Hochschule Bremen

Bachelorarbeit

Exposé

Konzeption und Implementierung einer Makrosprache in C++

Autor:
Roland Jäger
360956

10. Februar 2016

Inhaltsverzeichnis

1 Einleitung

Die Einführung von Automatisierung in ein Softwaresystem ist vergleichbar mit den Maschinen, die in der industrielle Revolution auftauchten. Anstelle, dass Menschen arbeiten müssen, um ein gewünschtes Ergebnis zu bekommen, drücken sie auf einen Knopf, und ein anderes System nimmt ihnen die aufwändige Arbeit ab. Dies führt dazu, dass Produkte schneller, mit weniger Arbeitsaufwand erstellt werden können. Zudem ist die entstehende Qualität immer auf einem gleichbleibenden Level und hängt nicht von dem Befinden der Arbeiter ab.

Die P3-group arbeitet mit Airbus, um Lösungen für den Flugzeugbau zu entwickeln. Dieser Markt ist hart umkämpft, wodurch minimale Gewinne einen großen Unterschied machen können. Ein Feld, welches seit Jahren immer weiter durch wissenschaftliche und technische Durchbrüche optimiert wird, sind die menschlichen Ressourcen. Automatisierung sorgt dafür, dass sich wiederholende Arbeitsabläufe – aus der Sicht des Nutzers – zu einem einzigen Schritt werden und so Zeit sparen.

Makros sind die Fließbänder der digitalen Welt, und diese Arbeit beschäftigt sich mit der Entwicklung eines Makro Systems bzw. Sprache.

Makros werden durch die Verbindung kleinerer Bausteine erstellt. Diese können andere Makros oder Befehle, die die Anwendungsumgebung bereitstellt, sein. Dies ist mit einem Fließband in der Autoindustrie zu vergleichen. Jede Station ist genau für eine Aufgabe zuständig und kümmert sich um nichts anderes.

Die Makrosprache ist ein Baukasten, mit dem Makros erstellt – "Fließbänder" für spezielle Aufgaben erzeugt – werden können.

2 Problemfeld

Softwaresysteme haben oft das Problem, dass sie mit einigen zentralen Features anfangen, die fest definiert werden (sollten), bevor ein Vertrag geschlossen wird. Für weitere Funktionalität, die über die Vereinbarungen im Vertrag hinausgehen, muss der Vertrag erweitert werden. Wenn der Vertrag erfüllt ist, und im Anschluss weitere Wünsche aufkommen, muss ein weiterer Vertrag aufgesetzt werden und die vorher gelieferte Software muss angepasst, gegebenenfalls erweitert werden. Dies kann zur Folge haben, dass große Teile der Software umgeschrieben werden müssen, oder sogar, dass die Architektur der gesamten Anwendung verändert werden muss.

Wenn frühzeitig ein Makrosystem/-sprache und ein entsprechendes Erweiterungskonzept für Module bzw. Plugins eingeführt wird, ist die Wahrscheinlichkeit, dass der Kern der Applikation angefasst werden muss, wesentlich geringer. Durch diese Kombination kann einfach ein weiteres Modul geladen werden, welches die neuen Grundbausteine der Applikation hinzufügt. Diese können dann in einem neuen, oder angepassten Makro

genutzt werden, um den Wunsch der Kunden zu erfüllen. Im Falle, dass es keiner neuen Grundbausteine bedarf, reicht es sogar, nur ein Makro zu liefern. Die Vorteile dieser Methode sind, dass – wenn man davon ausgeht, dass die benutzen Makros und Grundbausteine fehlerfrei durch ausreichendes Testen der Software sind – keine neuen Bugs in den Kern der Software eingeführt werden können und somit immens zu der Stabilität der Software beigetragen wird. Ein weiterer Vorteil ist, dass die Makros mit wesentlich weniger Aufwand entwickelt werden können, weil sie sich auf einem höheren Level befinden. Für Kunden ist eine nutzbare Makrosprache auch interessant, weil sie zum Teil, durch das hausinterne Personal Anforderungen an die Software realisieren können, ohne den langen Weg über eine Firma zu gehen. Dies bedeutet auch, dass die Software eine bessere Chance hat, die Zeit zu überdauern.

2.1 Anwendung in der Wirtschaft

Die P3-group ist daran interessiert, dass sie ihren Kunden Lösungen schnell und in hoher Qualität anbieten kann. Um dies zu erreichen arbeiten sie daran, dass alle Softwaresysteme, die von ihnen angeboten werden, Automatisierung über Makros unterstützen. Wirtschaftlich rentieren sich die Makros dadurch, dass sie von den Firmen gemietet und nicht nur einmal verkauft werden. Zum Beispiel, werden alle Flugzeugteile ausgewählt und dann deren Gewicht ermittelt. Ein anderes Mal sollen nur spezielle Teile aus einem bestimmten Werkstoff zusammen gezählt und deren Preis ermittelt werden. Anstelle, dass hier eine sehr komplexe Suchfunktion entwickelt wurde, können hier zwei Makros zum Einsatz kommen, die jeweils eine Aufgabe erfüllen und somit für den Benutzer sicher zu handhaben sind. Die Komplexität der Software wurde durch das zweite Makro nicht sonderlich beeinflusst, da dieses intern auf Funktionalität zurückgreifen kann, die schon vom ersten genutzt wird.

2.2 Anforderungen an Makros

Die Herausforderung, ein Makrosystem/-sprache in C++ zu implementieren fängt dann an, wenn man von den Makros will, dass diese nicht nur hintereinander abgearbeitet werden, ohne dass sie wissen, dass andere Makros vor ihnen bzw. nach ihnen ausgeführt werden – wie in Abbildung 1 zu sehen ist.

Abbildung 1: Sequenzielles Abarbeiten von Prozessschritten.

Abbildung 2 zeigt den ersten Schritt zu einer Herausforderung – und zu einer nützlichen Implementation – Logik. Hierbei bietet man an, dass der Makro-Entwickler durch Rückgabewerte aus Makros entscheiden kann, welche weiteren Makros er ausführen möchte.

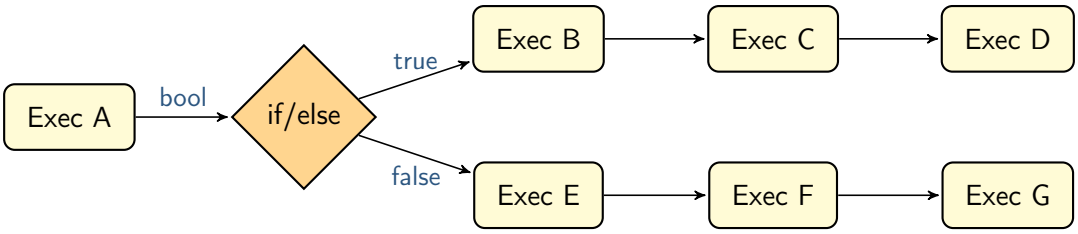

Abbildung 2: Logische Ausdrücke um bedingte Anweisungen zuzulassen.

Obwohl man mit solchen Makros schon einige Probleme lösen kann, ist es nicht das, was man zur Verfügung haben will, wenn man mit objektorientierten Sprachen arbeitet bzw. mehr als ein *'ja'* oder nein *'nein'* braucht.

Was ein Makrosystem/-sprache anbieten muss ist, dass Instanzen von verschiedenen Klassen/Typen zurückgegeben und beliebig viele Parameter (unterschiedlicher Klassen/Typen) dem Makro mitgegeben werden können. Leider sind gerade diese Punkte ein Problem in C++, weil C++ keine Reflexion[5, 2] unterstützt. Dies ist – da es die Zeit, die für die Bachelorarbeit bereitsteht, mehrfach sprengt – schon durch eine Implementation des Command-Patterns[1] von mir gelöst worden. Somit kommt in diesem Schritt 'nur' noch hinzu, dass es Schleifen geben kann, siehe Abbildung 3, und Rückgabewerte in Variablen speicherbar sind.

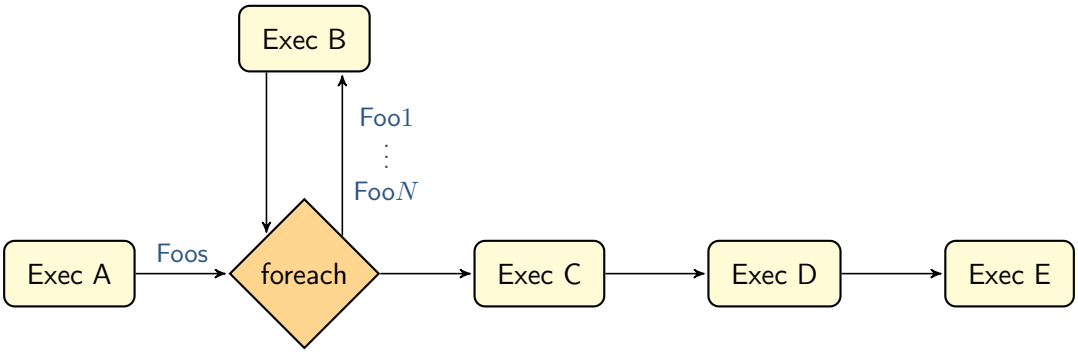

Abbildung 3: Schleife, die Anweisungen für ein Element aus der Liste aufrufen.

Letztendlich kann man also sagen, dass ein solches Makrosystem/-sprache eine Programmiersprache mit Interpreter[3, S.274] ist, deren Laufzeitumgebung eine anderes Softwaresystem ist.

[1] Das Command-Pattern wurde mit Hilfe des `any`[2] sowie `optional`[3] Types erstellt.
[2] `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3804.html`
[3] `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3793.html`

3 Lösungsansatz

Makros setzen auf dem Command-Pattern[3, S.263] auf und führen durch ihre Modularität und dynamischer Natur[4] unter anderem dazu, dass Endbenutzer das Verhalten und sowie Potenzial der Software beeinflussen können.

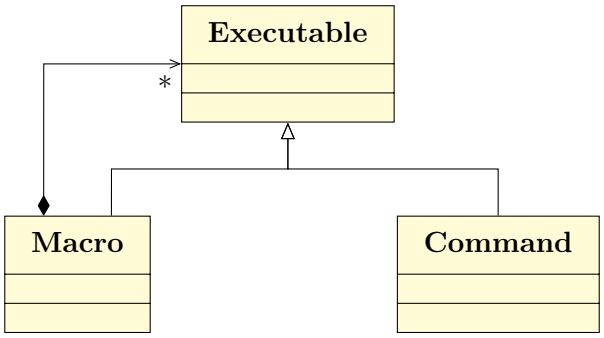

Abbildung 4: Das UML Diagramm zeigt die Beziehungen zwischen den Elementen aus den vorangegangenen Kapiteln.

In Abbildung 4 ist die Komposition[3, S.183] der Makros zu sehen – Erben von `Command` sind die *Grundbausteine* aus dem vorherigen Text, Instanzen von `Macro` sind *Makros*, die `Executable`s ausführen können, also andere Makros oder Grundbausteine.

Bevor die genannten Probleme gelöst werden können, muss recherchiert werden, was schon bei ähnlichen Problemen entwickelt wurde, und eine Anforderungsanalyse durchgeführt werden. Auf diesen Ergebnissen wird dann eine Basisarchitektur entwickelt, die als Grundlage für die Implementierung dienen wird.

3.1 Grober Lösungsansatz

Der Makroansatz besteht aus inkrementell kompletteren Lösungen. Diese Lösungen sind in drei Level aufteilbar.

Das erste Level beschäftigt sich mit hauptsächlich mit dem Parser/Tokenizer für den Interpreter und soll Makros ausschließlich sequenziell ausführen können. Das zweite Level befasst sich mit der Erweiterung des Interpreters sowie der Implementierung von Argumenten und Rückgabewerten. Das dritte und letzte Level beschäftigt sich damit, die Implementierungen von Makros und Interpreter weiter auszunutzen, um Schleifen von `while` bis `for-each` zu implementieren.

[4] Dynamische Programmiersprachen werden erst zur Laufzeit interpretiert – ein prominentes Beispiel wäre JavaScript.

3.2 Feiner Lösungsansatz

Der Parser für das erste Level bedarf einer Syntax, die er parsen kann. Diese muss für die späteren Iterationen ausreichend flexibel sein. Höchstwahrscheinlich wird der Parser den Makrostring in einen abstrakten Syntaxbaum (AST)[5] überführen. Der Interpreter muss in dieser Iteration nicht viel mehr können, als Makro nach Makro aufzurufen.

Im zweiten Level wird der Interpreter durch die Einführung der bedingten Anweisungen sowie von Argumenten und Rückgabewerten der Makros erweitert. Die Änderungen an dem Parser sollten hier nur noch minimal sein, weil dieser nur weitere Tokens erkennen und in den AST überführen muss.

Die Schleifen sowie Variablendeklarationen bauen auf den vorherigen Stufen auf und sollten bei einer (wider Erwarteten) perfekten Ausführung derer, mit wenig Aufwand zu implementieren sein. Sollte allerdings bei der Planung bzw. der Architektur ein Detail übersehen worden sein, kann dies zu größeren Änderungen führen.

4 Konkrete Aufgaben

Die in der Bachelor-Arbeit durchzuführenden Aufgaben sind:

- Anforderungsanalyse:
  - Was muss die Makrosystem/-sprache mindestens leisten, um von Nutzen zu sein?
  - Machbarkeitsanalyse – Was gibt C++ her?
- Basisarchitektur für alle Komponenten, die mit der Arbeit zu tun haben.
  - AST
  - Interpreter
  - Makros
  - Parser
  - Tokenizer
- Entwicklung eines Syntax für die Makrosystem/-sprache.
- Eine Referenzimplementierung von der Makrosystem/-sprache.
- Evaluation der erreichten Ergebnisse.
- Die Ergebnisse der Arbeit niederschreiben.

[5] *Abstract syntax tree* ist eine digitale Darstellung einer Programmiersprache.

5 Arbeitsumfeld

5.1 Software

- Clang[6] als C++11/14 Compiler
  - Address Sanitizer[7] um folgende Fehler zu finden:
    - "Out-of-bounds accesses to heap, stack and globals"
    - "Use-after-free"
    - "Use-after-return (to some extent)"
    - "Double-free, invalid free"
    - "Memory leaks (experimental)"
  - Thread Sanitizer[8] um folgende Fehler zu finden:
    - data races
    - mutex lock Reihenfolge (potenzielle deadlocks)
- Git[9] als Versionsverwaltung
- CMake[10] als Build-System-Generator
- Latex[11] für Text der kein Code ist und Vektorgrafiken
- Sublime Text 3[12] mit Plugins von der Package Control[13] als Editor/IDE
- Arch[14] und Ubuntu[15] als Betriebssystem
- Inkscape[16] und Visual Paradigm[17] für Grafiken
- P3-group interne Software als Arbeitsgrundlage

[6] http://clang.llvm.org/
[7] http://clang.llvm.org/docs/AddressSanitizer.html
[8] http://clang.llvm.org/docs/ThreadSanitizer.html
[9] http://git-scm.com/
[10] https://cmake.org/
[11] http://www.latex-project.org/
[12] http://www.sublimetext.com/3
[13] https://packagecontrol.io/
[14] https://www.archlinux.org/
[15] http://www.ubuntu.com/
[16] https://inkscape.org/en/
[17] http://www.visual-paradigm.com/

5.2 Literatur

[1] *cppreference.com*. 2015. URL: http://en.cppreference.com/w/cpp (besucht am 29. 12. 2015).

[2] Jacques Ferber. „Computational reflection in class based object-oriented languages“. In: *ACM Sigplan Notices*. Bd. 24. 10. ACM. 1989, S. 317–326.

[3] Erich Gamma u. a. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0201633612.

[4] *The C++ Standards Committee*. 2016. URL: http://www.open-std.org/JTC1/SC22/WG21/ (besucht am 17. 01. 2016).

[5] Steve Vinoski. „A time for reflection“. In: *Internet Computing, IEEE* 9.1 (2005), S. 86–89.

6 Planung

6.1 Wann

März bis Juni 2016

6.2 Wo

P3 engineering GmbH
Flughafenallee 26/28
28199 Bremen
www.p3-group.com

6.3 Arbeitspakete

- Recherche (ca. $\frac{1}{2}$ Woche)
- Anforderungsanalyse (ca. $\frac{1}{2}$ Woche)
- Konzeption (ca.1 Woche)
  - Level 1 – Abarbeiten von `Macros`
  - Level 2 – Logik / primitive Rückgabewerte (ca. $\frac{1}{2}$ Woche)
  - Level 3 – Komplexe Rückgabewerte (ca. $\frac{1}{2}$ Woche)

- Implementierung (ca.3 Wochen)
  - Level 1 – Abarbeiten von `Macros` (ca.$\frac{1}{2}$ Woche)
  - Level 2 – Logik / primitive Rückgabewerte (ca.1 Woche)
  - Level 3 – Komplexe Rückgabewerte (ca.$1\frac{1}{2}$ Wochen)
- Dokumentation (ca.$\frac{1}{2}$ Woche)
- Verfassen der Bachelor-Thesis (ca.3 Wochen)
  - Einleitung + Anforderungsanalyse (ca.1 Woche, ab 14. März)
    - Allgemeines
    - Kapitel 1: Einleitung
    - Kapitel 2: Anforderungsanalyse
  - Hauptteil (ca.$1\frac{1}{2}$ Wochen, ab 11. April)
    - Kapitel 3: Konzeption
    - Kapitel 4: Exemplarische Realisation
  - Schlussteil (ca.$\frac{1}{2}$ Woche, ab 25. April)
    - Kapitel 5: Evaluation
    - Kapitel 6: Zusammenfassung und Ausblick

6.4 Meilensteine

	Abschluss	Beginn
1. März:		Recherche & Anforderungsanalyse
7. März:	Recherche & Anforderungsanalyse	Konzeption
14. März:	Konzeption	Implementierung & Erster schriftlicher Teil
11. April:	Implementierung & Erster schriftlicher Teil	Dokumentation & Zweiter schriftlicher Teil
25. April:	Dokumentation & Zweiter schriftlicher Teil	Dritter schriftlicher Teil
28. April:	Dritter schriftlicher Teil	Korrektur, Binden der DA etc.
3. Mai:	Abgabe der Arbeit	

7 Gliederung der Arbeit

Allgemeines

Eidesstattliche Erklärung

Danksagung

Kapitel 1: Einleitung

1.1. Problemfeld

1.2. Ziele der Arbeit

1.3. Hintergründe und Entstehung des Themas

1.4. Struktur der Arbeit, wesentliche Inhalte der Kapitel

Kapitel 2: Anforderungsanalyse

2.1. Diskussion des Problemfeldes

2.2. Anforderungen an die angestrebte Lösung

Kapitel 3: Konzeption

3.1. Syntax

3.2. Level 1 – Grundarchitektur

3.3. Level 2 – Logik / primitive Rückgabewerte

3.4. Level 3 – Komplexe Rückgabewerte

Kapitel 4: Exemplarische Realisation

4.1. Tokenizer

4.2. Abstrakter Syntaxbaum

4.3. Parser

4.4. Interpreter

4.5. Makro

Kapitel 5: Evaluation

Kapitel 6: Zusammenfassung und Ausblick

Anhänge

8 Personen

8.1 Erster Gutachter

Name: Prof. Dr. Thorsten Teschke
E-Mail: Thorsten.Teschke@hs-bremen.de

8.2 Zweiter Gutachter

Name: Prof. Dr. Martin Hering-Bertram
E-Mail: Martin.Hering-Bertram@hs-bremen.de

8.3 Ansprechpartner

Name: Mirko Wiechmann
E-Mail: Mirko.Wiechmann@p3-group.com
Tel.: +49 421 55 83 64 300

8.4 Student

Name: Roland Jäger
Matrikelnr.: 360956
E-Mail: Roland@Wolfgang-Jaeger.de
Tel.: +49 163 636 43 02

9 Unterschriften

Ort	Datum	Prof. Dr. Thorsten Teschke
Ort	Datum	Roland Jäger