Konzeption und Implementierung einer Makrosprache in C++

Roland Jäger
360 956

Hochschule Bremen

16. Mai 2016

Konzeption und Implementierung einer Makrosprache in C++

Roland Jäger
360 956

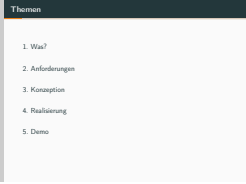
Hochschule Bremen

16. Mai 2016

1. Moin, ihr kennt alle meinen Namen und ...
2. Da wir nur wenig Zeit haben, muss ich euch in den nächsten 20 min 15.000 Zeilen Code erklären. Das sind 750 Zeilen pro Minute ...

Themen

1. Was?
2. Anforderungen
3. Konzeption
4. Realisierung
5. Demo



1. Da ich das allerdings keinem zumuten kann, werde ich in den folgenden Folien meine Arbeit sehr abstrakt erklären. Und in der Demo nur ein wenig auf den Code eingehen. Wodurch euch C++ Code größten Teils erspart bleibt.
2. Wer im Anschluss fragen zum Code hat – ich werde gerne Fragen beantworten (und wahrscheinlich länger über den Code reden, als euch recht ist).
3. Ich habe nichts dagegen, wenn während der Präsentation Fragen gestellt werden – für allgemeine Fragen bietet es sich an diese vor der Demo zu stellen.

Was?

Was?

- Fangen wir mit einer einfachen Frage an: ‘Was?’. . . .

Was ist eine Makrosprache?

Was ist eine Makrosprache?

1. Was ist eine Makrosprache?

Was ist eine Makrosprache?

- Eine Programmiersprache

Was ist eine Makrosprache?
• Eine Programmiersprache

1. Eine Makrosprache ist mit euch bekannten Programmiersprachen zu vergleichen.
2. Makros werden genutzt, um eintönige Arbeitsabläufe zu automatisieren.
3. Makros führen also Befehle aus, die der Nutzer hätte ausführen können. Somit ist eine Makrosprache eine der höchsten (abstraktesten) Programmiersprachen.
4. Zu vergleichen ist das am ehesten mit den Sprachen, die eine Virtuelle Maschine nutzen. Im Fall der Makrosprache ist die Anwendung die Virtuelle Maschine. Es sollte also kein Unterschied machen, ob ein Makro auf Windows geschrieben wurde und dann auf Linux oder XOS ausgeführt wird.

Was ist eine Makrosprache?

- Eine Programmiersprache
- Eine Möglichkeit statische Systeme zu dynamischen umzuwandeln

Was ist eine Makrosprache?
- Eine Programmiersprache
- Eine Möglichkeit statische Systeme zu dynamischen umzuwandeln

1. Makros sind dynamisch.
2. Makros sind Strings – also eine Reihenfolge von Zeichen – die erst zur Laufzeit der Anwendung erstellt werden. Diese werden entweder von dem Benutzer erzeugt oder z.B. von einer Datei gelesen. Das bedeutet, dass die Strings nicht zur Zeit der Entwicklung bzw. der Kompilierung oder der Auslieferung des Softwaresystems bekannt sein müssen.
3. Makros bieten den Nutzer an, eine Anwendung um Funktionen zu erweitern, ohne das die Anwendung erweitert werden muss.

Was ist eine Makrosprache?

- Eine Programmiersprache
- Eine Möglichkeit statische Systeme zu dynamischen umzuwandeln
- Kein Script

Was ist eine Makrosprache?
- Eine Programmiersprache
- Eine Möglichkeit statische Systeme zu dynamischen umzuwandeln
- Kein Script

1. Eine Makrosprache soll keine Scriptsprache sein. Zwar kann sich eine Makrosprache wie ein Script anfühlen, sollte aber keine sein.
2. Der Unterschied von Makros und Scripts liegt vor allem darin, dass Scripts komplett neue Funktionalitäten erzeugen und Makros genutzt werden um vorhandene Funktionen zu automatisieren. – Also ein Subset von Script darstellen.
3. Es ist ein kleiner Unterschied – der allerdings große Folgen hat. Eine Scriptsprache sollte eine wesentlich größere API anbieten, z.B. call-backs und threads support. Diese beiden würden Scripts ermöglichen, die nicht durch den Benutzer direkt angestoßen werden müssen und asynchron ihre Funktionen ausführen – z.B. eine automatische Sicherung nach 5 Minuten.

Was war die Gegebenheiten?

1. Was war gegeben?

Was war die Gegebenheiten?

- C++

Was war die Gegebenheiten?
• C++

1. Die Anwendung, für die diese Makrosprache entwickelt wurde, ist in C++ geschrieben.
2. Das hat den Vorteil, dass die Anwendung das Potenzial hat extrem effizient und schnell zu arbeiten.
3. Ein Nachteil ist, dass C++ eine statische Sprache ist – also nach dem Kompilieren nicht verändert werden kann was ausgeführt werden soll.
4. Wäre sie in z.B. Python geschrieben, wären Makros nur ein weiteres Python Script welches geladen und abgearbeitet werden würde. (Was ziemlich einfach wäre zu implementieren.)

Was war die Gegebenheiten?

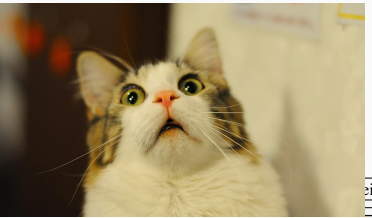
- C++
- Command-Pattern

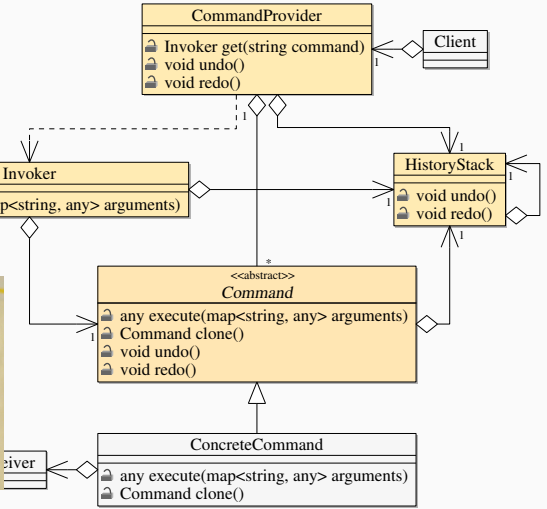
Was war die Gegebenheiten?
• C++
• Command-Pattern

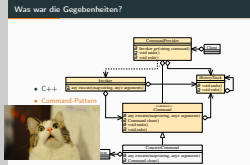
1. Ein zentrales Element ist eine implementation des CommandPatterns.
2. Das Command-Pattern der Anwendung, ist die Schnittstelle, die die Makrosprache nutzt, um mit der Anwendung zu interagieren.

Was war die Gegebenheiten?

- C++
- Command-Pattern



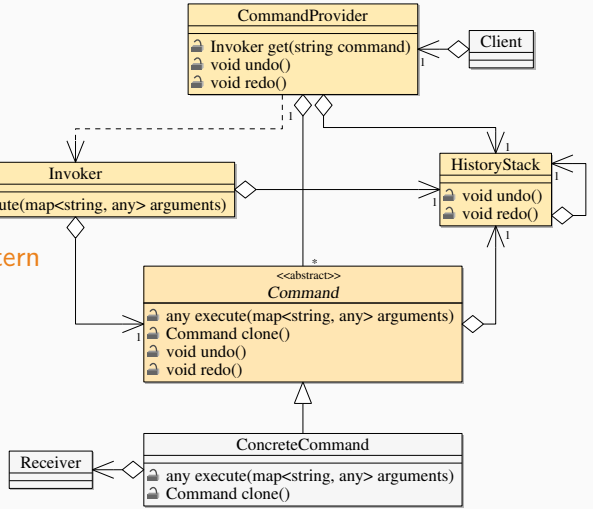


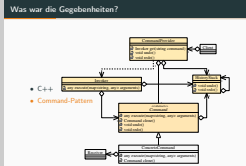


1. Nur weil wir die 750 Zeilen pro Minute sein lassen, heißt es nicht das es einfach wird!
2. Ich werde euch die wichtigsten Elemente präsentieren, die bei der Entwicklung der Makrosprache, bzw. Programmiersprache, durchlaufen wurden.
3. Dazu zählt auch dieses UMl Diagramm, welches eine wichtige Grundlage ist. – Es sieht schlimmer aus als es ist. . .

Was war die Gegebenheiten?

- C++
- Command-Pattern





1. Der **Invoker** wird von dem **CommandProvider** an den **Client** gegeben, der den nach nach einem **Command** gefragt hat.
2. Der **Client** kann z.B ein Knopf in einer GUI sein.
3. Der **HistoryStack** sorgt dafür, dass **Command**s rückgängig gemacht bzw. wiederhergestellt werden können.
4. Das **Command** ist die Elternklasse von dem **ConcreteCommand** welche ihre Funktionalität in der **execute** Methode implementiert.
5. Wichtig zu beachten ist die **map<string, any>** von dem **Command** diese erlaubt es beliebig viele Daten, mit beliebigen Datentypen, typsicher als Parameter zu übergeben – ohne die Methodensignatur zu verändern. Der any Typ wird in C++17 enthalten sein.

Anforderungen

1. Anforderungen an die Lösung...

Anforderungen

- Dynamisches Typesystem

Anforderungen
• Dynamisches Typesystem

1. Die Makros sollten davon profitieren, dass die Commands Rückgabewerte und Parameter haben. Das heißt, dass es eine Möglichkeit geben muss, dass “unbekannte” Datentypen zwischengespeichert werden.
2. Es soll also möglich sein so etwas zu schreiben . . .

Anforderungen

- Dynamisches Typesystem `var a = 10; a = "10";`

1. Hier wird der Variable a beim ersten Mal ein Integer zugewiesen und beim zweiten Mal ein String.
2. Die Lösung für das zwischenspeichern ist der any Typ, der von dem Command-Pattern genutzt wird.

Anforderungen

- Dynamisches Typesystem `var a = 10; a = "10";`
- Kontrollfluss

1. Makros sollten eine Möglichkeit haben, auf den zustand der Anwendung zu reagieren.
   Ein Beispiel wäre ...

Anforderungen

- Dynamisches Typesystem `var a = 10; a = "10";`
- Kontrollfluss `if(has_unsaved()) { save(); }`

1. Das ein Makro nur probiert zu speichern, wenn es etwas zum speichern gibt.
2. Generell sollte speichern schnell gehen, allerdings kann es passieren, dass eine mehrere Gigabyte große Datei geschrieben werden muss, was dem Nutzer nicht zumuten ist.

Anforderungen

- Dynamisches Typesystem `var a = 10; a = "10";`
- Kontrollfluss `if(has_unsaved()) { save(); }`
- Benutzerfreundlichkeit

Anforderungen
• Dynamisches Typesystem var a = 10; a = "10";
• Kontrollfluss if(has_unsaved()) { save(); }
• Benutzerfreundlichkeit

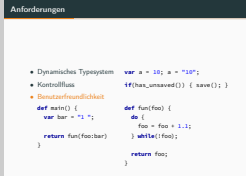
1. Die Makrosprache soll für den Benutzer leicht zu verstehen sein ...

Anforderungen

- Dynamisches Typesystem
- Kontrollfluss
- Benutzerfreundlichkeit

```
def main() {
  var bar = "1 ";

  return fun(foo:bar)
}
```

```
var a = 10; a = "10";

if(has_unsaved()) { save(); }

def fun(foo) {
  do {
    foo = foo + 1.1;
  } while(!foo);

  return foo;
}
```

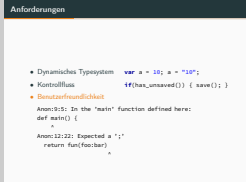



1. Der Syntax sollte also deutlich machen, was passieren wird.
2. Durch **def** wird automatisch klar, das eine Funktion definiert wird und **var** definiert eine Variable.
3. Da die Implementation des CommandPatterns eine Map nutzt, um die Command Parameter anzugeben, habe ich mich entschieden named parameter zu nutzen, um das überladen von Funktionen anhand der Parameternamen zu ermöglichen. (**foo:bar**)
4. Aber in meine Augen ist Syntax nur ein kleiner Punkt...

Anforderungen

- Dynamisches Typesystem `var a = 10; a = "10";`
- Kontrollfluss `if(has_unsaved()) { save(); }`
- Benutzerfreundlichkeit

```
Anon:9:5: In the 'main' function defined here:
def main() {
    ^
Anon:12:22: Expected a ';'
  return fun(foo:bar)
                     ^
```

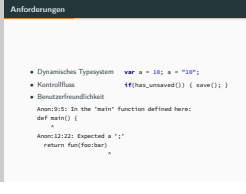



1. Wenn der Nutzer einen Fehler macht, soll dieser möglichst einfach zu finden sein – man will mit einem Tool arbeiten, nicht dagegen kämpfen.
2. Die Fehlermeldungen müssen dem Nutzer nicht nur sagen was falsch ist, sondern auch wo und weshalb.
3. In dem Beispiel wurde ein Semikolon in der main Methode vergessen, was die beiden Fehlermeldungen schnell deutlich machen.

Anforderungen

- Dynamisches Typesystem `var a = 10; a = "10";`
- Kontrollfluss `if(has_unsaved()) { save(); }`
- Benutzerfreundlichkeit

```
Anon:9:5: In the 'main' function defined here:
def main() {
    ^
Anon:12:22: Expected a ';'
  return fun(foo:bar)
                    ^
```

1. Neben der Wartbarkeit war auch die Macht dieser Makrosprache wichtig.
2. Wie der Syntax gezeigt hat, geht die Sprache in die Richtung von Scripts, man kann Funktionen anlegen, primitive Datentypen nutzen und hat Kontrollstrukturen zur Verfügung.
3. Da die Makros letztendlich nur auf Commands arbeiten können, ist das Gefahrenpotenzial – was von unwissenden Nutzern ausgeht – größtenteils eingeschränkt.

Konzeption

1. Konzeption
2. Die folgende Liste zeigt nicht nur, was entwickelt werden musste, sondern bildet auch den Ablauf beim Ausführen eines Makros ab.

Was wurde entwickelt?

- Tokenizer

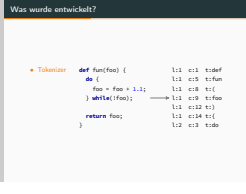
1. Ein Tokenizer.
2. Der Tokenizer muss den String / das Makro zu einer Liste von Tokens verwandeln ...

Was wurde entwickelt?
• Tokenizer

Was wurde entwickelt?

- Tokenizer

```
def fun(foo) {                  l:1  c:1  t:def
  do {                          l:1  c:5  t:fun
    foo = foo + 1.1;            l:1  c:8  t:(
  } while(!foo);        ⟶      l:1  c:9  t:foo
                                l:1  c:12 t:)
  return foo;                   l:1  c:14 t:{
}                               l:2  c:3  t:do
```

1. Wichtig ist, dass bei dem erstellen der Liste, keine Informationen verloren gehen.
2. Die Reihenfolge wird durch das Tokenizen zwar nicht verändert, allerdings gehen überflüssige Leerzeichen und Zeilenumbrüche verloren.
3. Diese sind für Fehlermeldungen extrem wichtig und müssen für die Tokens mitgespeichert werden.

Was wurde entwickelt?

- Tokenizer
- Abstrakter Syntaxbaum

Was wurde entwickelt?
- Tokenizer
- Abstrakter Syntaxbaum

1. Klassen die einen abstrakten Syntaxbaum bilden.
2. Ein Abstrakter Syntaxbaum beschreibt den Syntax aus dem Makro, ohne Informationsverlust. ASTs werden genutzt, da es einfacher ist mit ihnen als Datenstruktur zu arbeiten, als mit einem String.
3. Die meisten Elemente des Syntaxes lassen sich als Klasseninstanzen in dem AST wiederfinden. Prominente Ausnahmen sind Klammern, Semikolons und Leerzeichen.

Was wurde entwickelt?

- Tokenizer
- AST
- Parser

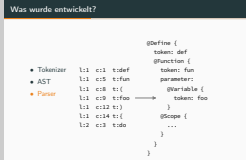
1. Ein Parser.
2. Der Parser ist dafür zuständig, dass die Liste von Tokens in einen abstrakten Syntaxbaum umgewandelt werden. ...

Was wurde entwickelt?

- Tokenizer
- AST
- Parser

```
l:1  c:1  t:def
l:1  c:5  t:fun
l:1  c:8  t:(
l:1  c:9  t:foo  ──────►
l:1  c:12 t:)
l:1  c:14 t:{
l:2  c:3  t:do
```

```
@Define {
  token: def
  @Function {
    token: fun
    parameter:
      @Variable {
        token: foo
      }
    @Scope {
      ...
    }
  }
}
```

1. Das heißt, dass der Parser, die Tokens in diese Form überführen muss.
2. Die **@-Symbole** weisen auf eine Objektinstanz hin und das was in den Klammern eingeschlossen ist, ist ein Bestandteil des umschließenden Objektes.

Was wurde entwickelt?

- Tokenizer
- AST
- Parser
- Analyser

Was wurde entwickelt?
- Tokenizer
- AST
- Parser
- Analyser

1. Ein Analyser.
2. Der Analyser ist dafür da, um Fehler zu finden, die nicht aus dem Parsen hervorgehen. Also Fehler die nicht syntaktisch sind, oder nur schwer beim Parsen zu finden sind. ...

Was wurde entwickelt?

- Tokenizer
- AST
- Parser
- Analyser `def fun(){ break; }`

Was wurde entwickelt?
- Tokenizer
- AST
- Parser
- Analyser `def fun(){ break; }`

1. Ein Beispiel wäre die Funktion `fun`.
2. Der AST kann den Code darstellen. Eine Funktionsdefinition, die in dem Funktionsscope ein break Element hat.
3. Bloß macht dies wenig Sinn, da `break` nur in Schleifen eine Funktion hat.

Was wurde entwickelt?

- Tokenizer
- AST
- Parser
- Analyser
- Interpreter

Was wurde entwickelt?
- Tokenizer
- AST
- Parser
- Analyser
- Interpreter

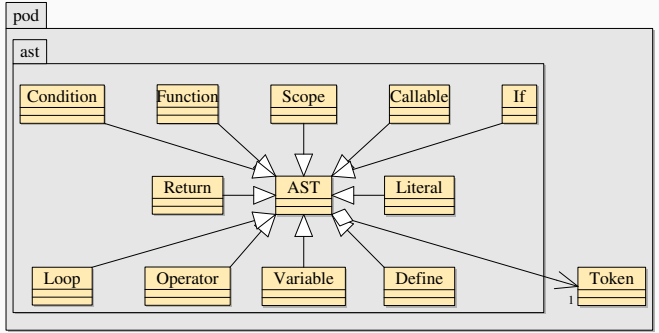
1. Letztlich bedarf es einem Interpreter.
2. Der Interpreter interpretiert den AST, den der Parser erzeugt hat und ist die Komponente – bzw. der Client aus dem CommandPattern – der die Commands ausführt.
3. Somit kommen wir zu wir ... (zu weiteren UML Diagrammen)

AST Klassenhierarchie

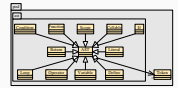
AST Klassenhierarchie

1. Zu weiteren UML Diagrammen.
2. Für meinen Geschmack helfen die Diagramme nicht genug.
3. Sie sind aber die beste Möglichkeit, die ich kenne, anderen Menschen mein Wissen, über die Datenstrukturen, zu übertragen.

AST Klassenhierarchie

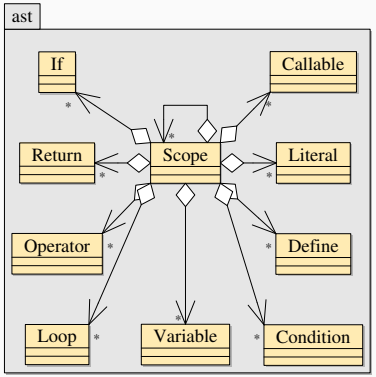


AST Klassenhierarchie

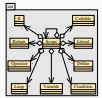


1. Die AST Klassenhierarchie.
2. Alle abstrakten Syntax Klassen haben die **AST** Klasse als Elternklasse. Diese hat ein **Token**, welches das Stück des Makros enthält, welches das Objekt repräsentieren soll.

AST Scope

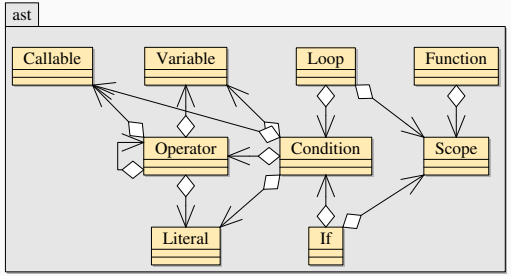


AST Scope

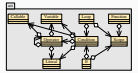


1. Die Scope Abhängigkeiten.
2. Das **Scope** kann Instanzen von allen Klassen aufnehmen.
3. Durch die Aufnahme von anderen AST Instanzen wird hier ein Teil des Baumes erstellt.

AST Klassen

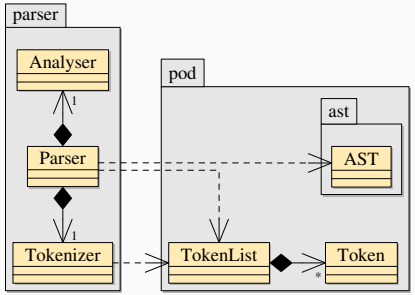


AST Klassen

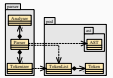


1. Dieses UML Diagramm zeigt die restlichen Abhängigkeiten der Klassen – die meisten Verbindungen sollten Selbstverständlich sein.
2. Eine **Funktion** besteht aus einem **Scope**. Ein **Loop** besteht aus einem **Scope** und einer **Condition**. Eine **Condition** kann aus **Variablen**, **Funktionsaufrufen** usw. bestehen.

Parser Klassen

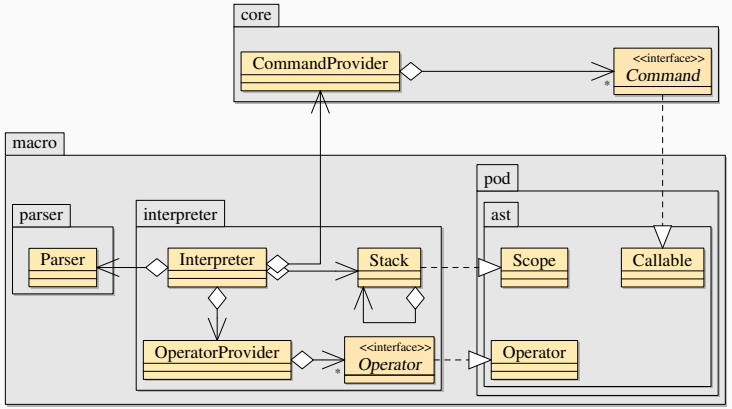


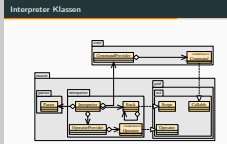
Parser Klassen



1. Die Parser Klassen.
2. Der **Parser** nutzt den **Tokenizer** um eine **TokenList** von dem Makrostring zu erstellen.
3. Diese TokenListe wandelt er dann in einen **AST** um.
4. Der produzierte AST wird dann von dem **Analyser** überprüft.

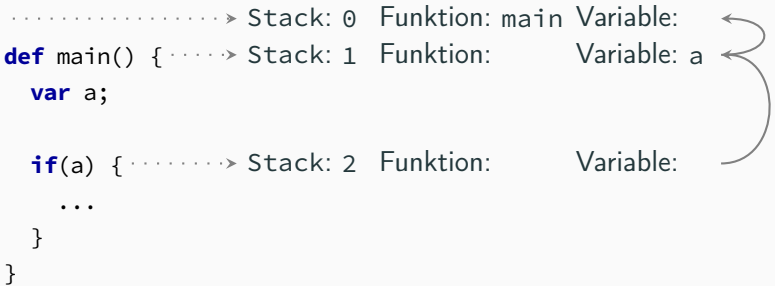
Interpreter Klassen





1. Letztlich die Interpreter Klassen.
2. Der **Interpreter** nutzt den **Parser** um einen AST zu erzeugen, den er dann interpretieren kann.
3. Um dies zu tun, muss der Interpreter auch Daten verwalten. Das wird von dem **Stack** übernommen.
4. Der Stack bildet einen Stack mit Instanzen von sich selber ...

Interpreter Klassen



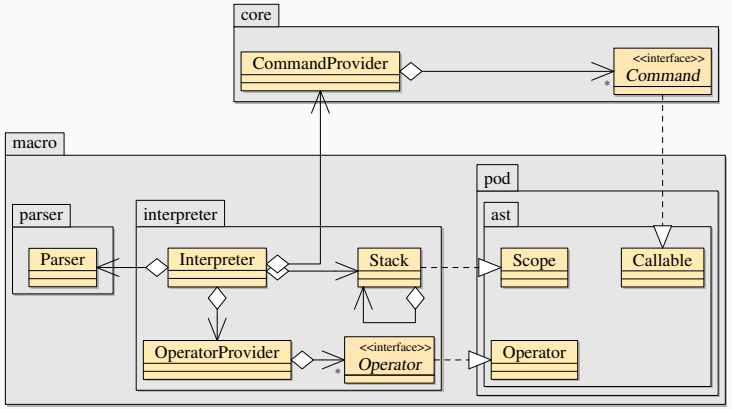
1. Diese Art und Weise einen Stack aufzubauen ist normal in C oder C++.
2. In dem Beispiel wird der Stack durch die drei Stack Instanzen (**1, 2 und 3**) gebildet.
3. Ein späteres Beispiel wird wahrscheinlich mehr Aufschluss über die Funktionsweise bringen. . . . (Denn dafür wisst ihr noch zu wenig.)

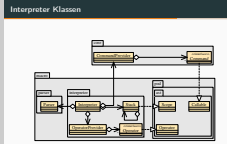
Interpreter Klassen

```
                  ·····> Stack: 0  Funktion: main  Variable:
def main() {      ·····> Stack: 1  Funktion:       Variable: a
  var a;

  if(a) {         ·····> Stack: 2  Fu
    ...
  }
}
```

1. Denn dafür wisst ihr noch zu wenig. Nehmt es erst einmal so hin, das dies Sinnvoll ist.

Interpreter Klassen





1. Der **OperatorProvider** wird von dem Interpreter genutzt um die **Operator**en – wie + und − auf Variablen anzuwenden.
2. Die Nutzung eines OperatorProviders sorgt dafür, dass Operatoren für beliebige Datentypen angeboten werden können, ohne den Interpreter anzupassen.
3. Der Code, des OperatorProviders ist sehr Interessant – leider haben wir nicht genug Zeit um diesen durchzugehen. Deswegen muss ich euch mit dem any Typ abspeisen, der auch hier ein wichtiger Eckpunkt ist.
4. Um Funktionsaufrufe auszuführen – die nicht zu selber definierten Funktionen führen – nutzt der Interpreter den **CommandProvider**.

Realisierung

1. Das war die trockene Theorie.
2. In den Folgenden Folien werde ich an dem Vorherigen Beispiel die einzelnen Schritte erklären, die beim Interpretieren durchlaufen werden.

Tokenizer

Tokenizer

Parser

Parser

Analyser

Interpreter

Probleme

Probleme

Fragen?

- **Katzen:** Ich habe genug theoretische und zum teil langweilige Präsentationen in meinem Studium gesehen. Dem Trend wollte ich nicht folgen – und auch wenn meine Präsentation nicht so viel Wissen vermittelt hat, hatte sie zumindest Katzen. Wer reine Theorie will, ist mit meiner THesis besser beraten als mit mir.

Demo

Demo

Referenzen I

Referenzen I