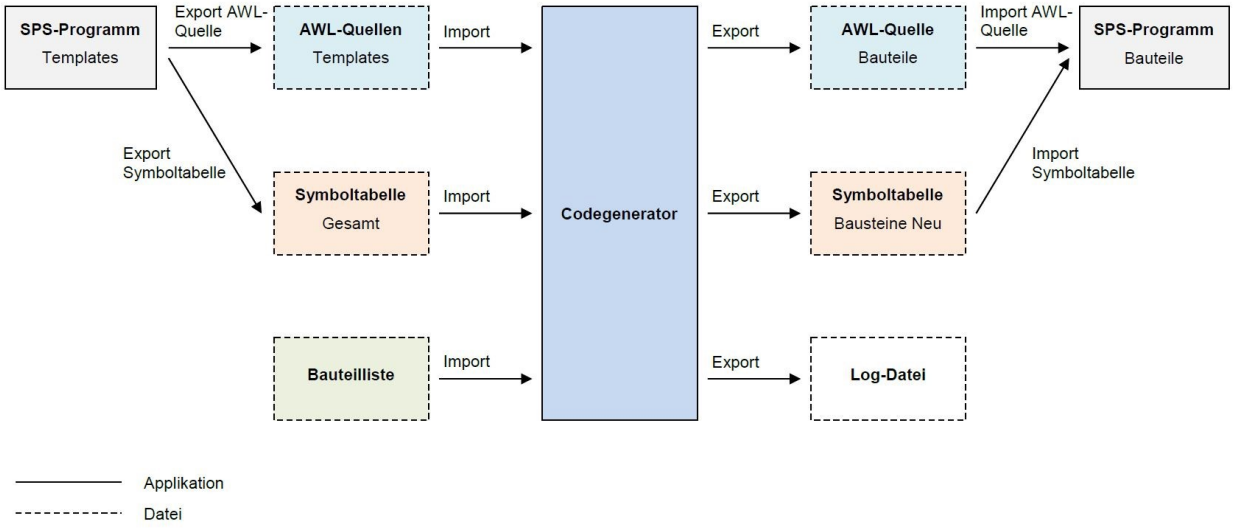


Abb. 4.3: Konzept des Codegenerators, Quelle: Schweigler (2012), S. 42.

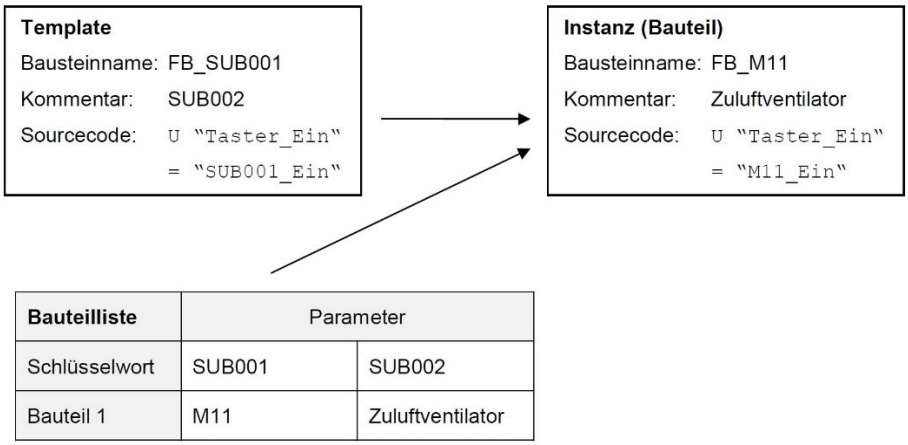


Abb. 4.4: Prinzip des Codegenerators, Quelle: Schweigler (2012), S. 44.

Mastilovich wendet in seiner Masterarbeit eine ähnliche Vorgehensweise an und ermöglicht dem Benutzer darüber hinaus das Modellieren von Abläufen mithilfe von Excel-Listen. Analog zu Herrn Schweigler verfasste er die Arbeit für die Firma Realcold Milmech Pty. Ltd mit dem Ziel einen Codegenerator zu entwickeln, welcher ein SPS-Programm für Rockwell-Steuerungen nach Unternehmensvorgaben erstellt, um den Mitarbeiter zu entlasten und Kosten einzusparen. Dabei konnten die Ausgaben für eine Gefrieranlage mit dem Namen Plate Freezer um 2,93 % gesenkt werden.[50]

Der Bediener trägt alle Ein- sowie Ausgänge der SPS mit ihren Funktionalitäten in Excel-Tabellen ein und legt den Ablauf der Maschine fest (siehe Abb. 4.5). Der Generator selbst ist mit Visual Basic for Applications (VBA) programmiert und nutzt die eingegebenen Informationen, um die Struktur der Anwendung festzulegen sowie Codeabschnitte, sogenannte Tags, einzufügen und mit Variablen zu verknüpfen. Das Resultat ist eine AWL-Textdatei, auch als L5K-Datei bezeichnet, welche von der Rockwell Entwicklungsumgebung RSLogix 5000 importiert werden kann.[51]

[50] Vgl. Mastilovich (2010), S. 1 u. S. 85.
[51] Vgl. Mastilovich (2010), S. 37 ff.

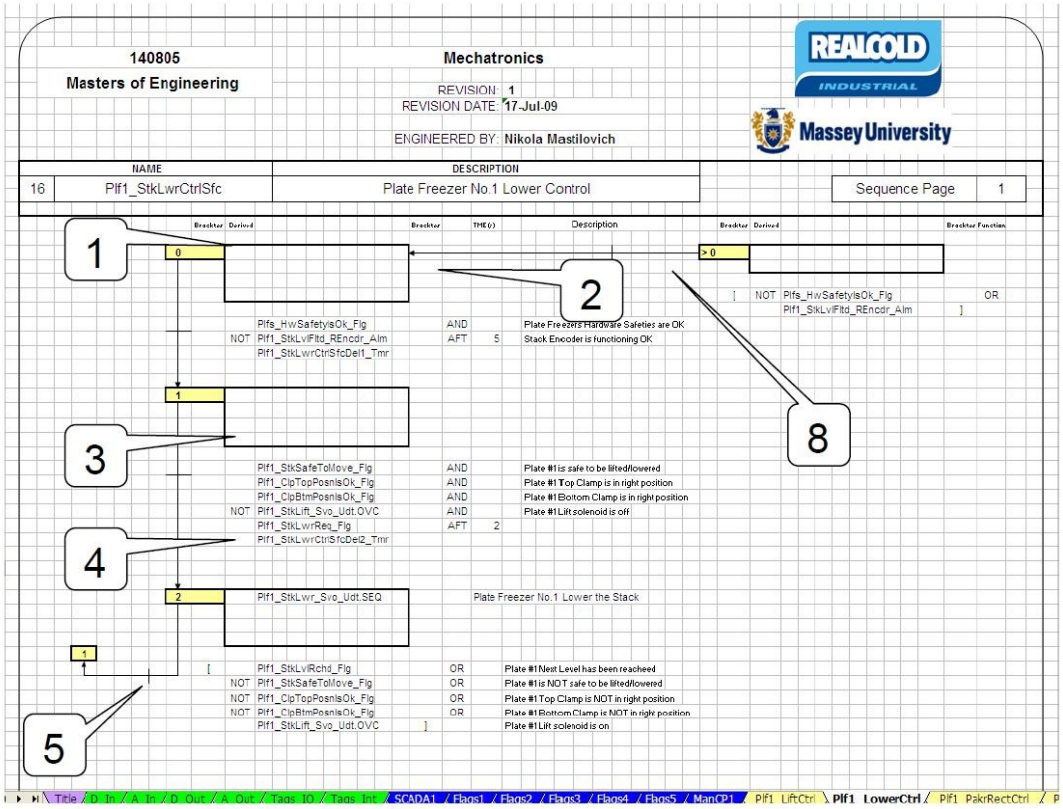
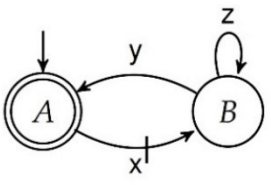


Abb. 4.5: Excel-Tabelle für das Ablaufdiagramm, Quelle: Mastilovich (2010), S. 73.

4.3 Petri-Netze und Automaten

Wie aus dem vorhergehenden Kapitel ersichtlich, ist es in der Automatisierungstechnik notwendig statische und dynamische Komponenten des SPS-Programms zu modellieren. Um Abläufe darzustellen, eignen sich besonders Automaten oder Petri-Netze. Im Nachstehenden wird die Masterarbeit von Nadine Gohert vorgestellt, die einen Codegenerator für die beiden Darstellungsformen entwickelt hat. Darüber hinaus sei auf die Werke[52] von Georg Frey der Universität Saarland verwiesen, welche die Einsatzgebiete von Petri-Netzen in der Steuerungstechnik behandeln.

Die Darstellung der Zustände und Ereignisse eines Automaten sind durch boolesche Variablen möglich. Abb. 4.6 zeigt die Transformation in SPS-Code. Sobald im Zustand *A* das Ereignis *x* eintritt, erfolgt ein Wechsel in den Zustand *B*, der über das Ereignis *y* verlassen wird.[53]




```
IF "B" AND "y" THEN
   "A":=1; "B":=0;
END_IF;

IF "A" AND "x" THEN
   "B":=1; "A":=0;
END_IF;

"x":="A";
```

Abb. 4.6: Automat mit dazugehörigem SPS-Code in SCL, Quelle: Gohert (2014), S. 39 (leicht modifiziert).

[52] Vgl. Google Scholar (o.J.), Online-Quelle [1.12.2017].
[53] Vgl. Gohert (2014), S. 39 f.

Steuerungstechnisch interpretierte Petri-Netze (SIPN) erweitern Petri-Netze um die Beschreibung von Ein- und Ausgängen. Die Generierung von Code unterliegt denselben Regeln wie jenen der Automaten, da sich die Stellen und ihre Zustandsänderungen, auch Transitionen genannt, ebenfalls als boolesche Werte interpretieren lassen (siehe Abb. 4.7). Ist eine Prästelle (*"p1"*) belegt, die Poststelle (*"p2"*) frei und die Transition (*"Eingang1" OR "Eingang2"*) erfüllt, wird die Prästelle zurückgesetzt und die Poststelle gesetzt. Zusätzlich ist es möglich, Übergänge mit Anweisungen zu verknüpfen (*"Zaehler":="Zaehler"+1*).[54]

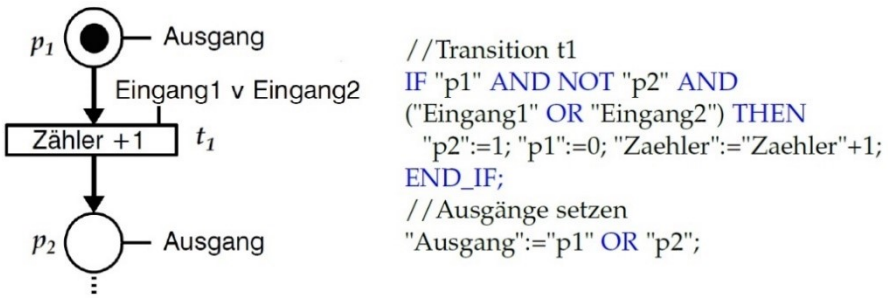


Abb. 4.7: Fragmentales SIPN mit SCL-Code, Quelle: Gohert (2014), S. 51.

Gohert wählt das frei erhältliche Programm DESTool[55] der Universität Erlangen-Nürnberg um Automaten zu modellieren und die Toolbox SPNBOX[56] für MathWorks MATLAB um Petri-Netze darzustellen. Der in C# erstellte Codegenerator ACArrow bildet das Bindeglied zwischen den Entwicklerwerkzeugen DESTool sowie MATLAB und der gewünschten SPS (siehe Abb. 4.8). Die Umwandlung der Eingangsdateien in Programmcode erfolgt wie zuvor erläutert (siehe Abb. 4.6 und Abb. 4.7) und die Resultate stehen entweder in Siemens-Syntax (AWL und SCL) oder in IEC 61131-3-Syntax (AWL und ST) für Steuerungen unterschiedlicher Hersteller zur Verfügung.[57]

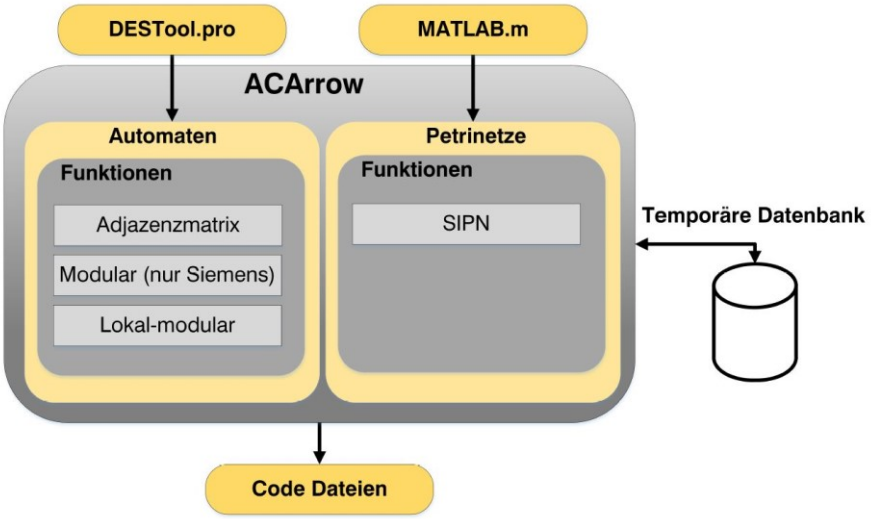


Abb. 4.8: Aufbau von ACArrow, Quelle: Gohert (2014), S. 58.

[54] Vgl. Gohert (2014), S. 41 u. S. 50 f.
[55] Vgl. Moor (2008), Online-Quelle [1.Dezember.2017].
[56] Vgl. LeTourneau University (o.J.), Online-Quelle [1.12.2017].
[57] Vgl. Gohert (2014), S. 37 f, 58 f u. S. 84.

4.4 Modellbasierte Entwicklung

Wie in Kapitel 4.3 erläutert verwendet Gohert eine Toolbox von MATLAB zur Umsetzung ihres Codegenerators. MATLAB bietet mit dem Simulink PLC Coder die Möglichkeit, IEC 61131-3 ST oder AWL aus Simulink-Modellen, Stateflow-Diagrammen oder Embedded MATLAB-Funktionen zu erzeugen (siehe Abb. 4.9). Das Resultat speichert die Anwendung in eine PLCopen XML-Datei (Kapitel 5.2 widmet sich ausführlich PLCopen XML) oder importiert es über Schnittstellen direkt in die Entwicklungsumgebungen unterschiedlicher Hersteller wie Beckhoffs TwinCAT 3 (The Windows Control and Automation Technology), Siemens SIMATIC Step 7 oder Rockwells RSLogix 5000.[58]

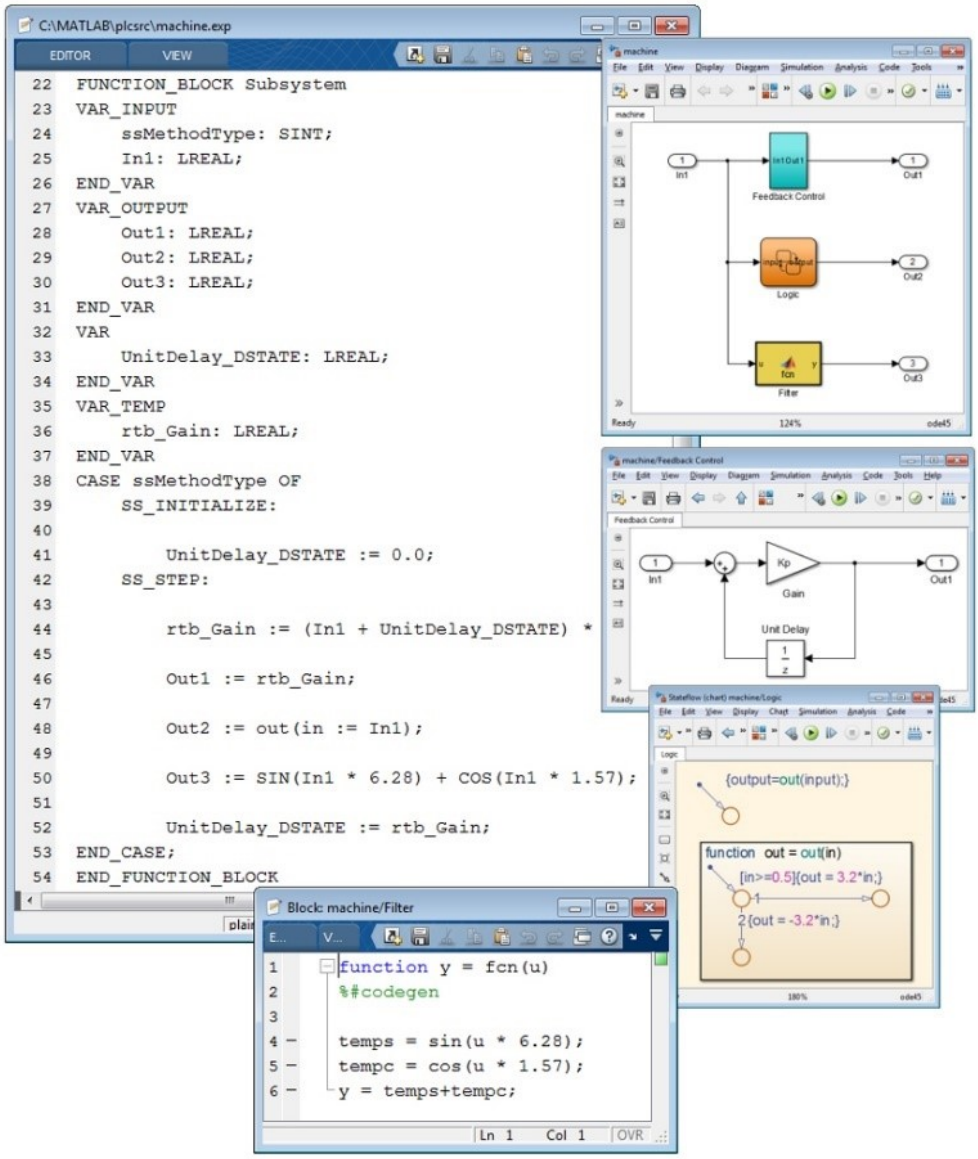


Abb. 4.9: MATLAB/Simulink PLC Coder mit seinen Funktionalitäten, Quelle: MathWorks (o.J.), Online-Quelle [1.12.2017].

Mit diesem Werkzeug unterstützt MATLAB die modellbasierte Entwicklung, da beispielsweise eine Regelstrecke samt PID-Regler in Simulink getestet und parametriert werden kann, um nach erfolgreicher Simulation den Code für die reale Umsetzung bereitzustellen.

[58] Vgl. MathWorks (o.J.), Online-Quelle [1.12.2017].

Industrie 4.0 skizziert eine Zukunft von intelligenten sowie vernetzten Systemen und sieht konventionelle Entwicklungsmethoden nicht in der Lage, die zunehmend komplexeren Anlagen handzuhaben. Die durchgängige modellbasierte Entwicklung über die gesamten Lebensphasen stellt die dritte Säule von Industrie 4.0 dar und sieht vor, Modelle aus vorherigen Stadien wiederzuverwenden, Abläufe anhand von Simulationsmodelle zu testen oder Maschinen virtuell in Betrieb zu nehmen und die Codegenerierung zu forcieren um Softwarefehler zu reduzieren.[59]

Hofmann, Menager und Mickelsons von der Firma Bosch Rexroth sowie Schweig von der Uni Duisburg-Essen nutzten die Modellierungssprache Modelica um einen Deltaroboter zu beschreiben und eine virtuelle Inbetriebnahme durchzuführen, mit dem Ziel lauffähigen SPS-Code zu erhalten. Das Modelica-Modell wird über zwei Compiler für die Bosch Entwicklungsumgebung IndraWorks aufbereitet und stellt dem Steuerungsprogramm Funktionsbausteine zur Verfügung (siehe Abb. 4.10). Der Vorgang ist voll automatisiert, sodass Änderungen am Modell direkt zu neugeneriertem Code führen.[60]

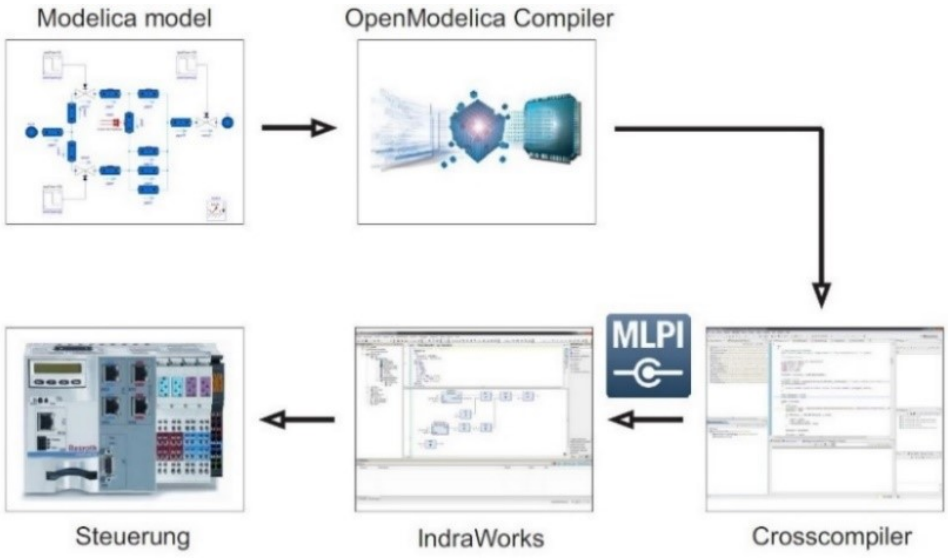


Abb. 4.10: Anwendungen vom Modelica-Modell bis zur Steuerung, Quelle: Hofmann/Menager/Schweig/Mikelsons (2015), S. 16.

Die modellbasierte Entwicklung geht mit den Konzepten der Objektorientierung Hand in Hand und dadurch findet neben der objektorientierten Erweiterung der IEC 61131-3 (siehe Kapitel 4.1) auch die UML (siehe Kapitel 3.2) vermehrt Einsatz in der Steuerungstechnik. Dabei kommen das Klassen- und Aktivitätsdiagramm sowie der Zustandsautomat (Zustandsdiagramm) zum Einsatz (siehe Abb. 4.11). Aus der Projektstruktur lässt sich das Klassendiagramm grafisch erstellen und das Aktivitätsdiagramm als auch der Zustandsautomat bieten eine Alternative zu den herkömmlichen Programmiersprachen um Abläufe festzulegen. Dadurch unterstützt die UML einen objektorientierten Softwareentwurf auf Basis von Modellen.[61]

[59] Vgl. Hofmann/Menager/Schweig/Mikelsons (2015), S.252 f.
[60] Vgl. Hofmann/Menager/Schweig/Mikelsons (2015), S. 263 ff.
[61] Vgl. Hess/Witsch (2011), S. 40 ff.

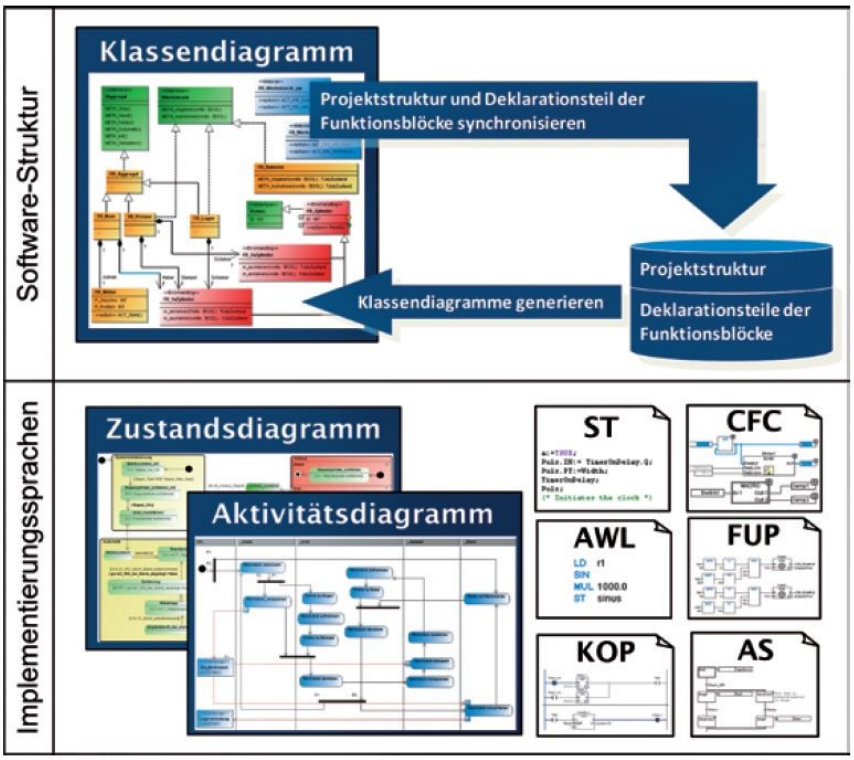


Abb. 4.11: Einsatz der UML in der Steuerungstechnik, Quelle: Hess/Witsch (2011), S. 41.

5 CODEGENERIERUNG FÜR BECKHOFF-STEUERUNGEN

Das Ziel dieser Masterarbeit ist die Entwicklung eines Codegenerators für Rundtaktanlagen der Firma Schunk Hoffmann Carbon Technology. In neuen Maschinen kommen Beckhoff-Steuerungen zum Einsatz. Aus diesem Grund beleuchtet das Kapitel die Möglichkeiten zur automatisierten Erstellung von SPS-Programmen speziell für den Hersteller Beckhoff.

Beckhoffs Entwicklungsumgebung TwinCAT ist in zwei Versionen erhältlich: TwinCAT 2 und TwinCAT 3. Die weiteren Ausführungen konzentrieren sich auf TwinCAT 3, da TwinCAT 2 das Automation Interface nicht in vollem Umfang unterstützt und die objektorientierte Erweiterung der IEC 61131-3 in keiner Weise. Darüber hinaus ist eine Modellierung mit der UML oder dem Simulink PLC Coder (siehe Kapitel 4.4) und der Ex- sowie Import von PLCopen XML-Dateien nur in TwinCAT 3 möglich. Weil der Einsatz der UML und des Simulink PLC Coders im vorherigen Anschnitt behandelt wurden, widmet sich das nachstehende Kapitel dem Automation Interface und dem PLCopen XML-Format.

5.1 Automation Interface

TwinCAT 3 bindet sich in Microsofts Entwicklungsumgebung Visual Studio ein und bietet mit den Automation Interface COM-fähigen Programmiersprachen, wie C# oder C++, eine Schnittstelle um SPS-Projekte, auch XAE Konfigurationen (eXtended Automation Engineering) genannt, automatisiert zu erstellen (siehe Abb. 5.1).[62]

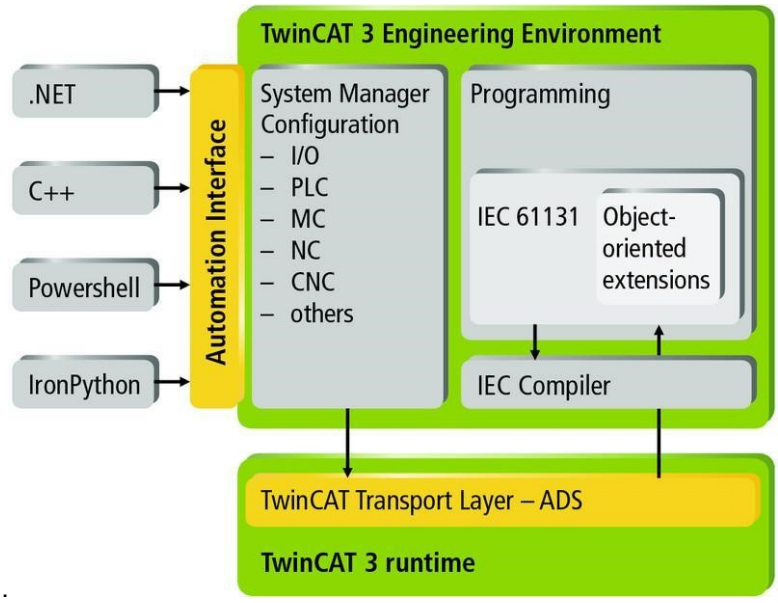


Abb. 5.1: Integration des Automation Interfaces in TwinCAT 3, Quelle: Beckhoff (2016), S. 8.

Der Aufbau von XAE Konfigurationen und dem Automation Interface basiert auf dem Visual Studio-Automatisierungsmodell[63]. Deshalb kann über die Schnittstelle jedes Element eines Projekts bearbeitet oder erzeugt werden.

[62] Vgl. Beckhoff (2016), S. 8 f.
[63] Vgl. Microsoft (o.J.), Online-Quelle [1.12.2017].

Dazu zählen:

- Allgemeine Einstellungen: Beispielsweise Konfigurationsdateien importieren.
- I/O: Zugriff auf Ein- und Ausgabeeinheiten der Steuerung.
- SPS: Erstellen von POEs und hinzufügen von Bibliotheken.
- Motion: Handhabung von NC-Programmen und der dazugehörigen Hardware.[64]

Darüber hinaus bietet das Automation Interface noch weitere Funktionen und stellt ein mächtiges Werkzeug dar, um einen Generator zu entwickeln, welcher im Stande ist, über Parametrierung ein vollständiges SPS-Projekt zu erstellen.

5.2 PLCopen XML

Bei PLCopen XML handelt sich um eine XML-Datei (siehe Kapitel 3.4), die den Vorgaben der PLCopen entspricht. Das Technical Committee 6 for XML der PLCopen definierte den Standard um den Austausch von Steuerungsprogrammen zu erleichtern sowie Drittanbietern den Import von SPS-Projekten zu ermöglichen. PLCopen XML unterstützt alle IEC 61131-3-Sprachen (siehe Tab. 4.1) und beinhaltet Informationen zum Projekt selbst bis hin zu den einzelnen Bausteinen (siehe Abb. 5.2).[65]

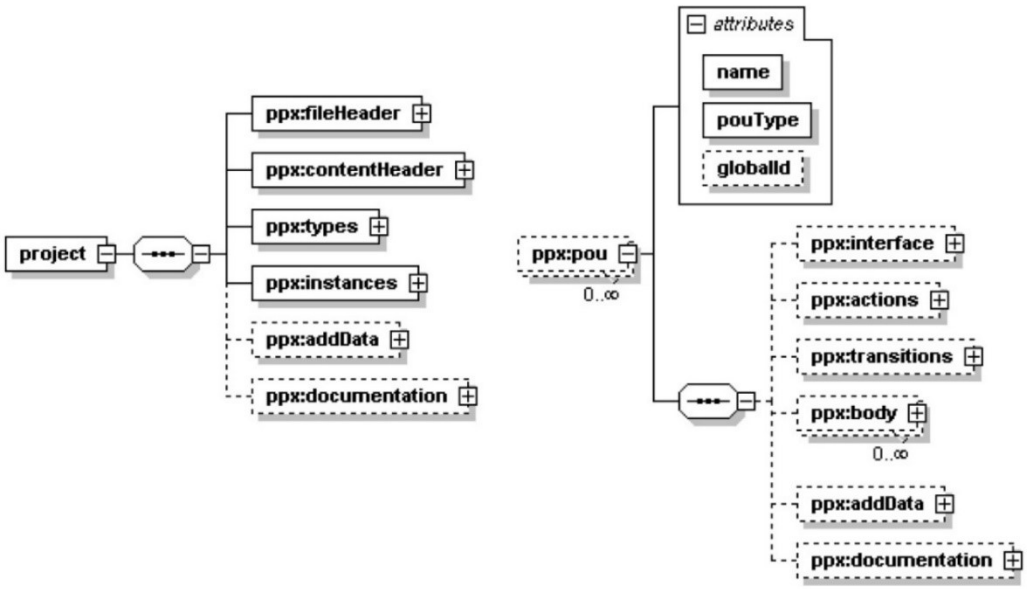


Abb. 5.2: Aufbau eines Projekts (links) und einer POU (rechts) laut PLCopen XML, Quelle: PLCopen (2009), S. 19 u. S. 24.

Als Beispiel zur Umsetzung eines PLCopen XML-Dokuments laut den Vorgaben aus Abb. 5.2 soll die Funktion *addition* dienen (siehe Abb. 5.3). Bei einer Funktion handelt es sich um eine Programmorganisationseinheiten (Program Organization Unit, POU), welche im Pfad *project/types/pous/pou* zu finden ist. Entsprechend dem Standard enthält das Element *pou* die Attribute *name* (*name="addition"*) sowie *pouType* (*pouType="function"*) und das Element *interface* spiegelt den Deklarationsteil der FC wieder. Der Anweisungsteil ist im Element *body* zu finden und beinhaltet die Anweisung *addition := zahl1 + zahl2;* als Strukturierter Text (siehe Abb. 5.4).

[64] Vgl. Beckhoff (2016), S. 10 ff.
[65] Vgl. PLCopen (o.J.), Online-Quelle [1.12.2017].

```
FUNCTION addition : INT

VAR_INPUT
    zahl1: INT;
    zahl2: INT;
END_VAR

addition := zahl1 + zahl2;

END_FUNCTION
```

Abb. 5.3: Funktion *addition* in ST, Quelle: Eigene Darstellung.

```
<project xmlns="http://www.plcopen.org/xml/tc6_0200">
 <types>
   <pous>
     <pou name="addition" pouType="function">
       <interface>
         <returnType>
           <INT />
         </returnType>
         <inputVars>
           <variable name="zahl1">
             <type>
               <INT />
             </type>
           </variable>
           <variable name="zahl2">
             <type>
               <INT />
             </type>
           </variable>
         </inputVars>
       </interface>
       <body>
         <ST>
           <xhtml xmlns="http://www.w3.org/1999/xhtml">addition := zahl1 + zahl2;</xhtml>
         </ST>
       </body>
     </pou>
   </pous>
 </types>
</project>
```

Abb. 5.4: Funktion *addition* als PLCopen XML-Dokument, Quelle: Eigene Dartstellung.

Eine PLCopen XML-Datei lässt sich beispielsweise über einen Generator erzeugen und automatisiert mit dem Automation Interface in ein Beckhoff SPS-Projekt einfügen. Alternativ kann eine POU manuell über die TwinCAT3-Menüfunktion *Export PLCopenXML* oder *Import PLCopenXML* exportiert sowie importiert werden.[66]

[66] Vgl. Beckhoff (2016), S. 66 f.

6 BETRACHTUNG DER METHODEN ZUR CODEGENERIERUNG

Die Kapitel 3 bis 5 stellen diverse Methoden zur Codegenerierung aus den Bereichen Informatik und Automatisierungstechnik vor, im Speziellen für Beckhoff-Steuerungen. Das Ziel dieser Masterarbeit ist es, einen Codegenerator für Rundtaktanlagen der Firma Schunk Hoffmann Carbon Technology zu entwickeln, wobei das resultierende SPS-Programm den Unternehmensvorgaben entsprechen soll. Diese Umstände sprechen für einen Generator, welchen der Benutzer über eine Oberfläche parametrieren kann und der in wenigen Schritten Code für Beckhoff-Steuerungen bereitstellt. Unter diesem Gesichtspunkt findet eine Betrachtung der zuvor erläuterten Methoden statt.

Informatik: Aufgrund der allgemeinen Anforderungen an einen Codegenerator (siehe Kapitel 2.3) bietet sich zur Umsetzung dieser Arbeit die Entwicklungsumgebung Visual Studio an. Sie ermöglicht das Erstellen von Visualisierungen und C# ist in Kombination mit dem .NET Framework eine mächtige Programmiersprache, mit der Templates, reguläre Ausdrücke sowie XML-Dateien verarbeitet werden können. Darüber hinaus bettet sich TwinCAT 3 in VS ein. Das Automation Interface dient als Schnittstelle zu C#.

Analyse, Architektur als auch das Design von Software sind in der Informatik stark von der UML geprägt. Da es sich bei C# um eine objektorientierte Sprache handelt, ist eine Verwendung der UML in diesen Prozessschritten unumgänglich. Die UML selbst als Basis für einen Generator nach dem Vorbild der UML/P zu nutzen (siehe Kapitel 3.3) oder einzelnen UML-Diagrammen als Generatoren einzusetzen (siehe Kapitel 4.4) scheint für diese Umsetzung nicht geeignet zu sein. Als Gegenargument zur UML/P ist anzuführen, dass eine domänenspezifische Anpassung, wie sie Martin Schindler in seiner Dissertation erläutert, als zu hoher Aufwand anzusehen ist (besonders da Beckhoff mit dem Automation Interface ein Werkzeug speziell für diese Aufgabe zur Verfügung stellt). Beckhoff unterstützt ebenfalls die Codegenerierung unter Zuhilfenahme einzelner UML-Diagramme aber die Objektorientierung findet nur langsam Einzug in die Automatisierungstechnik, so auch bei SHCT. Des Weiteren setzt diese Methode voraus, das der Bediener der Software mit der Notation der UML bzw. mit den objektorientierten Konzepten vertraut ist.

Automatisierungstechnik: Markus Irendorfer von der Firma Schunk Hoffmann Carbon Technology setzte bereits vor einigen Jahren einen Generator analog zur Arbeit von Nikola Mastilovich (siehe Kapitel 4.2) um. Jedoch ist dieser auf Grund der geringen Anwenderakzeptanz heute nicht mehr im Einsatz. Ein Vorteil von Excel ist sein hoher Bekanntheitsgrad und die daraus resultierende Vertrautheit mit der Bedienung für eine Vielzahl von Benutzern. Excel bietet aber nicht dieselben Optionen wie beispielsweise VS zur Erstellung einer Oberfläche. Generell ist in diesem Kontext die Verwendung von Excel als Visualisierung aus Gründen der Übersichtlichkeit zu hinterfragen.

Das Einlesen von Excel-Listen ähnlich der Implementierung von Hannes Schweigler (siehe Kapitel 4.2) ist in diesem Fall nicht zielführend, da eine RTA von SHCT nicht aus einer großen Menge an gleichen Bauteilen besteht. Die Parametrierung einzelner Programmbausteine sowie die Nutzung von Templates nach dem Vorbild von Hannes Schweigler sind auch für diese Arbeit vorstellbar.

Neben dem statischen Teil einer Anwendung für Rundtaktanlagen ist auch der dynamische zu generieren. Sofern der Ablauf einer RTA-Station nicht bereits aufgrund von Standardisierung durch SHCT vorgegeben ist, muss der Bediener des Codegenerators im Stande sein, einen Vorgang festzulegen. Hierfür spricht eine Modellierung auf Basis von Petri-Netzen oder Automaten, wie sie Nadine Gohert in Kapitel 4.3 behandelt. Ein Einsatz von MATLAB ist jedoch nicht vorgesehen, da die Anwendung ausschließlich in C# verfasst werden soll. Die Programmiersprache bietet in Kombination mit dem .NET Framework umfangreiche Werkzeuge, um dieses Ziel zu erreichen. Wie zuvor im Abschnitt zur Informatik erläutert, ist die Verwendung von Modelliersprachen, beispielsweise Modelica, zur Entwicklung des Generators oder die UML selbst als Basis für Generator nicht geplant.

Beckhoff-Steuerung: Das Automation Interface ist die Schnittstelle zu C# und ermöglicht darüber hinaus die automatisierte Erzeugung von XAE Konfigurationen (siehe Kapitel 5.1). Auch das PLCopen XML-Format für XML-Dokumente erweist sich als nützlich, weil diese XML-Dateien einerseits in eine Beckhoff-SPS importiert werden können und andererseits sind solche Dokumente dank der Standardisierung ebenfalls für Steuerungen anderer Hersteller verarbeitbar (siehe Kapitel 5.2). Darüber hinaus ist XML ideal geeignet, um Daten strukturiert zu speichern bzw. Parameter auszulesen (siehe Kapitel 3.4). All diese Argumente sprechen für eine Implementierung der beiden Methoden in den zu entwickelnden Codegenerator.

7 ANFORDERUNGEN AN DEN CODEGENERATOR

Nachdem die Methoden zur Codegenerierung aus den Bereichen Automatisierungstechnik sowie Informatik recherchiert und analysiert wurden, gilt es nun das erworbene Wissen in die Entwicklung des Codegenerators für das Unternehmen Schunk Hoffmann Carbon Technology einfließen zu lassen. Damit der Generator den Vorstellungen der Programmierabteilung entspricht, ist es entscheidend, die Anforderungen an die Software systematisch zu erfassen. Deshalb bietet dieses Kapitel zunächst eine Einführung in das Requirements-Engineering (RE), um anschließend eine Übersicht der Anforderungen an den Codegenerator zu geben. Den Abschluss des Kapitels bilden die Entscheidungen zum Programmaufbau, welche sich auf das RE stützen.

7.1 Requirements-Engineering

Das Requirements-Engineering hat seine Wurzeln in der Softwareentwicklung der 1970er Jahre. In den darauffolgenden Dekaden wurden die gewonnenen Erkenntnisse zu einem Standard zusammengefasst. Die ISO/IEC/IEE 29148 stellt die aktuellste Version dar.[67]

Neben dem Erfassen der Kundenanforderungen hat das RE zum Ziel, diese entsprechend zu dokumentieren, zu verifizieren und zu verwalten (siehe Abb. 7.1).

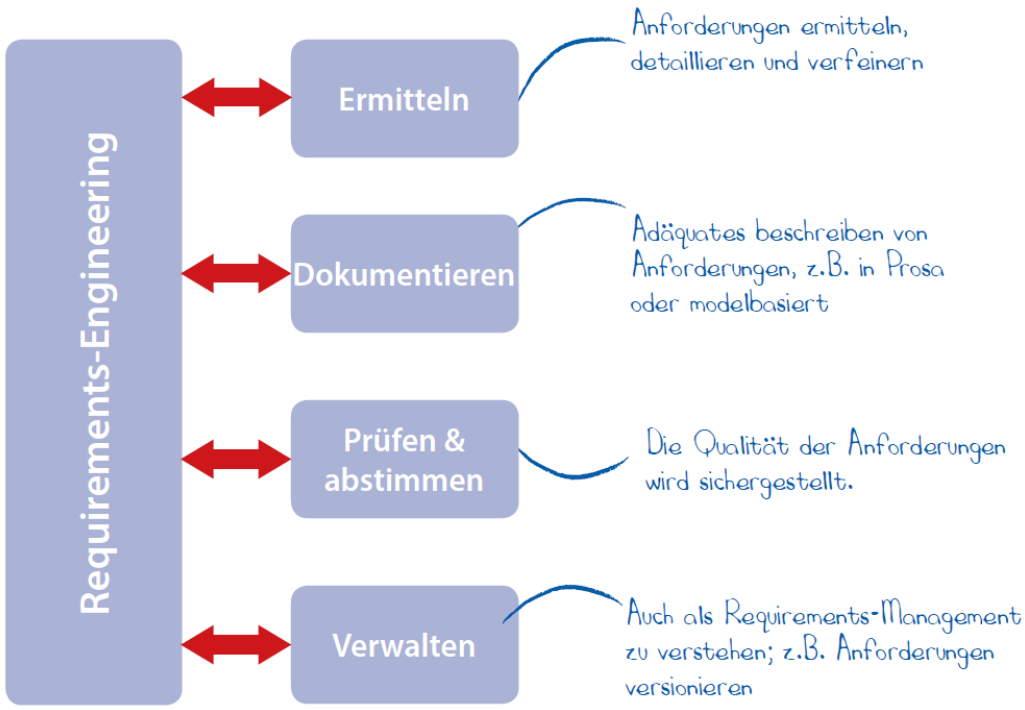


Abb. 7.1: Hauptaufgaben des Requirements-Engineerings, Quelle: Die SOPHISTen (Hrsg.) (2016), S. 10.

Grundsätzlich lässt sich zwischen funktionalen und nichtfunktionalen Anforderungen an ein Produkt unterscheiden. Erstere legen die Ergebnisse oder das Verhalten eines Systems fest. Letztere definieren die Eigenschaften, welche das System in seiner Gesamtheit betreffen, beispielsweise die Benutzeroberfläche, die Qualität der Software oder die Lieferbestandteile.[68]

[67] Vgl. Die SOPHISTen (Hrsg.) (2016), S. 7.
[68] Vgl. Sommerville (2012), S. 116.

Um Anforderungen zu dokumentieren, stehen natursprachliche oder modellbasierte Methoden zur Verfügung. Im Requirements-Engineering ist es üblich, Modelle anhand der UML zu gestalten, z. B. mit Use-Case- oder Klassendiagrammen. Dieses Vorgehen ermöglicht die Darstellung verschiedener Sichtweisen auf das System. Der große Nachteil der modellbasierten Dokumentation ist, dass eine Notation, wie jene der UML, erlernt werden muss. Aus diesem Grund setzt sich die natürliche Sprache oftmals durch um Anforderungen zu spezifizieren. Es liegt in der Natur des Menschen, Gesprochenes anders als vom Sender beabsichtigt zu interpretieren oder gar gänzlich misszuverstehen. Um diesen Umstand entgegen zu wirken, empfiehlt es sich, Vorlagen zu entwickeln, die helfen Mehrdeutigkeiten zu unterbinden. Ein Beispiel hierfür ist die Satzschablone der SOPHISTen (siehe Abb. 7.2).

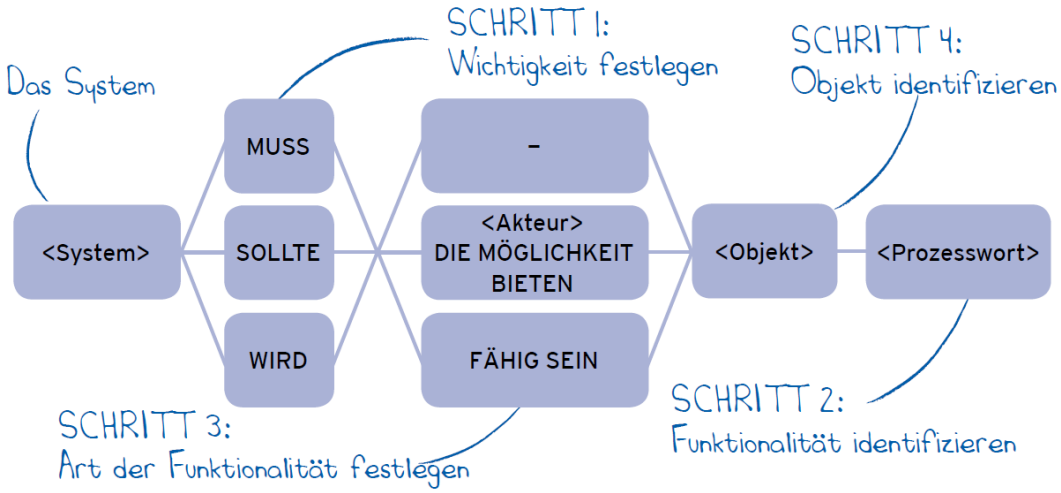


Abb. 7.2: Satzschablone, Quelle: Die SOPHISTen (Hrsg.) (2016), S. 35.

Das Festlegen der Anforderungen für den Codegenerator orientiert sich an der Satzschablone der SOPHISTen. Sie wird um das Feld *Status* erweitert, um den Fortschritt in den einzelnen Versionen des Requirements-Engineerings zu dokumentieren (siehe Tab. 7.1).

Wer/Wann	muss/sollte/wird	Handlung	Status
Codegenerator	muss	auf eine Bibliothek zugreifen können und Bausteine daraus importieren	umgesetzt
Codegenerator	muss	Stationen erstellen, die mindestens einer Leerstation entsprechen	umgesetzt

Tab. 7.1: Schablone für Anforderungen an den Codegenerator mit entsprechenden Beispielen, Quelle: Eigene Darstellung.

7.2 Anforderungen von Schunk Hoffmann Carbon Technology

Das Ziel dieser Masterarbeit ist es, einen Codegenerator für Rundtaktanlagen der Firma Schunk Hoffmann Carbon Technology zu erstellen. Um das Produkt entsprechend den Bedürfnissen und Vorstellungen der Programmierabteilung zu entwickeln, werden im ersten Schritt die Anforderungen an die Software mit Hilfe der zuvor vorgestellten Satzschablone festgehalten. Die nachfolgende Liste von Anforderungen stellt eine Sammlung der wichtigsten Forderungen dar (im Anhang ist das vollständige Dokument zum RE für den Codegenerator zu finden).

Wichtige funktionale Anforderungen an den Codegenerator:

- Er muss den Programmcode für eine Rundtaktanlage mit maximal acht Stationen generieren.
- Er muss Stationen erstellen können, welche mindestens einer Leerstation entsprechen.
- Er muss auf eine Bibliothek zugreifen können und Bausteine daraus in die Stationen importieren.
- Er muss die Stationsparameter, wie Stationsname oder -nummer, anpassen.
- Er muss den generierten Code über Kommentare kennzeichnen.
- Er muss die eingegebenen Konfigurationen als Datei speichern bzw. aus einer Datei laden.

Weitere funktionale Anforderungen an den Codegenerator:

- Der Software muss eine Bedienungsanleitung beiliegen.
- Die Benutzeroberfläche muss über die Funktionen *Speichern*, *Laden* und *Erstelle Projekt* verfügen.
- Alle Texte, welche für das Programm und die Benutzeroberfläche notwendig sind, müssen auf Deutsch sein.

7.3 Entscheidungen zum Aufbau des Codegenerators

Die Anforderungen an ein Programm bestimmen maßgeblich deren Aufbau. Aus den Forderungen von SHCT lassen sich für den Codegenerator drei grundsätzliche Use-Cases ableiten (eine differenzierte Aufteilung der Use-Cases findet sich im Anhang):

- Erstelle ein Projekt für eine Rundtaktanlage mit Stationen
- Speichere ein Projekt
- Lade ein Projekt

Neben diesen Aufgaben sind beim Entwurf des Generators auch die Prinzipien der Objektorientierung zu beachten. Sie lauten wie folgt:[69]

- Prinzip einer einzigen Verantwortung: Jedes Modul in einem Programm ist für eine bestimmte Aufgabe zuständig.
- Wiederholungen vermeiden.
- Offen für Erweiterungen, geschlossen für Änderungen: Ein Modul ist so aufzubauen, dass es an neue Situation angepasst werden kann ohne dessen inneren Aufbau zu verändern.
- Trennung der Schnittstelle von der Implementierung: Zwei Module kommunizieren immer über klar definierte Schnittstellen miteinander.
- Mach es testbar: Jedes Modul ist vor seiner Verwendung ausführlich zu testen. Vorzugsweise ist der Testvorgang automatisiert.

Es empfiehlt sich ebenfalls, bereits vorhandene Architekturmuster (wie in Kapitel 2.3 erläutert) für die zu erstellende Software in Betracht zu ziehen. Vor der Festlegung der Architektur ist die Frage zu klären, welcher Typ von Codegenerator (siehe Kapitel 2.2) zum Einsatz kommt. Die Anforderung ein Projekt zu speichern bzw. zu laden und somit mehrmals zu verwenden, spricht für einen aktiven Generator.

[69] Vgl. Lahres/Raýman/Strich (2016), S. 41-66.

Aufgrund der Tatsache, dass bereits Vorlagen für eine Rundtaktanlage und deren Stationen vorhanden sind, welche der Codegenerator parametrisiert bzw. in neues Format umwandelt, fällt die Wahl auf den Typ *Code Munger*.

Um die Software des Generators offen für Erweiterungen bzw. austauschbar zu halten und gleichzeitig eine klare Trennung der Aufgaben zu erzielen, kommt eine Schichtenarchitektur zum Einsatz (siehe Abb. 7.3).

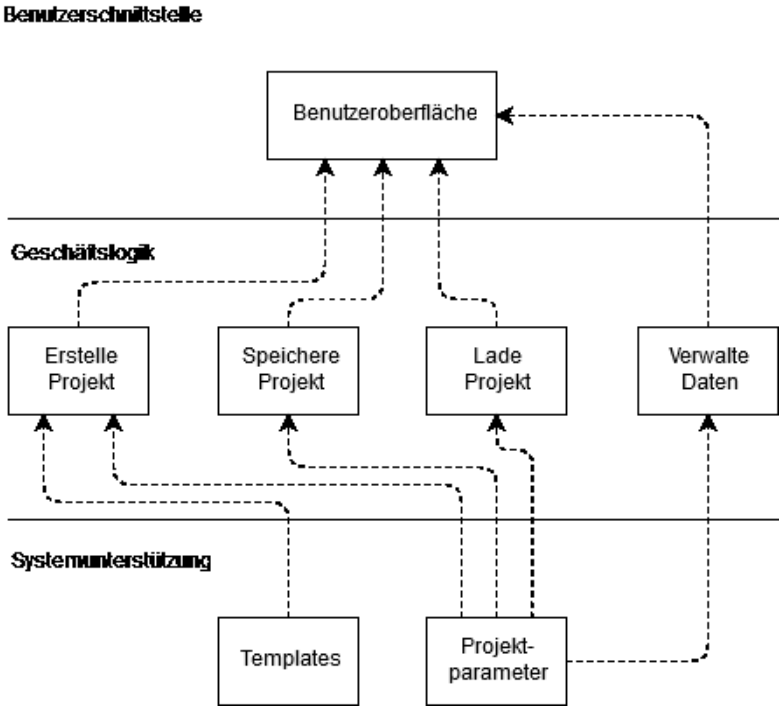


Abb. 7.3: Schichtenarchitektur Codegenerators mit den einzelnen Modulen, Quelle: eigene Darstellung.

Dabei beinhaltet die Schicht *Systemunterstützung* die Module *Templates* und *Projektparameter*. Die beiden Komponenten stellen der *Geschäftslogik* jene Daten zur Verfügung, die notwendig sind um ein Projekt zu erstellen bzw. eine Konfiguration zu speichern oder zu laden. Damit der Bediener mit all diesen Funktionalitäten über die *Benutzeroberfläche* interagieren kann, kommunizieren die einzelnen Elemente jeder Schicht über definierte Schnittstellen miteinander. Das Modul *Verwalte Daten* ist dafür verantwortlich, Änderungen an den Projektparametern durch den Bediener zu verarbeiten.

Damit die Software für den Codegenerator einer hohen Qualität entspricht und sie die geforderten Aufgaben fehlerfrei erfüllt, erfolgt eine Unterteilung in Use-Cases. Für jeden Use-Case ist ein Formular auszufüllen. Das Formular orientiert sich an der Use-Case-Schablone von Heide Balzert (siehe Abb. 7.4) und erweitert dieses um eine Liste von potentiellen Fehlerquellen (im Anhang finden sich alle Formulare zu den einzelnen Use-Cases). Somit ist sichergestellt, dass die einzelnen Module unter Berücksichtigung der zu erwartenden Fehler entwickelt werden und entsprechende Tests garantieren ihre ordentliche Funktionsweise.

Geschäftsprozeß: Name, bestehend aus zwei oder drei Wörtern (was wird getan?).
Ziel: globale Zielsetzung bei erfolgreicher Ausführung des Geschäftsprozesses.
Kategorie: primär, sekundär oder optional.
Vorbedingung: Erwarteter Zustand, bevor der Geschäftsprozeß beginnt.
Nachbedingung Erfolg: Erwarteter Zustand nach erfolgreicher Ausführung des Geschäftsprozesses, d.h. Ergebnis des Geschäftsprozesses.
Nachbedingung Fehlschlag: Erwarteter Zustand, wenn das Ziel nicht erreicht werden kann.
Akteure: Rollen von Personen oder andere Systeme, die den Geschäftsprozeß auslösen oder daran beteiligt sind.
Auslösendes Ereignis: Wenn dieses Ereignis eintritt, dann wird der Geschäftsprozeß initiiert.
Beschreibung:
1 Erste Aktion
2 Zweite Aktion
Erweiterungen:
1a Erweiterung des Funktionsumfangs der ersten Aktion
Alternativen:
1a Alternative Ausführung der ersten Aktion
1b Weitere Alternative zur ersten Aktion

Abb. 7.4: Schablone für Geschäftsprozesse (Use-Cases), Quelle: Balzert (2011), S. 64 f.

8 FUNKTIONSWEISE DES CODEGENERATORS

Nachdem die Anforderungen an den Codegenerator feststehen und bereits erste Entscheidungen zum Aufbau des Generators gefällt wurden, startet die Umsetzung der Applikation bei der Datenschicht. Für diese Schicht ist entscheidend, in welcher Form die Daten für den Codegenerator vorliegen. Aus diesem Grund beginnt das Kapitel mit der Betrachtung der zur Verfügung stehenden Optionen.

Der anschließende Abschnitt behandelt die Aufteilung der Datenschicht in die beiden Module: Templates und Projektparameter. Die Templates orientieren sich dabei an den Bestandteilen einer Rundtaktanlage von Schunk Hoffmann Carbon Technology: Ein Rahmen bildet das Grundgerüst für eine beliebige Anzahl an Stationen, die wiederum selbst aus einer Vielzahl an Bauteilen bestehen.

Aus den Vorlagen und Projektparametern muss der Generator ein ausführbares SPS-Programm für eine RTA erstellen. Die Erfüllung dieser Forderung ist Teil des letzten Kapitels.

8.1 Auswahl des Datenformats

Bei der Umsetzung ist als Erstes die Frage zu klären, in welcher Form die Daten für den Codegenerator vorhanden sein sollen. Prinzipiell bieten sich dazu zwei Optionen an.

Option 1: Die Programmierabteilung der Firma Schunk Hoffmann Carbon Technology erstellt ein TwinCAT 3 XAE-Projekt, das die Grundlage für eine Rundtaktanlage bildet und verwendet dieses für jede neue Maschine wieder. Die klaren Nachteile der Methode sind, dass die Vorlage jede erdenkliche Stationskonfiguration enthalten muss und eine manuelle Nacharbeit des Programms – abhängig von der Stationsanzahl – notwendig ist. Das Ziel dieser Masterarbeit ist die Auslagerung dieser Vorgänge an eine Applikation. Somit scheidet Option eins aus.

Option 2: Es steht ein TwinCAT 3 XAE-Projekt zur Verfügung, das ebenfalls die Basisinformationen zu einer RTA enthält und der Codegenerator passt die Vorlage automatisiert an die Anforderungen einer neuen Maschine an. Die Entwicklungsumgebung TwinCAT 3 ist im Stande, ein XAE-Projekt auf zwei Arten zu sichern: Als eine Reihe von Tc-Dateien oder als PLCopen XML-Datei. Um eine Entscheidung zwischen den beiden Datenformaten zu treffen, folgt eine genauere Betrachtung dieser Typen.

Standardmäßige speichert TwinCAT 3 ein XAE-Projekt als eine Sammlung von Tc-Dateien. Als Beispiel hierfür dient das Programm *STARTUP*, welches fixer Bestandteil einer Rundtaktanlage ist (siehe Abb. 8.1). Aus Abb. 8.1 geht hervor, dass das Format einer Tc-Datei dem einer XML-Datei entspricht. Das Element *POU* beinhaltet die beiden Subelemente *Declaration* und *Implementation*. Somit stehen die Informationen zum Deklarations- und Anweisungsteil eines Programmbausteins zur Verfügung. Der Nachteil beim Deklarationsbereich ist, dass alle Variablen als Klartext vorliegen. Dadurch muss der ganze Inhalt des Elements beispielsweise mit regulären Ausdrücken (siehe Kapitel 2.3) gefiltert werden, um eine neue Variable hinzuzufügen oder den Inhalt einer bereits vorhandenen Variable zu ändern.

```
<?xml version="1.0" encoding="utf-8"?>
<TcPlcObject Version="1.1.0.1" ProductVersion="3.1.4020.6">
  <POU Name="STARTUP" Id="{bdb26f0f-29cc-4002-9a6a-314ce39bb24b}" SpecialFunc="None">
    <Declaration><![CDATA[PROGRAM STARTUP
VAR
    FiSc: BOOL;
    Start: INT;
END_VAR
]]></Declaration>
    <Implementation>
      <ST><![CDATA[FiSc:=FALSE;

IF FiSc THEN
(*---------------------------------------------------------------
begin of coding area (insert code beyond this comment-block)
---------------------------------------------------------------*)
    MODULES.ModuleDone[1]:=TRUE;
(*---------------------------------------------------------------
end of coding area
---------------------------------------------------------------*)
END_IF;
(*All part holders activated*)
WPCD.WPCDsystem[1].active:=TRUE;
WPCD.WPCDsystem[2].active:=TRUE;
WPCD.WPCDsystem[3].active:=TRUE;
WPCD.WPCDsystem[4].active:=TRUE;
WPCD.WPCDsystem[5].active:=TRUE;
WPCD.WPCDsystem[6].active:=TRUE;
WPCD.WPCDsystem[7].active:=TRUE;
WPCD.WPCDsystem[8].active:=TRUE;

(*All parts NOK after Strartup*)
Globale_Variablen.reFiSc(CLK:=FiSc);]]></ST>
    </Implementation>
  </POU>
</TcPlcObject>
```

Abb. 8.1: Das Programm *STARTUP* als Tc-Datei, Quelle: Eigene Darstellung.

Darüber hinaus findet sich zum Aufbau der Tc-Datei keine Dokumentation. Das Attribut *Id* beinhaltet eine Zahlen- und Buchstabenkombination, welche die POU eindeutig im XAE-Projekt identifiziert. Ist sie nicht oder in falscher Form vorhanden, weil beispielsweise der Codegenerator das Programm erstellt, ist ein Import in die Entwicklungsumgebung TwinCAT 3 ausgeschlossen. Eine Nachfrage bei zuständigen Beckhoff-Mitarbeitern über die Zusammenstellung des Attributs *Id* blieb ergebnislos. Ein weiterer Nachteil der Tc-Datei ist das fehlende XML-Schema zur Validierung.

Ein XML-Schema legt den Aufbau bzw. die Struktur eines XML-Dokuments fest und überprüft ob die Datei den vorgeschriebenen Regeln entspricht. Dabei kann ein XML-Schema einfache und komplexe Typdefinitionen sowie Element- oder Attributdeklarationen enthalten. Das Schema selbst ist als XML-Anwendung mit entsprechendem Vokabular aufgebaut.[70]

Als Beispiel für ein XML-Schema dient die Überprüfung einer simplen Notiz in Form eines XML-Dokuments. Die Notiz besteht aus dem Wurzelelement *note* und beinhaltet die Elemente *to*, *from*, *heading* und *body* (siehe Abb. 8.2).

[70] Vgl. Vonhoegen (2015), S. 116 f.

```xml
<?xml version="1.0"?>

<note
xmlns="https://www.w3schools.com"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="https://www.w3schools.com/xml/note.xsd">
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

Abb. 8.2: Eine Notiz als XML-Datei, Quelle: w3schools (o.J.), Online-Quelle [1.12.2017].

Das dazugehörige XML-Schema enthält zusätzliche Informationen zu den einzelnen Elementen (siehe Abb. 8.3).

Beispielsweise muss der Inhalt des Elements *to* dem Datentyp *string* entsprechen, d. h. es darf nur eine Folge von Unicode-Zeichen vorhanden sein. Darüber hinaus definiert das Element *xs:sequence* die Reihenfolge der Elemente *to*, *from*, *heading* und *body*.[71]

Enthält eines der Elemente der Notiz-XML-Datei nicht den vorgegebenen Datentypen oder stimmt die Anordnung der Elemente nicht mit dem XML-Schema überein, lassen sich diese Abweichungen feststellen.

```xml
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="https://www.w3schools.com"
xmlns="https://www.w3schools.com"
elementFormDefault="qualified">

<xs:element name="note">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="to" type="xs:string"/>
      <xs:element name="from" type="xs:string"/>
      <xs:element name="heading" type="xs:string"/>
      <xs:element name="body" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

</xs:schema>
```

Abb. 8.3: Das XML-Schema zur Notiz als XML-Datei, Quelle: w3schools (o.J.), Online-Quelle [1.12.2017].

[71] Vgl. Vonhoegen (2015), S. 115 u. 123.

Dieses einfache Beispiel zeigt, wie unkompliziert der Test eines XML-Dokuments auf seine Korrektheit erfolgen kann. Um ein entsprechendes Schema für eine Tc-Datei festlegen zu können, ist es essentiell, den Aufbau des zu überprüfenden Dokuments zu kennen. Da aber keine Dokumentation zu der Tc-Datei vorliegt und die zuvor genannten Punkte ebenfalls gegen eine Verwendung der Datei als Datengrundlage für den Codegenerator sprechen, scheidet diese Option aus.

Die Alternative zu den Tc-Dateien heißt PLCopen XML (Kapitel 5.2 widmete sich bereits diesem standardisiertem XML-Format). Am Beispiel der Funktion *addition* als PLCopen XML-Datei ist ersichtlich, dass die POU im Gegensatz zur Tc-Datei in eine Vielzahl von Elementen aufgetrennt wird (siehe Abb. 8.4). Dadurch ist der Zugriff auf jede Variable oder den Anweisungsteil über entsprechende XML-Elemente möglich.

```
<project xmlns="http://www.plcopen.org/xml/tc6_0200">
 <types>
   <pous>
     <pou name="addition" pouType="function">
       <interface>
         <returnType>
           <INT />
         </returnType>
         <inputVars>
           <variable name="zahl1">
             <type>
               <INT />
             </type>
           </variable>
           <variable name="zahl2">
             <type>
               <INT />
             </type>
           </variable>
         </inputVars>
       </interface>
       <body>
         <ST>
           <xhtml xmlns="http://www.w3.org/1999/xhtml">addition := zahl1 + zahl2;</xhtml>
         </ST>
       </body>
     </pou>
   </pous>
 </types>
</project>
```

Abb. 8.4: Die Funktion *addition* als PLCopen XML-Datei, Quelle: Eigene Darstellung.

Darüber hinaus liegt für PLCopen XML eine Dokumentation über den Aufbau der Datei auf der Homepage von PLCopen[72] vor. Des Weiteren stellt die Organisation PLCopen ein XML-Schema zur Validierung von PLCopen XML-Dateien zur Verfügung, welches ebenfalls auf ihrer Homepage erhältlich ist. Diese Vorteile sprechen für den Einsatz von PLCopen XML als Datenbasis des Codegenerators. Abschließend ist anzumerken, dass auch andere Steuerungshersteller als Beckhoff einen Import von PLCopen XML-Dateien unterstützen.

[72] Vgl. PLCopen (o.J.), Online-Quelle [1.12.2017].

8.2 Aufbau der PLCopen XML-Datei

Nachdem die Entscheidung für das Datenformat feststeht, ist der nächste Schritt die Aufteilung der Daten in einzelne Module. Der zu entwickelnde Codegenerator hat den Generatortyp *Code Munger* zum Vorbild (siehe Kapitel 2.2). Der *Code Munger* durchsucht seine Quellen nach markanten Merkmalen und nutzt diese, um eine neue Datei zu erstellen. Dabei kommen häufig reguläre Ausdrücke, Parser oder Templates zum Einsatz. Der Codegenerator für das Unternehmen Schunk Hoffmann Carbon Technology soll bereits vorhandene Vorlagen zu einer Rundtaktanlage aufbereiten, damit der Bediener sie parametrieren kann. Der Generator erstellt aus den Templates und Vorgaben den Basisprogrammcode für eine neue Anlage. Somit steht die grundsätzliche Aufteilung der Daten fest: Templates und Projektparameter.

Die Templates orientieren sich am Aufbau einer RTA (siehe Kapitel 1.2). Sie besteht aus einem Rahmen mit einem Drehteller. Dieses Grundgerüst unterliegt selten Änderungen im Gegensatz zu den einzelnen Stationen. Eine Rundtaktanlage kann theoretisch eine beliebige Anzahl an Stationen enthalten. Die Maschinenbauabteilung von SHCT stellte in den letzten Jahren vermehrt Anlagen mit maximal acht Stationen her. Es ist allgemein ein Trend zu kompakten Nachbearbeitungsmaschinen mit einer geringen Stationsanzahl zu erkennen. Aus diesem Grund entstand auch die Forderung an den Codegenerator, das Programm für eine RTA mit einer variablen aber geringen Anzahl an Stationen zu erstellen.

Jede Station kann eine Vielzahl von Bauteilen enthalten, welche sich als Codebausteine im Programm der Anlage wiederfinden. Diese Bausteine befinden sich in Bibliotheken, welche die Programmierabteilung von SHCT verwaltet. Der Codegenerator muss laut Requirements-Engineering im Stande sein, jeder Station eine unbestimmte Anzahl von Bibliotheksbausteinen hinzuzufügen.

Aus den genannten Anforderungen ergibt sich folgender Aufbau für die Templates (siehe Abb. 8.5). Der Rahmen ist in der Lage eine Vielzahl an Stationen aufzunehmen und enthält alle Bestandteile einer Rundtaktanlage, welche immer vorhanden sind. Für eine Station ist eine Vorlage vorhanden, die einer Leerstation entspricht. Eine Leerstation ist innerhalb des Rundtaktanlagenprogramms voll funktionsfähig, enthält aber keine weiteren Elemente. Das Stationstemplate dient wiederum als Träger für eine unbestimmte Anzahl an Bibliotheksbausteinen, welche die Bestandteile einer Station repräsentieren.

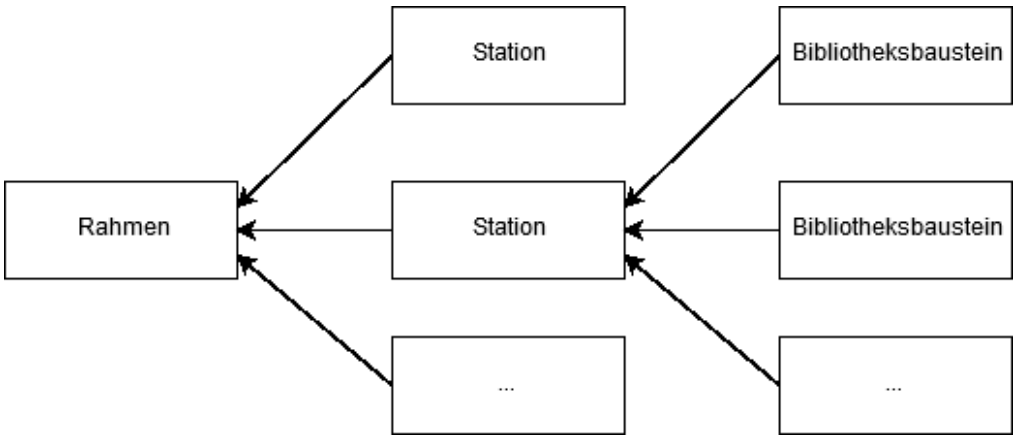


Abb. 8.5: Aufteilung der Templates für den Codegenerator, Quelle: Eigene Darstellung.

Das Ziel ist es, aus den Vorlagen mit den Projektparametern eine PLCopen XML-Datei zu erstellen, welche in die TwinCAT 3 Entwicklungsumgebung importiert werden kann. Dazu erstellt ein Programmierer von SHCT einmalig ein TwinCAT 3 XAE-Projekt, das den Rahmen mit der Leerstation enthält und exportiert es als PLCopen XML. Diese Datei bildet die Grundlage für die parametrierbaren Templates.

Beim Exportieren eines TwinCAT 3 XAE-Projekts in eine PLCopen XML-Datei tritt jedoch folgendes Problem auf: Die von TwinCAT 3 erstellte Datei entspricht zwar dem offiziellen PLCopen XML-Format, aber die Projektinformationen sind alle im Element *addData* enthalten (siehe Abb. 8.6).

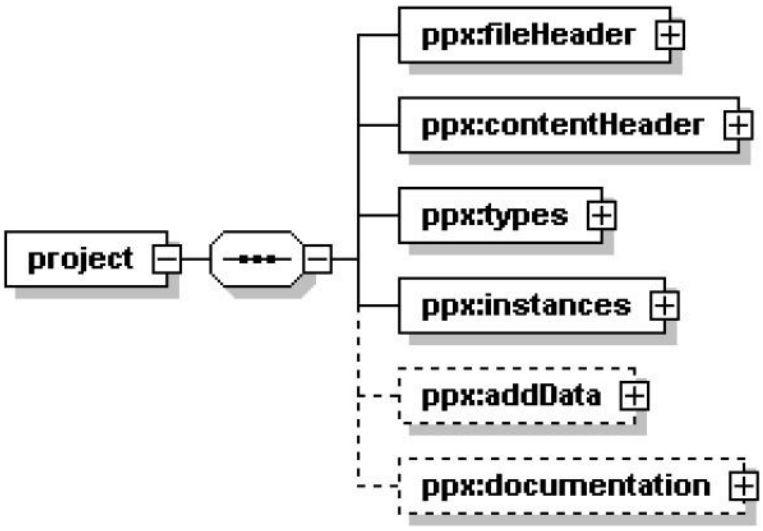


Abb. 8.6: Projektstruktur einer PLCopen XML-Datei, Quelle: PLCopen (2009), S. 19.

Die Organisation PLCopen sieht vor, dass eine PLCopen XML-Datei mit dem Wurzelelement *project* beginnt und die verpflichtenden Elemente *fileHeader*, *contentHeader*, *types* sowie *instances* enthält. Optional sind die Elemente *addData* und *documentation*.

Der *fileHeader* enthält herstellerspezifische Informationen zum XML-Dokument, beispielsweise den Firmen- oder Produktnamen. Im *contentHeader* sind der Projektname und Angaben zu den grafischen Programmiersprachen zu finden. Der typspezifische Teil *types* beinhaltet Strukturen oder POUs und das Element *instances* globale Variablenlisten eines SPS-Projekts. Der optionale Abschnitt *addData* dient dem Ersteller der PLCopen XML-Datei dazu, zusätzliche Informationen anzuhängen, z. B. Parameter, welche für den Import in eine anwenderspezifische Applikation notwendig sind. Aus diesem Grund unterliegt *addData* keinen inhaltlichen Restriktionen.[73]

Wie aus Abb. 8.7 ersichtlich entsprechen die Elemente *fileHeader* und *contentHeader* einer TwinCAT 3 PLCopen XML-Datei den Vorgaben von PLCopen XML. Jedoch bleiben die beiden Elemente *types* sowie *instances* leer und im Element *addData* befinden sich alle Informationen zu einem SPS-Projekt. Dieser Abschnitt entspricht zum großen Teil dem offiziellen PLCopen XML-Format, enthält aber auch spezifische Ausdrücke der Firma Beckhoff.

[73] Vgl. PLCopen (2009), S. 19-29 u. S. 59-61.

```
<project xmlns="http://www.plcopen.org/xml/tc6_0200">
  <fileHeader companyName="Beckhoff Automation GmbH" productName="TwinCAT PLC Control"
  productVersion="3.5.8.40" creationDateTime="2017-10-14T15:33:00.8032763" />
  <contentHeader name="Template_Station" modificationDateTime="2017-10-14T15:33:00.8032763">
    <coordinateInfo>
      <fbd>
        <scaling x="1" y="1" />
      </fbd>
      <ld>
        <scaling x="1" y="1" />
      </ld>
      <sfc>
        <scaling x="1" y="1" />
      </sfc>
    </coordinateInfo>
  </contentHeader>
  <types>
    <dataTypes />
    <pous />
  </types>
  <instances>
    <configurations />
  </instances>
  <addData>
    <data name="http://www.3s-software.com/plcopenxml/application" handleUnknown="implementation">
      <resource name="Template_Station">
        <task name="PlcTask" interval="PT0S" priority="20">
          <pouInstance name="MAIN" typeName="">
            <documentation>
              <xhtml xmlns="http://www.w3.org/1999/xhtml" />
            </documentation>
          </pouInstance>
          <addData>
            <data name="http://www.3s-software.com/plcopenxml/tasksettings" handleUnknown="implementation">
              <TaskSettings KindOfTask="Cyclic" Interval="10000" IntervalUnit="us">
                <Watchdog Enabled="false" TimeUnit="ms" />
              </TaskSettings>
            </data>
            <data name="http://www.3s-software.com/plcopenxml/objectid" handleUnknown="discard">
              <ObjectId>75456686-27f1-471d-aa35-928b1e7b0744</ObjectId>
            </data>
            ...
          </addData>
        </task>
      </resource>
    </data>
  </addData>
</project>
```

Abb. 8.7: Ausschnitt aus einer TwinCAT 3 PLCopen XML-Datei, Quelle: Eigene Darstellung.

Um aus dieser Datei die benötigten Templates zu extrahieren und sie entsprechend dem PLCopen XML-Standard aufzubereiten, kommt XSLT zum Einsatz. Bei XSLT handelt es sich um eine in XML definierte Sprache, welche die Transformation eines XML-Dokuments in ein XML-, HTML- oder Textdokument ermöglicht (siehe Kapitel 3.4).

8.2.1 Rahmen

Das TwinCAT 3 XAE-Projekt, welches als Vorlage für den Codegenerator dient, beinhaltet alle Programme, Strukturen sowie globale Variablenlisten für den Rahmen und zusätzlich die Leerstation (siehe Abb. 8.8). Im Ordner *DUTs* sind die Strukturen zu finden, im Ordner *GVLs* die globalen Variablenlisten und im Ordner *POUs* die Programme eines SPS-Projekts. Bei dem Programm *MODULE_template* handelt es sich um die Leerstation.

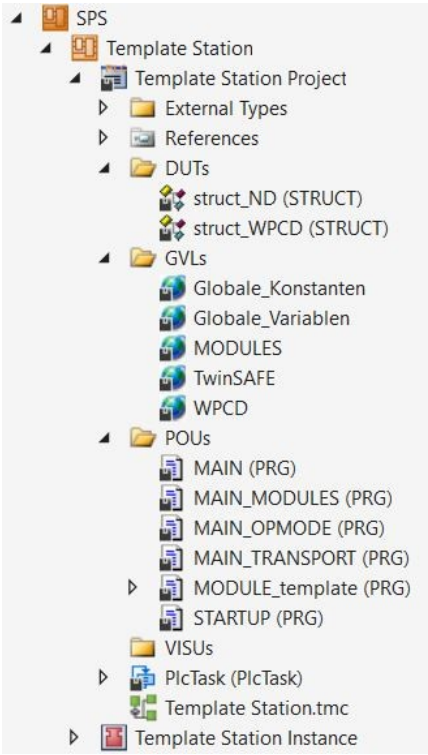


Abb. 8.8: Ausschnitt eines TwinCAT 3 XAE-Projekts, Quelle: Eigene Darstellung.

Wie im vorherigen Abschnitt erwähnt, ist eine XSLT-Datei notwendig, um das Template für den Rahmen aus der TwinCAT 3 PLCopen XML-Datei zu extrahieren. Das XSLT-Dokument besteht aus konstanten und variablen Teilen. Fixer Bestandteil ist die Grundstruktur einer PLCopen XML-Datei (siehe Abb. 8.9).

```
<xsl:transform version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="/">
        <project>
            <fileHeader companyName="SCT-AT" productName="Code Monkey"
            productVersion="1.0.0" creationDateTime="2017-09-01T15:00:00" />
            <contentHeader name="Maschine xy">
                <coordinateInfo>
                    <fbd>
                        <scaling x="1" y="1" />
                    </fbd>
                    <ld>
                        <scaling x="1" y="1" />
                    </ld>
                    <sfc>
                        <scaling x="1" y="1" />
                    </sfc>
                </coordinateInfo>
            </contentHeader>
            <types>
                <dataTypes/>
                <pous/>
            </types>
            <instances>
                <configurations>
                    <configuration name="cmConfig">
                        <resource name="cmResource"/>
                    </configuration>
                </configurations>
            </instances>
        </project>
    </xsl:template>
</xsl:transform>
```

Abb. 8.9: Die konstanten Bestandteile der XSLT-Datei für das Rahmen-Template, Quelle: Eigene Darstellung.

Die Elemente *dataTypes*, *pous* sowie *resource* werden durch XSLT-Anweisungen mit Strukturen, POUs und globalen Variablenlisten gefüllt. Als Beispiel für den Vorgang dient das Hinzufügen eines Programms. Das XSLT-Template *pouProgram* wählt anhand der Parameter *name* sowie *path* das gewünschte Programm aus der TwinCAT 3 PLCopen XML-Datei aus und wandelt es in das offizielle PLCopen XML-Format um (siehe Abb. 8.10).

```
<xsl:template name="pouProgram">
    <xsl:param name="name"/>
    <xsl:param name="path"/>
    <xsl:element name="pou">
    <xsl:attribute name="name"><xsl:value-of select="$name"/></xsl:attribute>
    <xsl:attribute name="pouType">program</xsl:attribute>
        <xsl:copy-of select="$path/pou [@name=$name]/interface"/>
        <xsl:copy-of select="$path/pou [@name=$name]/actions"/>
        <xsl:copy-of select="$path/pou [@name=$name]/body"/>
    </xsl:element>
</xsl:template>
```

Abb. 8.10: XSLT-Template zum Extrahieren eines POUs aus einer TwinCAT 3 PLCopen XML-Datei, Quelle: Eigene Darstellung.

In der XSLT-Datei selbst erfolgt ein Aufruf des Templates *pouProgram* im Element *pous* mit den geforderten Parametern, in diesem Fall für das Programm *MAIN* (siehe Abb. 8.11).

```
<types>
    <dataTypes/>
    <pous>
        <xsl:call-template name="pouProgram">
        <xsl:with-param name="name" select="'MAIN'"/>
        <xsl:with-param name="path" select="project/addData/data/resource/addData/data"/>
    </xsl:call-template>
    </pous>
</types>
```

Abb. 8.11: Aufruf des XSLT-Templates *pouProgram* in der XSLT-Datei für den Rahmen, Quelle: Eigene Darstellung.

Nach der Transformation der TwinCAT 3 PLCopen XML-Datei durch das XSLT-Dokument steht dem Codegenerator ein Template für den Rahmen nach PLCopen XML-Vorgaben zur Verfügung. Der große Vorteil der Datenaufbereitung durch XSLT ist, dass sich der Rahmen an ändernde Standards der Maschinenbauabteilung von SHCT anpassen lässt, unabhängig von der darüberliegenden Geschäftslogik des Generators. Der Codegenerator hat lediglich die Aufgabe, die parametrierten Stationen in das Element *pous* des Rahmen-Templates hinzuzufügen.

Eine weitere Aufgabe des Generators ist es, den Rahmen selbst an die Benutzereingaben anzupassen. Grundsätzlich sind dazu zwei Funktionen notwendig: Codebereiche in das Template einzufügen und Abschnitte durch die eingegebenen Parameter zu ersetzen. Nachdem ein Programmierer von Schunk Hoffmann Carbon Technology die Vorlage für den Codegenerator in TwinCAT 3 erstellt, liegt es nahe, die generatorspezifischen Stellen im Template durch Kommentare kennzuzeichnen. Kommentare haben den Vorteil, dass sie sowohl anpassbar als auch frei platzierbar sind, beim Export als PLCopen XML-Datei erhalten bleiben und durch reguläre Ausdrücke verarbeitet werden können. Zusätzlich erfüllt dieses Vorgehen die Forderung, Generate des Codegenerators durch Kommentare hervorzuheben.

Als Beispiel für ein Kommentar zum Einfügen eines Codeabschnitts bildet das Programm *MAIN_MODULES*. Es ist fixer Bestandteil einer RTA und verwaltet die einzelnen Stationen der Anlage. Durch den Kommentar *(*$ CM_FÜGE_HINZU $*)* ist der Generator im Stande, die Aufrufe der benutzerdefinierten Stationen einzufügen (siehe Abb. 8.12).

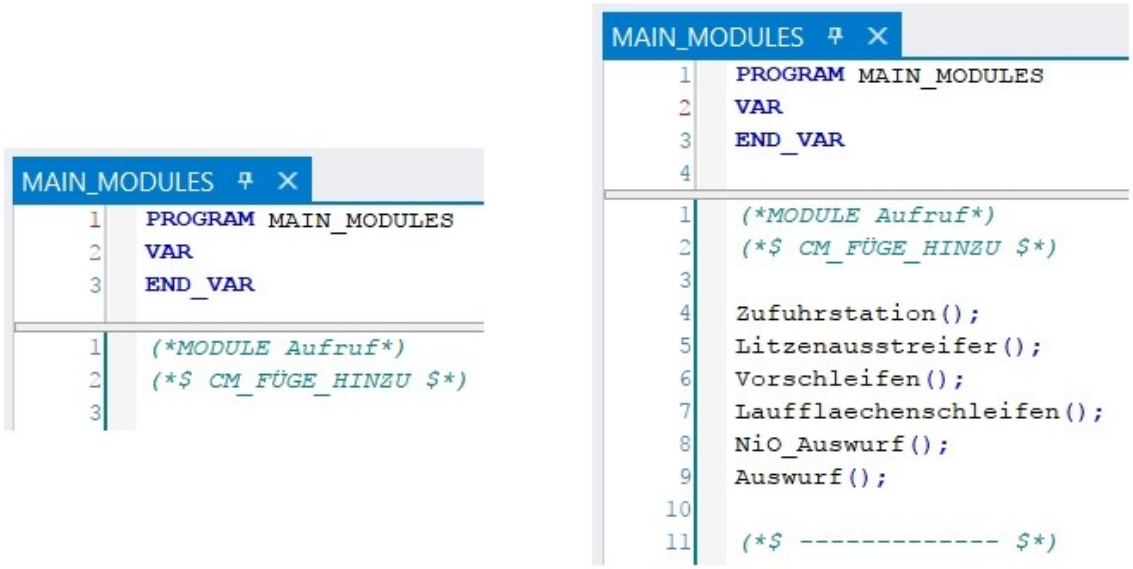


Abb. 8.12: Programm MAIN_MODULES mit Kommentar zum Hinzufügen vor (links) und nach (rechts) der Bearbeitung durch den Codegenerator, Quelle: Eigene Darstellung.

Neben dem Kommentar zum Einfügen weisen Kommentare zum Ersetzen in der TwinCAT 3-Vorlage den Codegenerator auf Stellen hin, die von den Projektparametern abhängig sind.

8.2.2 Leerstation

Das Template für die Leerstation befindet sich ebenfalls im TwinCAT 3 XAE-Projekt (siehe Abb. 8.8). Die Vorlage wird über ein entsprechendes XSLT-Dokument aus der TwinCAT 3 XML-Datei extrahiert und in das offizielle PLCopen XML-Format umgewandelt. Auch bei der Leerstation kommt unter anderem der Kommentar *(*$ CM_FÜGE_HINZU $*)* zum Einsatz um den Bereich für die hinzuzufügenden Bibliotheksbausteine zu kennzeichnen. Durch die Verwendung von Kommentaren und XSLT ist der Inhalt der Leerstation für den Codegenerator unerheblich und das Template bleibt für Änderungen durch die Programmierabteilung von SHCT offen.

8.2.3 Bibliotheksbausteine

Die letzte Gruppe der Templates bilden die Bibliotheksbausteine. In TwinCAT 3 ist eine Bibliothek als XAE-Projekt aufgebaut und lässt sich somit als PLCopen XML-Datei exportieren. Deshalb ist es möglich, die einzelnen Bausteine einer Bibliothek mittels XSLT für den Codegenerator aufzubereiten. Der Generator ist im Stande, einen ausgewählten Bibliotheksbaustein nach Benutzerangaben zu parametrieren und ihn in die entsprechende XML-Datei der Station einzufügen.

8.2.4 Projektparameter

Wie eingangs in Kapitel 8.2 erwähnt, besteht die Datenschicht des Codegenerators aus den Templates und den Projektparametern. Der Aufbau der Projektparameter orientiert sich ebenfalls an einer Rundtaktanlage und besteht aus den folgenden drei Klassen (siehe Abb. 8.13). Weitere Klassendiagramme zum Generator sind im Anhang zu finden.

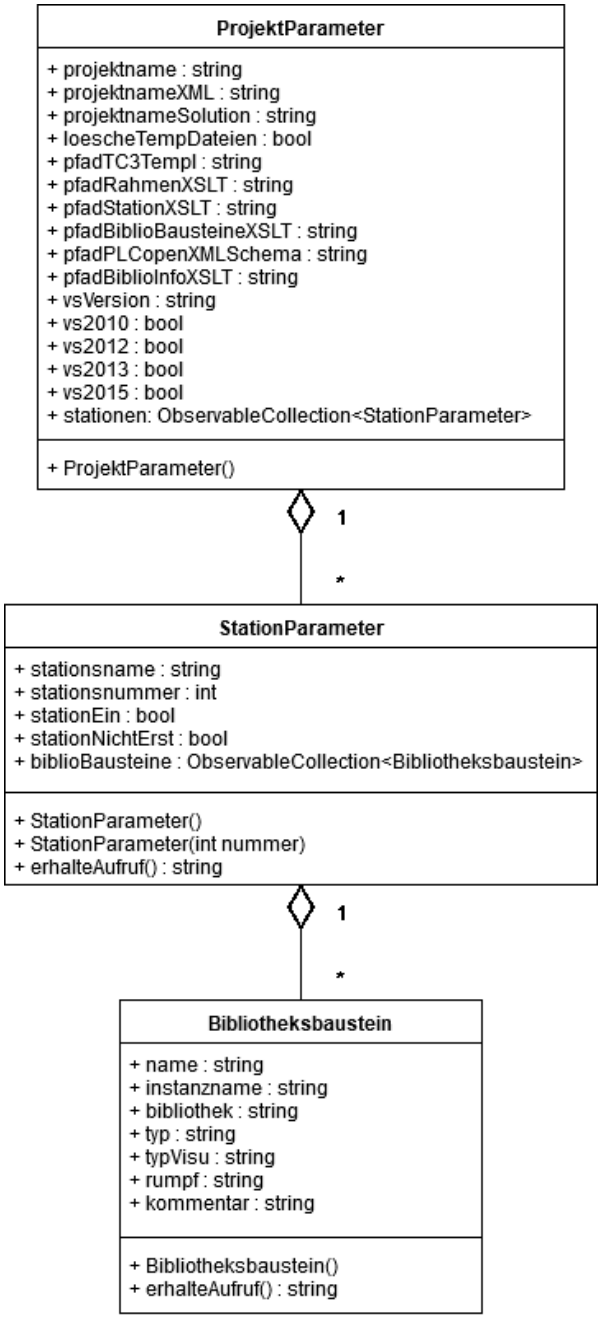


Abb. 8.13: Die Datenklassen des Codegenerators, Quelle: Eigene Darstellung.

Die Klasse *ProjektParameter* beinhaltet die allgemeinen Informationen zu einem Projekt und ist in der Lage eine beliebige Anzahl der Klasse *StationParameter* aufzunehmen. Diese Klasse speichert die Parameter zu den einzelnen Stationen und enthält eine frei wählbare Menge an Objekten der Klasse *Bibliotheksbaustein*, welche einen Baustein aus einer Bibliothek repräsentieren.

Der Vorteil von Klassen ist, dass der Codegenerator im Stande ist, sie als XML-Datei zu sichern bzw. sie aus einer XML-Datei zu laden. Dadurch können die Projektparameter unabhängig von den Templates gehandhabt werden. Die Kombination aus Templates und Projektparametern ermöglicht den Generator das Erstellen einer offiziellen PLCopen XML-Datei, welche das Basisprojekt für eine Rundtaktanlage von SHCT bildet. Zusätzlich kann eine Validierung des Generats anhand des PLCopen XML-Schemas erfolgen.

8.3 TwinCAT 3 XAE-Projekt

Bis zu diesem Zeitpunkt ist der Codegenerator im Stande, ein PLCopen XML-Dokument für eine RTA nach Benutzervorgaben zu erstellen. Die Datei selbst ist noch auf keiner Beckhoff-Steuerung ausführbar. Dazu muss sie in ein TwinCAT 3 XAE-Projekt importiert werden. Das PLCopen XML-Dokument enthält neben dem Programmcode eine Vielzahl an Bausteinen aus unterschiedlichen Bibliotheken. Die Bibliotheken müssen ebenfalls im TwinCAT 3 XAE-Projekt enthalten sein, um ein lauffähiges SPS-Programm zu erhalten. Das Automation Interface von Beckhoff (siehe Kapitel 5.1) ist im Stande, die geforderten Funktionalitäten umzusetzen.

Die Entwicklungsumgebung TwinCAT 3 integriert sich vollständig in Microsofts Visual Studio, d. h. die Struktur eines TwinCAT 3 XAE-Projekts basiert auf der Struktur eines Visual Studio-Projekts. VS stellt eine Anwenderschnittstelle, das Visual Studio DTE, zur Verfügung, um über eine externe Applikation das Grundgerüst für ein Visual Studio-Projekt zu erstellen. Mit dem Automation Interface ist möglich, ein TwinCAT 3 XAE-Projekt in das Visual Studio-Grundgerüst einzufügen (siehe Abb. 8.14).[74]

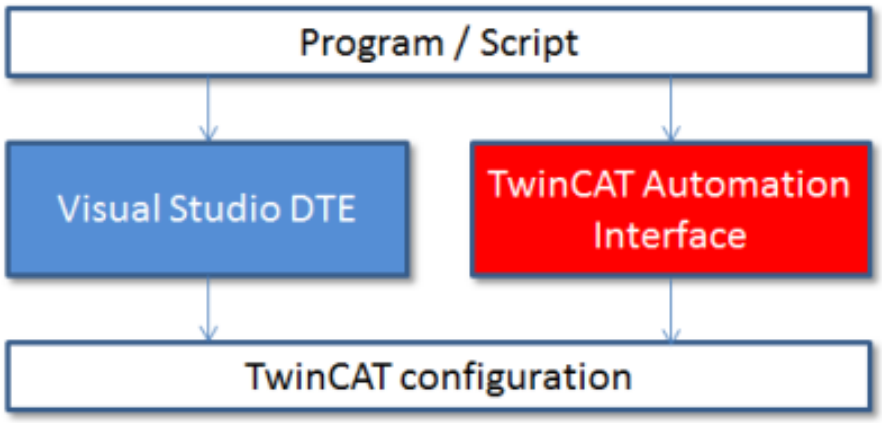


Abb. 8.14: Die Schnittstellen zu einem TwinCAT 3 XAE-Projekt, Quelle: Beckhoff (2016), S. 19.

Somit lässt sich ein leeres TwinCAT 3 XAE-Projekt automatisiert durch den Codegenerator erstellen. Das Automation Interface beinhaltet Klassen mit denen durch die baumartige Projektstruktur navigiert werden kann. Jedes Element entspricht einem sogenannten *Tree Item* (siehe Abb. 8.15), mit dem der Benutzer durch das Automation Interface vollständige Kontrolle über jeden Bestandteil des TwinCAT 3 XAE-Projekts erhält.[75]

[74] Vgl. Beckhoff (2016), S. 18 f.
[75] Vgl. Beckhoff (2016), S. 21-23.

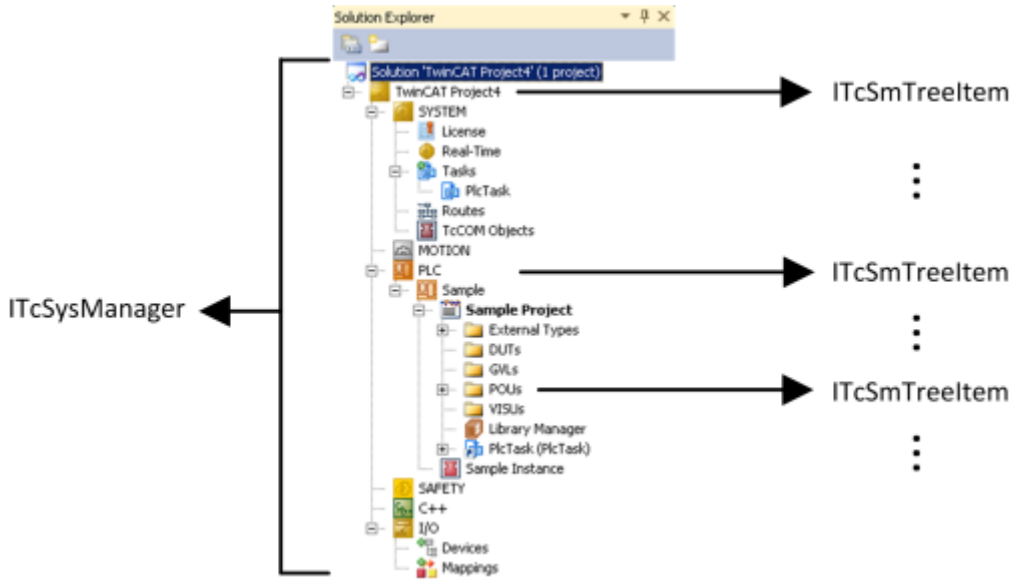


Abb. 8.15: Aufteilung des TwinCAT 3 XAE-Projekts in Klassen des Automation Interfaces, Quelle: Beckhoff (2016), S. 22.

Die vom Codegenerator erzeugte PLCopen XML-Datei enthält eine Menge an Bibliotheksbausteinen. Diese Bausteine sind nach einem Import in das TwinCAT 3 XAE-Projekt zwar vorhanden, aber aufgrund der fehlenden Referenzen zu den Bibliotheken nicht ausführbar. Die notwendigen Bibliotheken sind in der vom Bediener erstellten TwinCAT 3-Vorlage bereits vorhanden, in diesem Fall *HOS_Common* und *HOS_RTA* (siehe Abb. 8.16 rot umrandet).

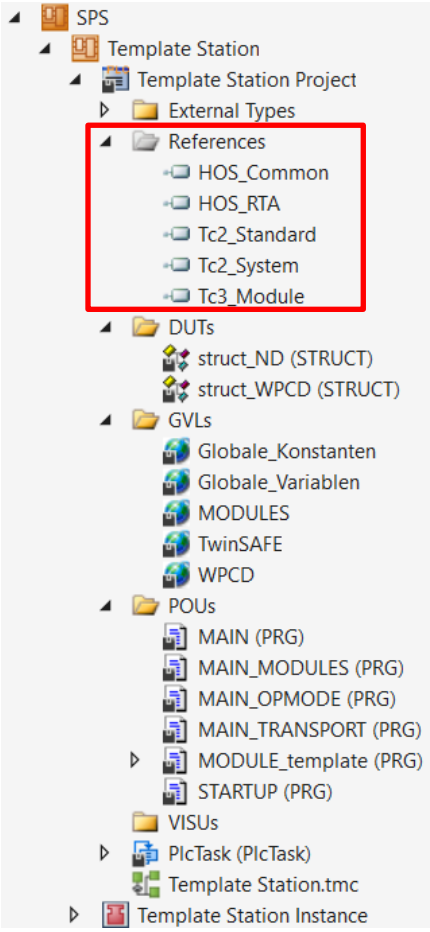


Abb. 8.16: Ausschnitt der TwinCAT 3 XAE-Vorlage, Quelle: Eigene Darstellung.

Diese Bibliotheken müssen auch im vom Codegenerator erzeugten TwinCAT 3 XAE-Projekt enthalten sein. Über das Automation Interface lässt sich ein leeres TwinCAT 3 XAE-Projekt erstellen.

Standardmäßig enthält dieses Projekt die drei TwinCAT 3-Bibliotheken *Tc2_Standard*, *Tc2_System* und *Tc3_Module* (siehe Abb. 8.16). Mit dem Automation Interface ist es möglich auf das Tree Item *References* zu navigieren und zusätzliche Bibliotheken hinzuzufügen. Dazu ist es notwendig den Namen und den Verteiler der Bibliothek zu kennen (siehe Abb. 8.17).[76]

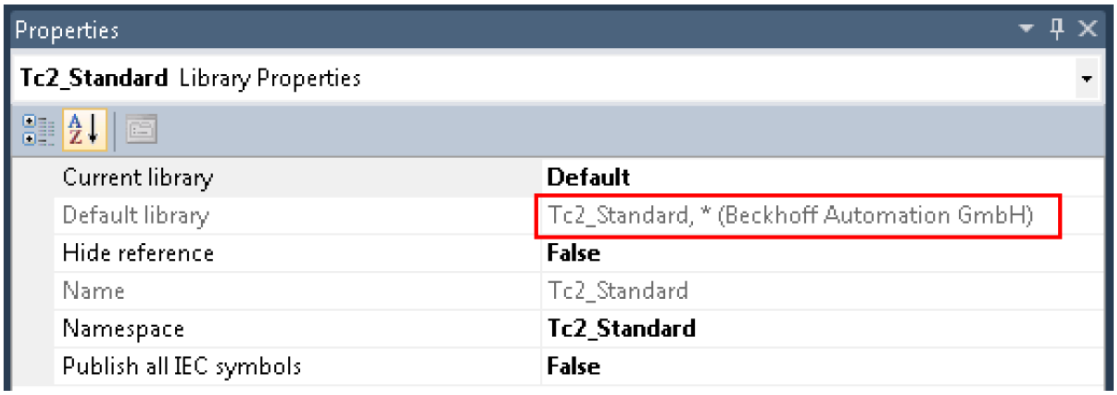


Abb. 8.17: Informationen zu einer TwinCAT 3-Bibliothek, Quelle: Beckhoff (2016), S. 59.

Da Mitarbeiter der Programmierabteilung von SHCT eine beliebige Anzahl an Bibliotheken im TwinCAT 3-Template festlegen können, muss der Codegenerator einen Zugriff auf die sich ändernden Bibliotheksinformationen erhalten. Nach dem Export der TwinCAT 3-Vorlage als PLCopen XML-Datei stehen exakt diese Parameter zur Verfügung (siehe Abb. 8.18).

```
<Libraries>
   <Library Name="#HOS_Common" Namespace="HOS_Common"
   DefaultResolution="HOS_Common, * (HOS)" />
   <Library Name="#HOS_RTA" Namespace="HOS_RTA"
   DefaultResolution="HOS_RTA, * (HOS)" />
   <Library Name="#Tc2_Standard" Namespace="Tc2_Standard"
   DefaultResolution="Tc2_Standard, * (Beckhoff Automation GmbH)" />
   <Library Name="#Tc2_System" Namespace="Tc2_System"
   DefaultResolution="Tc2_System, * (Beckhoff Automation GmbH)"/>
   <Library Name="#Tc3_Module" Namespace="Tc3_Module"
   DefaultResolution="Tc3_Module, * (Beckhoff Automation GmbH)" />
 </Libraries>
```

Abb. 8.18: Ausschnitt der Bibliotheksinformationen der TwinCAT 3 PLCopen XML-Datei, Quelle: Eigene Darstellung.

Das Element *Library* enthält im Attribut *DefaultResolution* den Namen und den Verteiler einer Bibliothek. In diesem Fall ist es mittels eines XSLT-Dokuments möglich, die Bibliotheksinformationen für den Codegenerator aufzubereiten. Der Generator nutzt reguläre Ausdrücke um die standardmäßigen TwinCAT 3-Bibliotheken herauszufiltern und fügt die übrigen Bibliotheken dem zu erzeugenden TwinCAT 3 XAE-Projekt über das Automation Interface hinzu.

[76] Vgl. Beckhoff (2016), S. 57-59.

Neben dem Erstellen eines TwinCAT 3 XAE-Projekts mit den dazugehörigen Bibliotheken ist der Import der generierten PLCopen XML-Datei ausständig. Das Automation Interface bietet auch für diese Anforderung entsprechende Methoden an.[77] Jedoch findet der Import einer PLCopen XML-Datei über das Automation Interface ohne den projektierten globalen Variablenlisten statt. Eine fehlerhaft erzeugte PLCopen XML-Datei durch den Generator kann ausgeschlossen werden, da die Validierung anhand des PLCopen XML-Schemas erfolgreich verläuft. In TwinCAT 3 ist der Import einer PLCopen XML-Datei auch manuell über einen entsprechenden Menüpunkt möglich. Wie aus Abb. 8.19 ersichtlich ist, enthält bei dieser Vorgehensweise die generierte PLCopen XML-Datei die globalen Variablenlisten, z. B. *MODULES*, *Globale_Konstanten* oder *Globale_Variablen*.

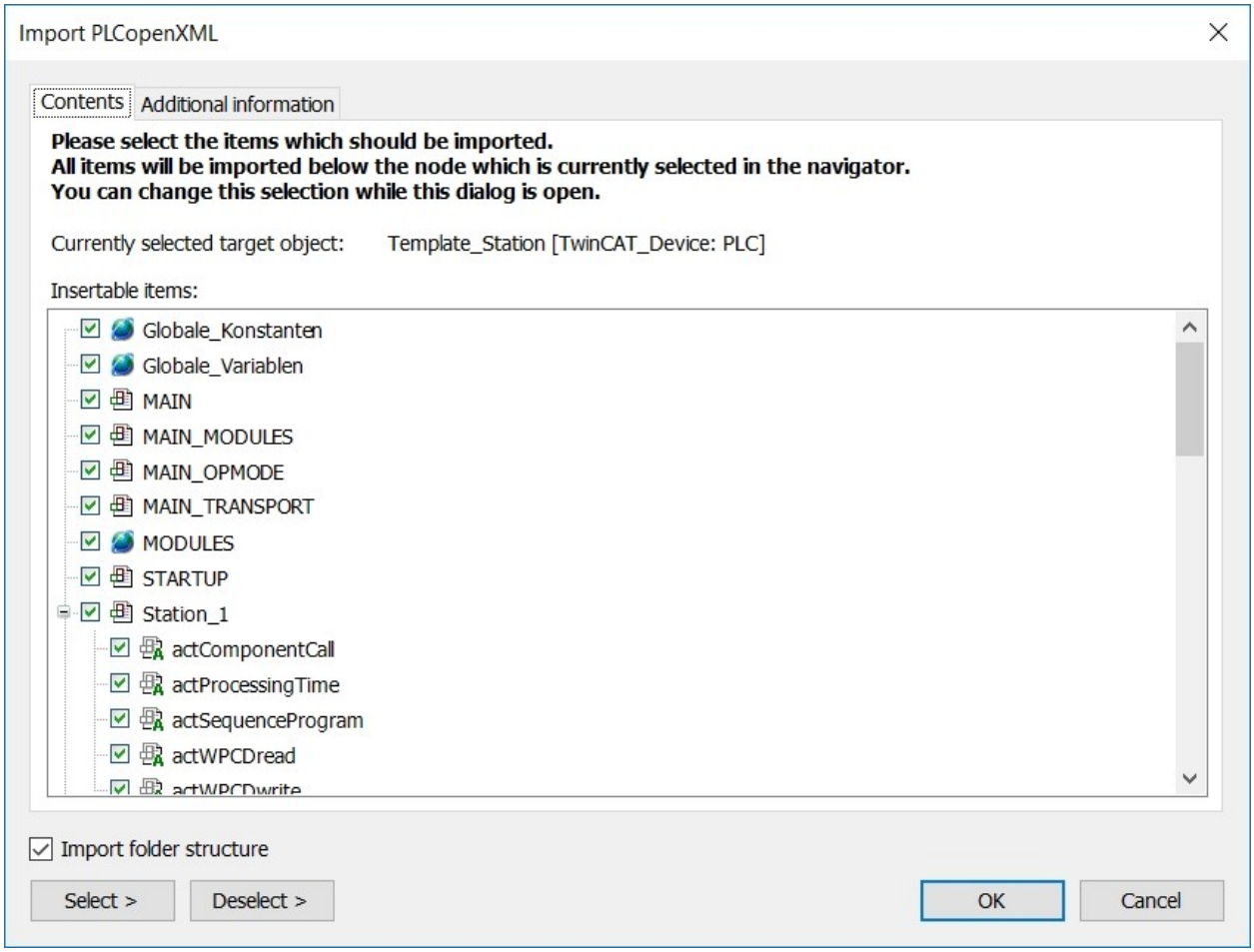


Abb. 8.19: Der Import-Dialog für eine PLCopen XML-Datei in TwinCAT 3, Quelle: Eigene Darstellung.

Zusammengefasst nutzt der Codegenerator parametrierbare Templates, um eine PLCopen XML-Datei zu erstellen, welche das Basisprogramm für eine Rundtaktanlage von Schunk Hoffmann Carbon Technology enthält. Zusätzlich ist der Generator im Stande ein TwinCAT 3 XAE-Projekt mit den dazugehörigen Bibliotheken zu erzeugen. In dieses Projekt importiert der Bediener manuell das generierte PLCopen XML-Dokument und erhält somit ein ausführbares SPS-Programm für eine RTA.

[77] Vgl. Beckhoff (2016), S. 66 f.

9 CODE MONKEY

9.1 Software-Prototyp

Um die Funktionalitäten des Codegenerators, welche Kapitel 8 erläutert werden zu evaluieren, findet die Entwicklung eines Prototyps in der Programmsprache C# als Konsolenanwendung statt. Dieser Anwendungstyp bietet den Vorteil, dass er bereits Befehle für eine einfache Form der Bedienerinteraktion zur Verfügung stellt. Der Nachteil ist, dass es sich dabei um keine grafische Benutzeroberfläche handelt, sondern um einen schlichten Dialog der auf textbasierten Kommandos basiert (siehe Abb. 9.1).

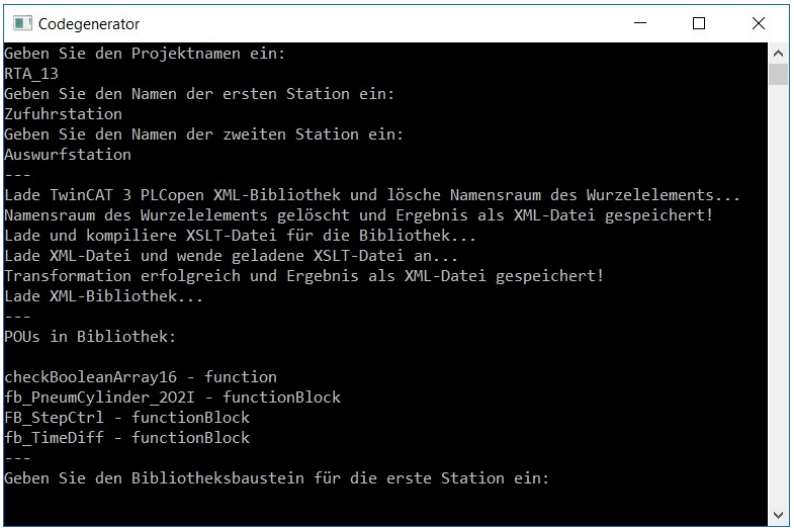


Abb. 9.1: Textbasierte Interaktion mit dem Prototyp des Codegenerators, Quelle: Eigene Darstellung.

Um der Programmierabteilung von Schunk Hoffmann Carbon Technology die Bedienung des Codegenerators zu erleichtern und den Generator selbst grafisch aufzubereiten, kommt eine WPF-Anwendung (Windows Presentation Foundation) zum Einsatz. Die Applikation erhält den Namen Code Monkey. Der englische Ausdruck *code monkey* bezeichnet in der Softwareindustrie etwas scherzhaft einen Programmierer, der keine Entscheidungen zum Aufbau oder zur Architektur der Anwendung trifft, sondern Code anhand von Anweisungen erstellt. Nachfolgend erfolgt eine kurze Einführung in die Handhabung von Code Monkey (eine ausführliche Bedienungsanleitung ist im Anhang zu finden).

9.2 PLCopen XML-Datei erstellen

Beim Start von Code Monkey öffnet sich der Reiter *Allgemein* (siehe Abb. 9.2). Im Punkt *Projektname* erfolgt die Namensvergabe der zu erstellenden Rundtaktanlage z. B. *RTA_13*. Der Codegenerator benennt die zu erzeugende PLCopen XML-Datei automatisch nach dem Projekt. Zusätzlich sind in diesem Abschnitt die Pfade zu den folgenden Dateien auszuwählen:

- Das TwinCAT 3 XAE-Projekt als PLCopen XML-Datei, welche als Vorlage für den Generator dient (siehe Kapitel 8.2).
- Eine Reihe von XSLT-Dateien, die notwendig sind, um das PLCopen XML-Template für den Generator aufzubereiten (siehe Kapitel 8.2).
- Das PLCopen XML-Schema zur Validierung der zu erstellenden PLCopen XML-Datei durch den Codegenerator.

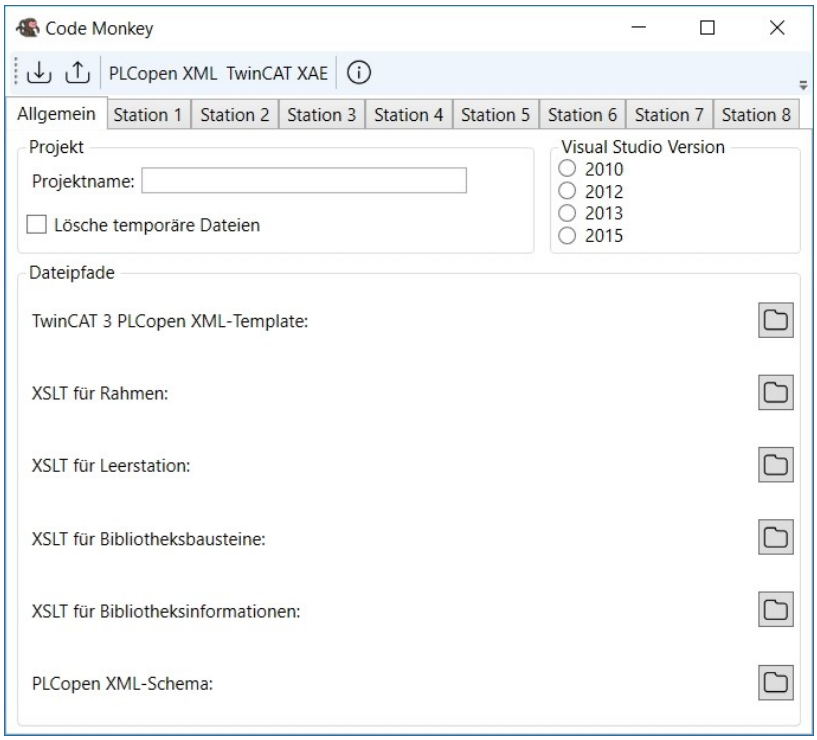


Abb. 9.2: Reiter *Allgemein* von Code Monkey, Quelle: Eigene Darstellung.

Neben den allgemeinen Einstellungen findet der Bediener in der Oberfläche acht identische Reiter zur Konfiguration der einzelnen Stationen einer Rundtaktanlage vor (siehe Abb. 9.3). Jeder Reiter bietet ein Feld an, um die Station zu benennen und durch das Klicken auf die Schaltfläche mit dem grünen Pluszeichen (siehe Abb. 9.3 rot umrandet) erfolgt das Hinzufügen einer beliebigen Anzahl von Bibliotheksbausteinen.

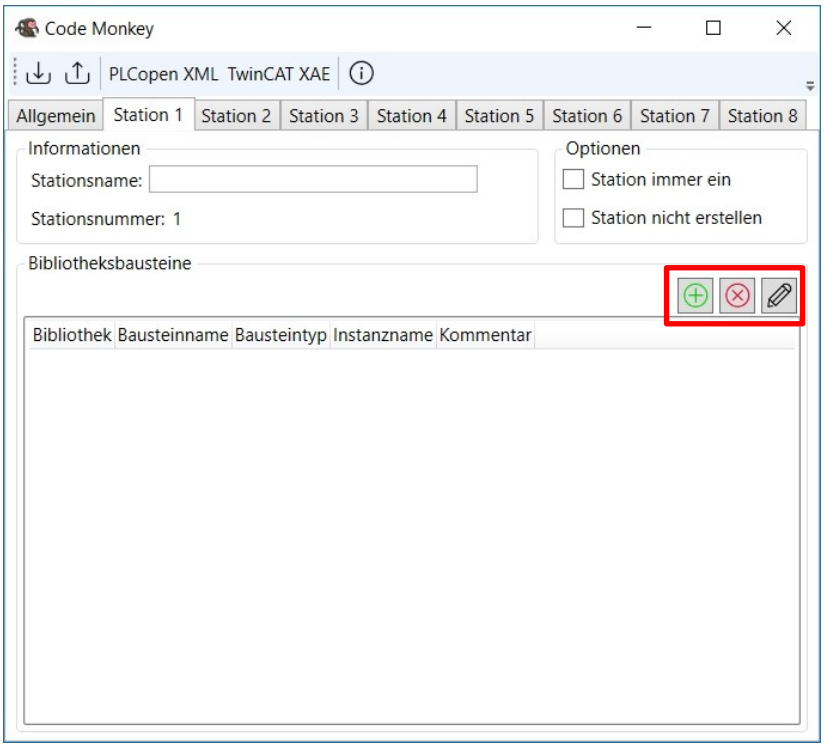


Abb. 9.3: Der Reiter einer Station in Code Monkey, Quelle: Eigene Darstellung.

Durch das Klicken auf die Schaltfläche erscheint ein Dialog, über den sich eine Bibliothek, welche als PLCopen XML-Datei vorhanden ist, öffnen lässt (siehe Abb. 9.4).

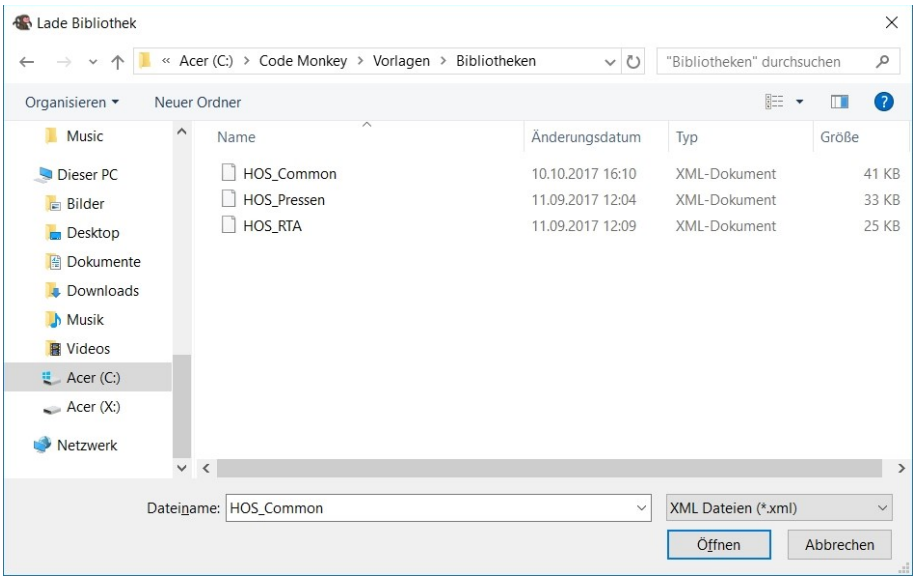


Abb. 9.4: Datei öffnen-Dialog für eine Bibliothek, Quelle: Eigene Darstellung.

Sobald die Auswahl einer Bibliothek feststeht, wendet der Generator die entsprechende XSLT-Datei an, um die darin enthaltenen Bausteine samt ihrem späteren Programmaufruf anzuzeigen (siehe Abb. 9.5). Wählt der Bediener einen Funktionsblock aus der Liste von Bibliotheksbausteinen, muss er einen dazugehörigen Instanznamen angeben. Unabhängig vom Bausteintyp steht es dem Benutzer frei die Funktion oder den Funktionsblock mit einem Kommentar zu versehen.

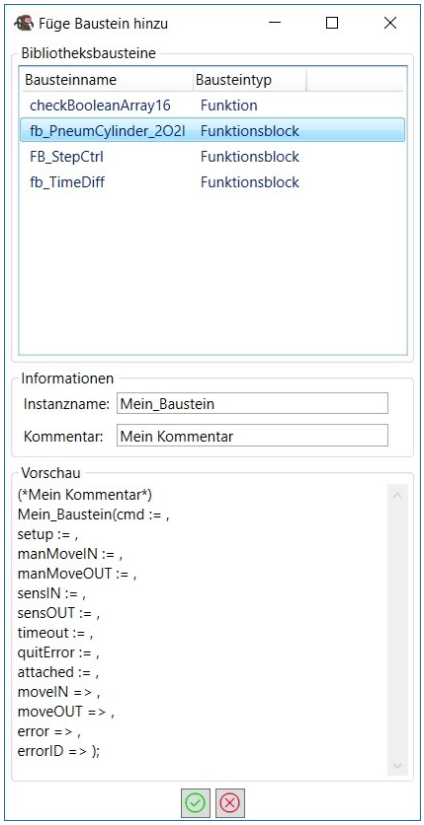


Abb. 9.5: Dialog zum Hinzufügen eines Bibliotheksbausteins in Code Monkey, Quelle: Eigene Darstellung.

Nach Bestätigung der Angaben befindet sich der Bibliotheksbaustein in der Liste von Stationsbausteinen (siehe Abb. 9.6). Um einen Baustein aus der Liste zu löschen, ist dieser anzuwählen und anschließend auf die Schaltfläche mit dem rot eingekreisten Kreuz zu klicken (siehe Abb. 9.6 rot umrandet).

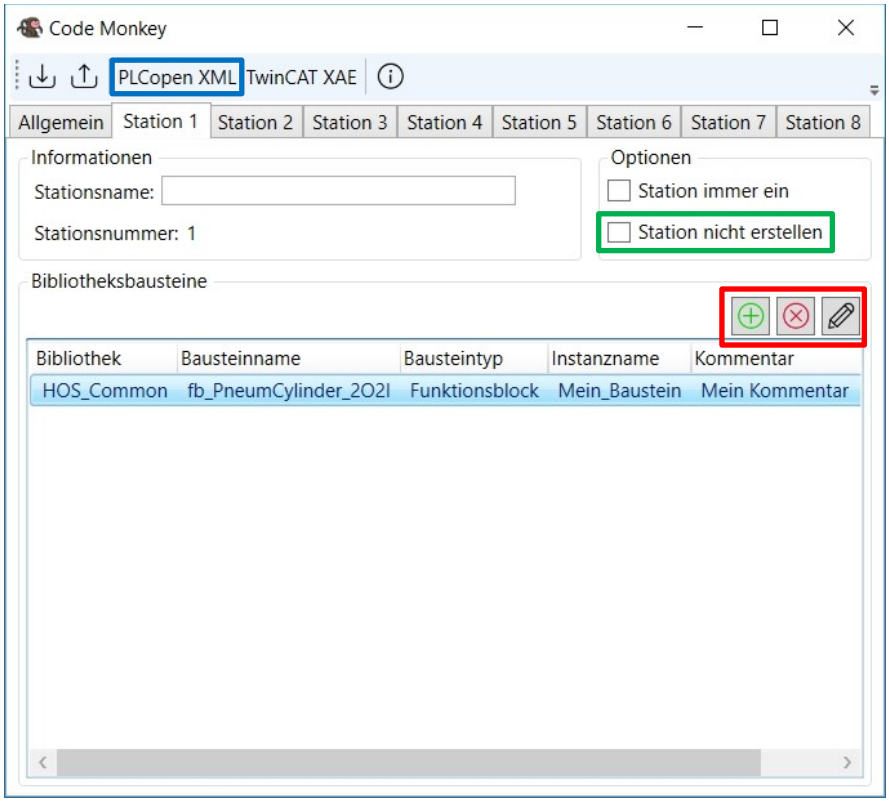


Abb. 9.6: Eine Station mit einem Bibliotheksbaustein, Quelle: Eigene Darstellung.

Eine nachträgliche Bearbeitung eines Bibliotheksbausteins findet über die Schaltfläche mit der Abbildung eines schwarzen Stifts statt (siehe Abb. 9.6 rot umrandet). Ein Klick auf das Steuerelement öffnet einen Dialog mit dem der Benutzer den Instanznamen oder den Kommentar des Bausteins anpassen kann (siehe Abb. 9.7).

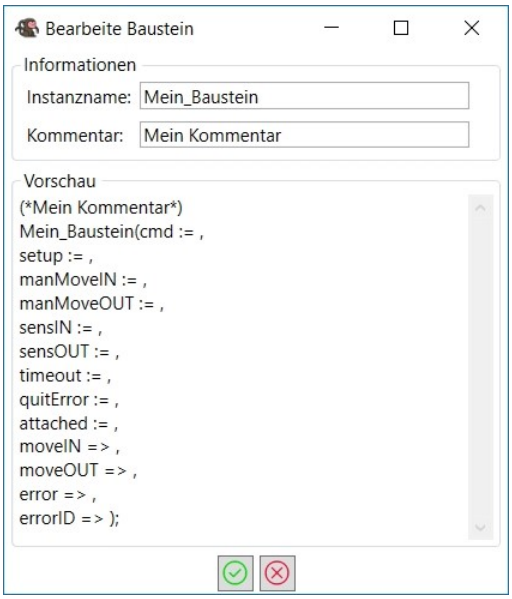


Abb. 9.7: Dialog zum Bearbeiten eines Bibliotheksbausteins in Code Monkey, Quelle: Eigene Darstellung.

Die maximale Anzahl der Stationen einer RTA wird über die Option *Station nicht erstellen* festgelegt (siehe Abb. 9.6 grün umrandet). Erfolgt die Auswahl z. B. bei der fünften Station, dann erstellt der Codegenerator eine PLCopen XML-Datei für eine Rundtaktanlage mit vier Stationen. Zum Erzeugen der PLCopen XML-Datei selbst dient die Schaltfläche *PLCopen XML* in der Menüleiste von Code Monkey (siehe Abb. 9.6 blau umrandet).

9.3 TwinCAT 3 XAE-Projekt erstellen

Eine PLCopen XML-Datei ist erst nach dem Import in ein TwinCAT 3 XAE-Projekt auf einer Beckhoff-Steuerung ausführbar (siehe Kapitel 8.3). Zum Erstellen eines TwinCAT 3 XAE-Projekts benötigt Code Monkey den Projektnamen und den Pfad zur XSLT-Datei für die Bibliotheksinformationen. Beide Parameter befinden sich im Reiter *Allgemein* (siehe Abb. 9.8). Da Beckhoff die Entwicklungsumgebung TwinCAT 3 in Microsofts Visual Studio integriert, muss der Codegenerator über die verwendete Visual Studio Version Bescheid wissen, um ein korrektes TwinCAT 3 XAE-Projekt zu erstellen. Die Schaltfläche *TwinCAT XAE* in der Menüleiste der Oberfläche startet den Generierungsvorgang (siehe Abb. 9.8 rot umrandet).

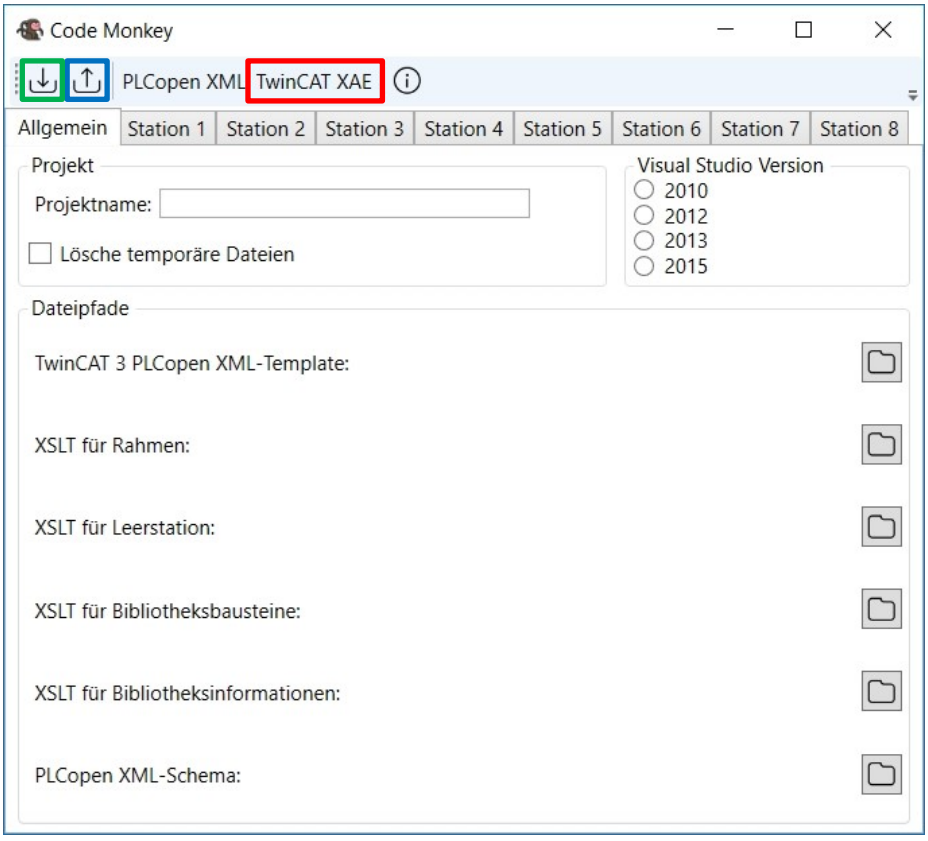


Abb. 9.8: Der Reiter *Allgemein* in Code Monkey, Quelle: Eigene Darstellung.

9.4 Code Monkey-Projekt speichern und laden

Ein Code Monkey-Projekt kann über die jeweiligen Steuerelemente gespeichert (siehe Abb. 9.8 grün umrandet) oder geladen (siehe Abb. 9.8 blau umrandet) werden. Die Sicherung der Benutzereingaben erfolgt als XML-Datei, deren Aufbau den Datenklassen aus Kapitel 8.2 entspricht. Aufgrund dieser Funktionalitäten steht es dem Bediener frei, an zeitlich verschobenen Zeitpunkten an einem Code Monkey-Projekt zu arbeiten oder auf bereits vorhandene Dateien zuzugreifen.

10 RESÜMEE UND AUSBLICK

Das Ziel der Masterarbeit einen Codegenerator für Rundtaktanlagen der Firma Schunk Hoffmann Carbon Technology zu entwickeln, ist als erfüllt anzusehen: Die Mitarbeiter der Maschinenbauabteilung können nun das Basisprogramm für eine neue Anlage in nur wenigen Schritten von der Applikation Code Monkey generieren lassen. Dazu muss einmalig eine Vorlage, welche das Grundgerüst einer RTA repräsentiert, mit der Entwicklungsumgebung TwinCAT 3 erstellt und als PLCopen XML-Datei exportiert werden. Der Generator bereitet die Vorlage für den Bediener auf, damit dieser das Template entsprechend seinen Vorstellungen konfiguriert. Aufgrund der frei wählbaren Benutzereingaben ist der Generator im Stande das Basisprogramm für eine Vielzahl an unterschiedlichen Rundtaktanlagen mit nur einer Vorlage zu erzeugen. Durch den Import der generierten PLCopen XML-Datei in das dazugehörige TwinCAT 3 XAE-Projekt entsteht ein ausführbares SPS-Programm (siehe Abb. 10.1).

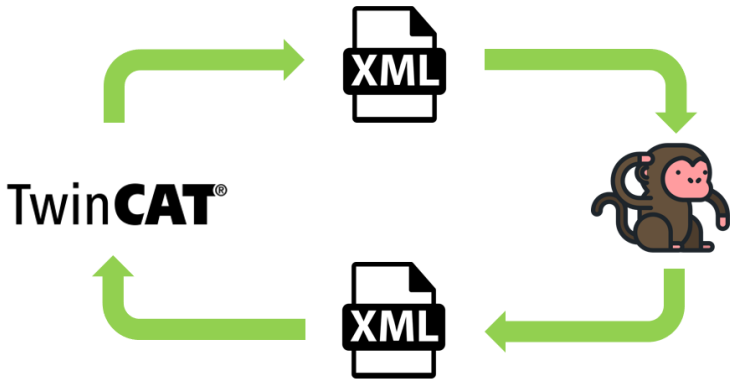


Abb. 10.1: Vereinfachte Darstellung des Codegenerierungsprozesses, Quelle: Eigene Darstellung.

Durch den überwiegenden Einsatz von XML-basierten Dokumenten und parametrisierbaren Templates bleibt Code Monkey offen für Änderungen sowie Anpassungen durch die Programmierabteilung von SHCT. Dieses Verhalten beruht auf dem modularen Aufbau des Codegenerators und ist Teil der Forderungen des Unternehmens. Der Einsatz eines Requirements-Engineerings erleichtert es, die Applikation entsprechend den Benutzeranforderungen zu entwickeln. Besonders die Dokumentation mehrerer Versionen samt ihres Status stellt sicher, dass die Software den Wünschen und Vorstellungen der Programmierabteilung entspricht. Durch die Ableitung der Use-Cases mit ihren potentiellen Fehlerquellen aus dem RE, lässt sich das Projekt in einzelne Arbeitspakete bzw. Elemente auftrennen und entsprechend testen.

Während der Entwicklungsphase des Codegenerators bewahrheitet sich besonders ein Nachteil der festgelegten Schichtenarchitektur: Die Schichten lassen sich oft nicht sauber trennen und müssen teilweise über mehrere Ebenen hinweg interagieren. Der Einsatz von WPF zur Visualisierung des Generators macht das anfänglich vorgesehene Modul *Verwalte Daten* überflüssig (siehe grüne Markierung in Abb. 10.2), da WPF eigene Funktionalitäten vorsieht, um eine Bindung zwischen der Benutzerfläche und den notwendigen Daten zu erzielen (als rot strichlierter Pfeil in Abb. 10.2 dargestellt). Dieses Vorgehen hat zur Folge, dass die oberste Ebene direkt auf die Schicht *Systemunterstützung* zugreift. Trotz dieser Schwierigkeiten ist die Schichtenarchitektur die richtige Wahl für den Aufbau von Code Monkey, da sie ein modulares Verhalten der Software unterstützt.

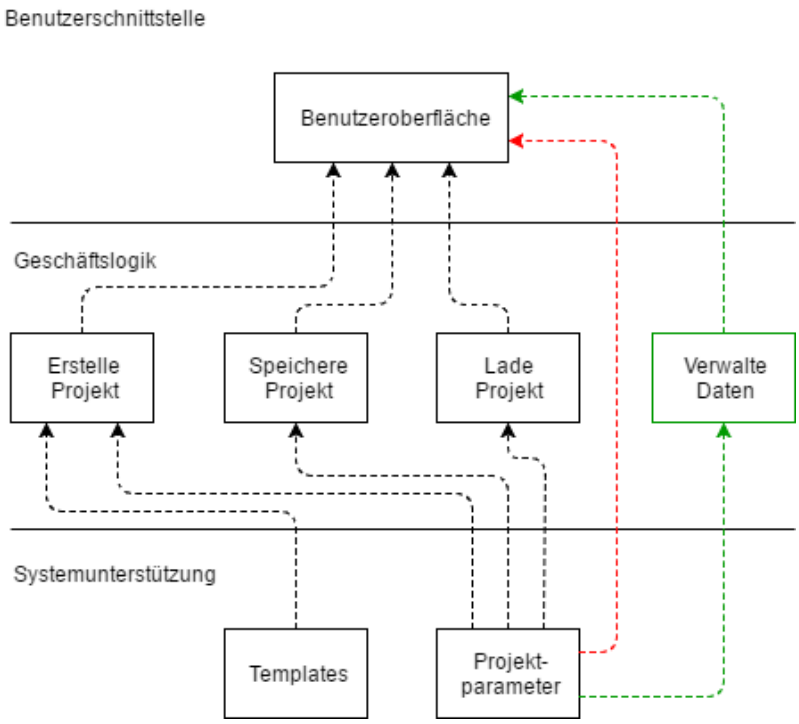


Abb. 10.2: Geplante (grüne Markierung) und finale (rote Markierung) Architektur des Codegenerators, Quelle: Eigene Darstellung.

Des Weiteren sind zwei Verbesserungsmöglichkeiten für den Codegenerator festzuhalten. Die Erste behandelt die Kommentare zum Hinzufügen und Ersetzen von Codeabschnitten (siehe Kapitel 8.2). Bei dem Kommentar zum Ersetzen von Programmteilen gibt der letzte Teil, beispielsweise *MSG* oder *INF*, Aufschluss über den auszutauschenden Bereich (siehe Abb. 10.3). Die dazugehörigen Informationen sind derzeit in den Ressourcen der WPF-Anwendung definiert. Damit sind sie durch einen Programmierer von SHCT änderbar, aber für jeden Kommentar ist ein eigener Eintrag notwendig. Die nächste Version von Code Monkey hat zum Ziel, einen allgemeinen Kommentar zum Ersetzen festzulegen und mit weiteren Symbolen im dazwischenliegenden Codeabschnitt den Generator auf die durchzuführende Aufgabe hinzuweisen, z. B. einen Ausdruck zu inkrementieren. Eine weitere Überlegung stellt die Auslagerung der Kommentar-definitionen in eine Konfigurationsdatei dar, um den Codegenerator flexibler zu gestalten.

```
IF ((*$ CM_ERSETZE_MSG $*)MODULES.MessageField_S1[i](*$ CM_ERSETZE_ENDE $*) <> 0) THEN
    tmpErrorCnt:=tmpErrorCnt+1;
END_IF
IF ((*$ CM_ERSETZE_INF $*)MODULES.InfoField_S1[i](*$ CM_ERSETZE_ENDE $*) <> 0) THEN
    tmpWarningCnt:=tmpWarningCnt+1;
END_IF
```

Abb. 10.3: Kommentare zum Ersetzen von Codeabschnitten, Quelle: Eigene Darstellung.

Die zweite Anmerkung betrifft die Ablaufdefinierung einer Station der Rundtaktanlage anhand von Petri-Netzen oder Automaten (siehe Kapitel 6). Eine entsprechende Diskussion zu diesem Thema mit der Programmierabteilung von SHCT hat zum Ergebnis, dass aufgrund des derzeitigen Aufbaus des SPS-Programms eine Implementierung dieser Überlegung nicht sinnvoll erscheint.

Nach dem das Programm zum größten Teil aus Funktionsbausteinen besteht und nicht aus Klassen mit entsprechenden Methoden, sind keine klar definierten Schnittstellen zum Ausführen von Aktionen vorhanden. Ein Funktionsbaustein kann zum Durchführen verschiedener Vorgänge eine unterschiedliche Anzahl von Parametern benötigen. Das Problem dabei: Ohne den inneren Aufbau des Bausteins zu kennen, lassen sich die für eine Aktion notwendigen Parameter nicht nach außen hin kennzeichnen. Im Gegensatz dazu fasst jede Methode die benötigten Elemente zusammen und macht somit das Wissen über die Vorgänge innerhalb der Methode bzw. der Klasse obsolet. Solange das SPS-Programm einer Rundtaktanlage nicht einem objektorientierten Aufbau entspricht, ist der Bediener dazu gezwungen, viele Einstellungen für den Ablauf im Codegenerator selbst vorzunehmen und somit lassen sich keine Vorteile gegenüber einer manuellen Ablaufdefinition erkennen. In der derzeitigen Ausführung von Code Monkey enthält die Leerstation das Grundgerüst eines Stationsablaufes und der Programmierer muss im Generat nachträglich die spezifischen Vorgänge einer Station festlegen.

Zusammengefasst verbessert Code Monkey durch die automatisierte Erstellung eines Basisprogramms für eine Rundtaktanlage den Entwicklungsprozess bei Schunk Hoffmann Carbon Technology. Das Kopieren bereits vorhandener Projekte und das manuelle Nachbesseren entfällt, da der Codegenerator diese monotonen und fehleranfälligen Arbeiten übernimmt. Die zusätzlich gewonnene Zeit steht den Mitarbeitern zur Verfügung, um sich auf kritische Herausforderungen an der Anlage zu konzentrieren oder lässt sich direkt zur Senkung der Projektkosten nutzen.

Eine Quantisierung der Einsparungen, wie sie aus den Masterarbeiten von Schweigler und Mastilovich hervorgehen (siehe Kapitel 4.2), ist für diese Arbeit nicht möglich. Der Grund dafür liegt im Aufbau des Codegenerators, welcher das Automation Interface von Beckhoff nutzt. TwinCAT 3 unterstützt die Schnittstelle im notwendigen Umfang – im Gegensatz zu seinem Vorgänger TwinCAT 2. Derzeit stellt das Unternehmen SHCT alle Rundtaktanlagen unter der Verwendung von TwinCAT 2 her und steigt erst nach Fertigstellung dieser Masterarbeit auf den Nachfolger um. Deshalb stehen keine Daten zu den Kosten bzw. den Entwicklungszeiten für eine Anlage mit TwinCAT 3 zur Verfügung und unterbinden damit einen Vergleich eines Projekts mit und ohne den Einsatz von Code Monkey. Schweigler erzielt mit Hilfe seines Generators eine Einsparung der Projektkosten um 12,26 % und Mastilovich reduziert die Ausgaben für die Erstellung der SPS-Anwendung um 12,95 %.

Ausgehend von den zu diesem Zeitpunkt gesammelten Erfahrungen mit dem Codegenerator durch die Programmierabteilung von SHCT liegt das Einsparungspotential von Code Monkey in derselben Größenordnung wie bei den Generatoren von Schweigler und Mastilovich. Als weitere Schritte für die Applikation sind die Umsetzung der genannten Verbesserungen für den Kommentar zum Ersetzen von Codeabschnitten und die Erweiterung der Software für den Einsatz von B&R-Steuerungen (Bernecker und Rainer) geplant.

LITERATURVERZEICHNIS

Gedruckte Werke (23)

Die SOPHISTen (Hrsg.) (2016): *Requirements-Engineering - Die kleine RE-Fibel*, 3. Auflage, SOPHIST, Nürnberg

Balzert, Heide (2011): *Lehrbuch der Objektmodellierung - Analyse und Entwurf mit der UML 2*, 2. Auflage, Spektrum Akademischer Verlag, Heidelberg

Beckhoff (2016): *Handbuch TwinCAT 3 - Automation Interface*, 1.2 Auflage, Beckhoff, o.O.

Gohert, Nadine (2014): *Automatische SPS-Codegenerierung für Syntheseverfahren der Supervisory Control Theory*, 1. Auflage, Hochschule für angewandte Wissenschaften Hamburg, Hamburg

Herrington, Jack (2003): *Code Generation In Action*, 1. Auflage, Manning Publications, Greenwich

Kecher, Christoph (2005): *UML 2.0 - Das umfassende Handbuch*, 1. Auflage, Galileo Press, Bonn

Kersken, Sascha (2013): *IT-Handbuch für Fachinformatiker - Der Ausbildungsbegleiter*, 6. Auflage, Galileo Press, Bonn

Lahres, Bernhard; Raýman, Gregor; Strich, Stefan (2016): *Objektorientierte Programmierung - Das umfassende Handbuch*, 3. Auflage, Rheinwerk, Bonn

Mastilovich, Nikola (2010): *Automatisation in programming of a PLC Code*, 1. Auflage, Massey University, Auckland

PLCopen (2009): *XML Formats for IEC 61131-3*, 2.01 Auflage, PLCopen, o.O.

Poetzsch-Heffter, Arnd (2009): *Konzepte objektorientierter Programmierung - Mit einer Einführung in Java*, 2. Auflage, Springer, Hagen

Rumpe, Bernhard (2004): *Modellierung mit UML - Sprache, Konzept und Methodik*, 1. Auflage, Springer, Berlin

Rumpe, Bernhard (2012): *Agile Modellierung mit UML - Codegenerierung, Testfälle, Refactoring*, 2. Auflage, Springer, Berlin

Rupp, Chris; Queins, Stefan; SOPHISTen, die (2012): *UML 2 glasklar - Praxiswissen für die UML-Modellierung*, 4. Auflage, Carl Hanser Verlag, München

Schindler, Martin (2012): *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*, 1. Auflage, Shaker, Dortmund

Schmitt, Karl (2011): *SPS-Programmierung mit ST: nach IEC 61131 mit CoDeSys und mit Hinweisen zu STEP 7 V11*, 1. Auflage, Vogel Business Media

Schweigler, Hannes (2012): *Automatisierte Erstellung von Softwarekomponenten für Speicherprogrammierbare Steuerungen*, 1. Auflage, Fachhochschule CAMPUS 02, Graz

Sommerville, Ian (2012): *Software Engineering*, 9. Auflage, Paerson, Hallbergmoos

Vogel-Heuser, Birgit; Wannagat, Andreas (2009): *Modulares Engineering und Wiederverwendung mit CoDeSys V3*, 1. Auflage, Oldenbourg Industrieverlag, München

Vonhoegen, Helmut (2015): *Einstieg in XML - Grundlagen, Praxis, Referenz*, 8. Auflage, Rheinwerk Computing, Bonn

Walter, Doberenz; Thomas, Gewinnus (2013): *Visual C# 2012 - Grundlagen und Profiwissen*, 1. Auflage, Carl Hanser Verlag, München

Wellenreuther, Günter; Zastrow, Dieter (2011): *Automatisieren mit SPS - Theorie und Praxis*, 5. Auflage, Vieweg + Teubner, Wiesbaden

Wirth, Niklaus (2011): *Grundlagen und Techniken des Compilerbaus*, 3. Auflage, Oldenbourg Verlag, München

Wissenschaftliche Artikel (2)

Hess, Dieter; Witsch, Daniel (2011): *SPS-Programmierung - die UML-Integration*, in: Computer & Automation, 12/2011, WEKA FACHMEDIEN, S. 40-43

Hofmann, Andreas; Menager, Nils; Schweig, Stephan; Mikelsons, Lars (2015): *Model-Based Engineering mit Industriesteuerungen*, in: VVD 2015 - Verarbeitungsmaschinen und Verpackungstechnik, Selbstverl. der Technischen Universität Dresden, Dresden, S. 251-273

Online-Quellen (16)

Google Scholar (o.J.): *Google Scholar*
https://scholar.google.at/citations?user=iQo_-kMAAAAJ&hl=de [Stand: 1.12.2017]

ITWissen (o.J.): *ITWissen*
https://www.itwissen.info/MDA-model-driven-architecture.html [Stand: 1.12.2017]

MathWorks (o.J.): *MATLAB Simulink PLC Coder*
https://de.mathworks.com/products/sl-plc-coder.html [Stand: 1.12.2017]

Microsoft (o.J.): *Microsoft Developer Network - Automation Object Model Chart*
https://msdn.microsoft.com/en-us/library/za2b25t3(VS.80).aspx [Stand: 1.12.2017]

Microsoft (2016): *Microsoft Developer Network - Code Generation and T4 Text Templates*
https://msdn.microsoft.com/de-de/library/bb126445.aspx [Stand: 1.12.2017]

Modeliosoft (o.J.): *Modelio SD C++*
https://www.modeliosoft.com/en/products/modelio-sd-cpp.html [Stand: 1.12.2017]

MVTec Software GmbH (o.J.): *MVTec*
http://www.mvtec.com [Stand: 1.12.2017]

PLCopen (o.J.): *PLCopen XML*
http://www.plcopen.org/pages/tc6_xml/xml_intro/index.htm [Stand: 1.12.2017]

Schunk Carbon Technology (o.J.): *Schunk Carbon Technology*
http://www.schunk-carbontechnology.com/de/produkte/electrical-carbon/ [Stand: 1.12.2017]

Schunk Hoffmann Carbon Technology AG (o.J.): *Schunk Hoffmann Carbon Technology AG*
http://www.hoffmann.at/de/hos/Impressum/schunk01.c.42295.de [Stand: 1.12.2017]

LeTourneau University (o.J.): *SPNBOX - A Toolbox for the Supervisory Control of Petri Nets*
http://www.letu.edu/people/marianiordache/abs/spnbox/ [Stand: 1.12.2017]

SPS Magazin (2016): *SPS Magazin*
http://www.sps-magazin.de/?inc=artikel/article_show&nr=112569 [Stand: 1.12.2017]

Altova (o.J.): *UModel 2017*
https://www.altova.com/de/umodel.html [Stand: 1.12.2017]

w3schools (o.J.): *w3schools*
https://www.w3schools.com/ [Stand: 1.12.2017]

Object Management Group (o.J.): *SysML*
http://www.omgsysml.org/ [Stand: 13.1.2018]

Moor, Thomas (2008): *DESTool*
http://www.rt.eei.uni-erlangen.de/FGdes/destool/ [Stand: 1.Dezember.2017]

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

ANHANG

Der Anhang zur Masterarbeit gliedert sich in vier Bestandteile auf:

1. Das Requirements-Engineering zum Codegenerator mit der Satzschablone der SOPHISTen (siehe Kapitel 7.2).
2. Die Use-Cases mit den potentiellen Fehlerquellen, welche sich aus dem Requirements-Engineering ableiten (siehe Kapitel 7.3).
3. Die Klassendiagramme aus der Entwicklungsphase des Codegenerators (siehe Kapitel 8.2.4).
4. Die Bedienungsanleitung zur Applikation Code Monkey (siehe Kapitel 9).