Aspect C / C++

Johannes Nicolai

Seminar Aspektorientierte Programmierung – SS 2005

Gliederung

- Warum überhaupt AspectC++ ?
- Übersicht über Aspect C++
- Wichtige Begriffe
- BSA Beispielaspekt
- Gegenüberstellung von AspectC++ und AspectJ
- Zusammenfassung
- Quellen

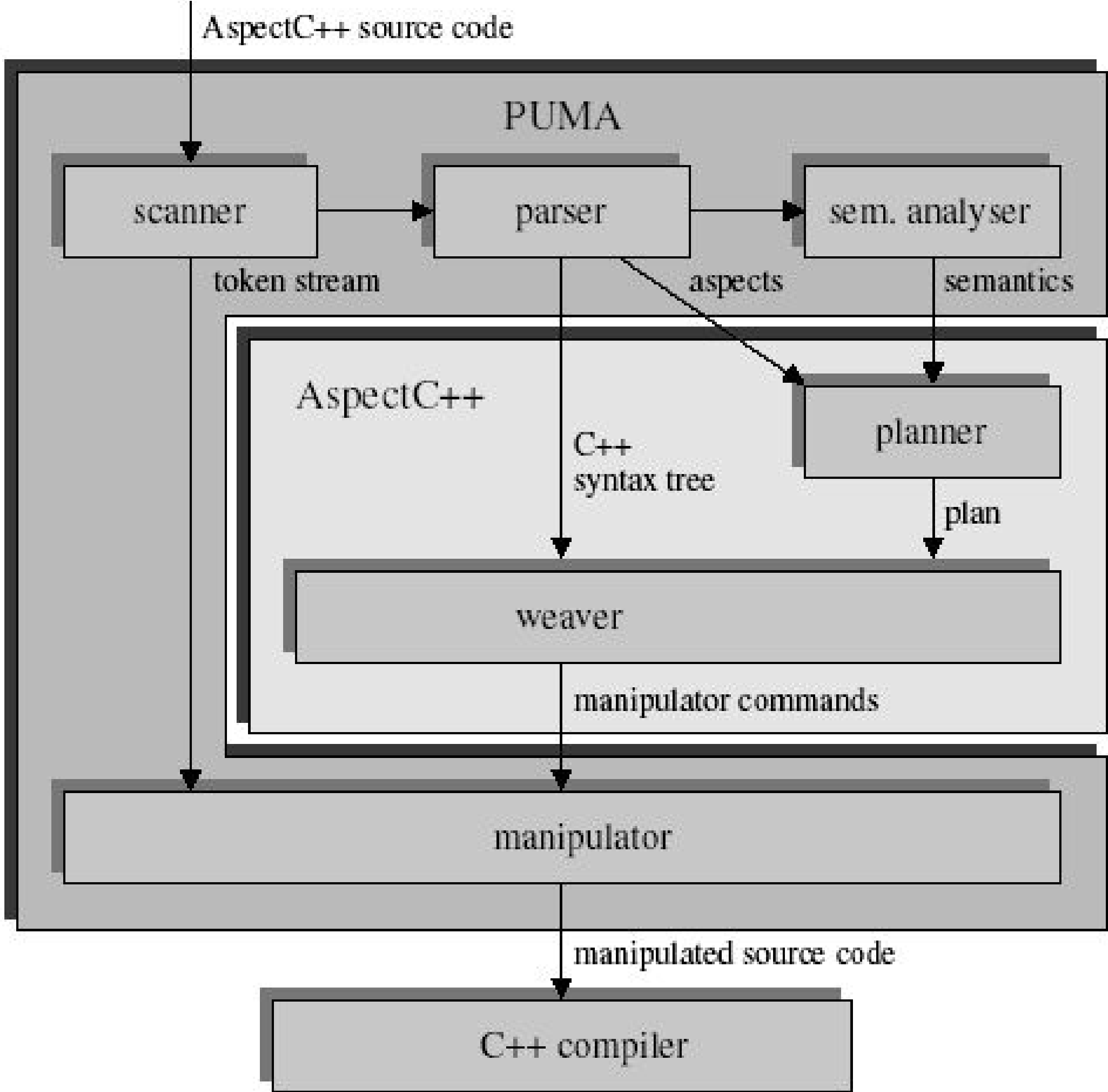
Warum überhaupt AspectC++ ?

- aspektorientierte Programmierung ist mächtiges Konzept
- Effizienz ist und bleibt ein wichtiger Faktor
  - Java ist für eingebettete Systeme manchmal nicht performant genug oder verbraucht zu viele Ressourcen
  - viele Applikationen werden weiterhin in C/C++ entwickelt
- Aspektorientierung in C++ auch nativ über Templates
  - umständlich, kompliziert und weniger mächtig
- Aspektweber für C++ kann für C Code genutzt werden
- wenige Alternativen in der C++ Community verfügbar

Übersicht über AspectC++

- www.aspectc.org
  - universitäres Forschungsprojekt
  - frei verfügbar für Linux, Windows, Solaris, Mac OS X
  - Einbettung in Eclipse möglich
  - Support und Addin für Visual Studio von pure systems
- genutzt, um Betriebsystem (PURE) zu implementieren
  - sehr niedrige Ressourcenansprüchen (8k Hauptspeicher)
- Wirkungsweise / Prinzip:
  - Syntax sehr eng angelehnt an Syntax von Aspect J
  - statischer Weber für C und C++ Programme
  - basiert auf PUMA (Quellcodetransformationssystem)

Übersicht über AspectC++



HPI

Wichtige Begriffe

Join Points:

- Punkte im Programm, auf die sich ein Aspekt beziehen kann
- Join Points können sein:
  - Funktion (name join point)
  - Klassen (name join point)
  - Objekte (name join point)
  - Methode (name join point)
  - Attribut (name join point)
  - Typ (name join point)
  - Variable (name join point)
  - Punkt, an dem Join Point erreicht / referenziert wird (code join point)

Wichtige Begriffe

Pointcuts:

- charakterisieren eine Menge von Join Points
- Advice / Introduction bezieht sich immer auf Pointcut
- können innerhalb oder außerhalb eines Aspekts definiert werden
- Möglichkeit der virtuellen Deklaration (später dazu mehr)
- zwei verschiedene Arten: Name Pointcut, Code Pointcut

Wichtige Begriffe

Name Pointcuts:

- beschreiben Menge von name join points
- durch Matching Expression charakterisiert
- Beispiele für Matching Expressions:
  - `int C::%(...)` : alle Methoden der Klasse C, die int zurückgeben
  - `% printf(const char*, ...)` : Funktion printf, const char* 1. Parameter
  - `const %& ...::%(...)` : irgendeine konstante Referenz
  - `%List && !derived ("Queue")` : Klassen, welche mit List enden und nicht von Queue abgeleitet sind

- Beispieldeklaration: `pointcut lists()=derived("List");`

Wichtige Begriffe

Code Pointcuts:

- beschreiben Menge von code join points, welche im Kontrollfluss des Programms zu finden sind
- durch Pointcut Expressions charakterisiert
- Beispiele für Pointcut Expressions:
  - `call ("void draw()") && within ("Shape")` : Aufruf der Funktion draw innerhalb einer Methode der Klasse Shape
  - `execution("int foo()")` : bezieht sich auf Implementierung der Funktion foo
  - `execution("int foo()") && cflow(execution("int bar()"))` : bezieht sich auf Implementierung von foo, aber nur, wenn sie innerhalb von bar aufgerufen wurde
- Beispieldeklaration: `pointcut virtual methodcall()=0;`
  - Kann in vererbten Aspekt überschrieben werden

Wichtige Begriffe

Advices:

- beziehen sich auf code Pointcut
- definieren Programmcode, der an jedem beschriebenen code Join Point eingefügt wird
- können nur in Aspekt definiert / überschrieben werden
- Beispiele:
  - `advice execution("void login(...)"): before() { cout << "Logging in" << endl; }`
  - `advice call("int foo()"): after() { cout << "Called function foo" << endl;}`
  - `advice call("int bar()"): around() { cout << "Calling bar" << endl; tjp->proceed(); cout << "Finished" << endl; }`

Wichtige Begriffe

Introductions:

- beziehen sich auf name Pointcut
- erweitern name Join Points, auf die sich Pointcut beziehen um Attribute, Basisklassen oder Methoden
- können nur in Aspekt definiert / überschrieben werden
- Beispiele:
  - `pointcut shapes() = "Circle" || "Polygon";`
  - `advice shapes(): bool m_shaded;`
  - `advice shapes(): void shaded(bool state) {m_shaded=state;}`
  - `advice shapes(): baseclass(SuperShape);`

HPI

Wichtige Begriffe

Aspects:

- **fasst Advices und Introductions zusammen, um einen "cross cutting concern" zu implementieren**
- **Erweiterung des C++ Klassenkonzeptes:**
  - **eigene Methoden, Attribute möglich, auf die advice code zugreifen kann**
  - **Vererbung von Klassen und Aspekten möglich**
  - **Überschreibung von Pointcuts möglich**
  - **Sichtbarkeit von Introductions entscheidet über Sichtbarkeit in name Join Point (public introduction fügt public Attribut ein)**
- **definiert in .ah Datei (aspect header)**

Wichtige Begriffe

Beispiel für Aspekt, welcher Instanzen zählt:

```
aspect Counter {
  static int m_count;
  Counter():m_count(0) {};
  pointcut counted()= "Circle" || "Polygon";
  advice counted(): class Helper {
    Helper() {Counter::m_count++;}
  } m_counter;

  advice execution("% main(...)"): after() {
    cout << "Final count: " << m_count << " objects" << endl;
  }
};
```

Wichtige Begriffe

Join Point API:

- benötigt, um auf Charakteristika der Join Points im advice code Bezug nehmen zu können, Zugriff über Variable tjp
- Methoden der API (kleiner Auszug):
  - static const char* signature()
  - Result *result()
  - That *that()
  - Target *target()
  - void proceed()
  - void* arg(int number)
  - Arg<i>::ReferredType *arg()
- Typen der API (kleiner Auszug):
  - Result, That, Target
  - ARGS, Arg<i>::ReferredType, 0<=i<ARGS

BSA Beispielaspekt

- Idee kam aus BSA Übung und Beispiel für Win32
  - jeder Verstoß führte zu Punktabzug
  - robuste Programmierung als Aspekt
- Wirkungsweise
  - Aspekt prüft Resultat von Systemfunktionen
  - bei Fehler wird Exception geworfen
  - Aspekt nimmt intensiv Bezug auf Join Point API
  - keine Introductions (siehe Counter Aspekt)
- Code verfügbar unter http://myhpi.de/~nicolai/
- Test des Aspekts mittels systrace

BSA Beispielaspekt

```
aspect ThrowLinuxErrors {
...
   advice call( Linux() ) : after() {
      if( IsErrorResult(tjp->result()) ) {
         std::ostringstream os;
         stream_header( os, tjp->signature(), errno );
         stream_params<JoinPoint, JoinPoint::ARGS >::process(os, tjp );
         stream_footer_and_throw( os, errno );
      }
   }
   static void stream_header(std::ostream& os, const char* sig, int code ) {
      os << "LINUX ERROR " << errno << ": " << GetErrorText(code) << endl;
      os << "WHILE CALLING: " << sig << endl;
      os << "WITH: ";
      os << "(";
   }
   static void stream_footer_and_throw( std::ostringstream& os, int code ) {
      os << ")";
      throw Exception( os.str(), code );
   }
...
};
```

BSA Beispielaspekt

```
inline bool IsErrorResult( int* res ) {
   return *res <0;
}

inline bool IsErrorResult( void** res ) {
   return res == 0;
}

inline bool IsErrorResult( FILE** res ) {
   return res == 0;
}

inline std::string GetErrorText( int code ) {
   return strerror(code);
}

pointcut  Linux() ="% open(...)"||"% write (...)"||"% close(...)"||
"% fork(...)"||"% waitpid(...)" ||"% malloc(...)"||"% write(...)"||
"% fopen(...)";
```

BSA Beispielaspekt

```
aspect CatchLinuxErrors {
   advice execution( "% main(...)" ) : around() {
      try {
         tjp->proceed();
      }
      catch( Exception& e ) {
         cerr << "An exception occured: " << e.where() << endl;
      }
   }
};
```

- **Compilieren des Beispiels mittels Tool “ag++”**
  - **ag++ funktioniert wie g++, bindet .ah Dateien automatisch ein**
- **Test nur durch Provozieren von syscall Fehlern möglich**
  - **Mittel der Wahl zur Manipulation von syscalls: systrace**

Gegenüberstellung von AspectC++ und AspectJ

- **Grundsätzlich ähnliche Funktionalität und Syntax**
- **Vorteile AspectC++**
  - **Einheitlichere Behandlung von name und code join points**
  - **Support für generic advices (Nutzung von statischen Typinformationen der JoinPoint API)**
  - **Sehr hohe Performance, minimaler Overhead**
  - **Sehr guter Kontakt zu Entwicklern, da vergleichweise kleines Projekt**
- **Vorteile AspectJ**
  - **Unterstützt get/set pointcut functions**
  - **Declare error / warning Konzept**
  - **Eingebauter Support für Exceptions (after Throwing)**
  - **Größerer Support von der AOP community**

Zusammenfassung

- AspectC++ ist statischer Weber für C / C++ Code
- AspectC++ ist frei unter der GPL verfügbar
- Syntax und Funktionalität sehr ähnlich zu AspectJ
- wichtige Konzepte, um mit AspectC++ arbeiten zu können:
  - name / code Join Points
  - name / code Pointcuts
  - matching / pointcut expressions
  - advices / introductions
  - aspects
  - Join Point API
- Hervorragende Dokumentation unter www.aspectc.org

Quellen

- www.aspectc.org
- On Typesafe Aspect Implementations in C++, Lohmann, Spinczyk
- Aspect C++: An Aspect- Oriented Extension to C++, Spinczyk, Gal, Schröder-Preikschat
- Aspect-Oriented Programming with C++ and Aspect C++, AOSD 2005 Tutorial, Lohmann, Spinczyk
- Program Instrumentation for Debugging and Monitoring with AspectC++, Mahrenholz, Spinczyk, Schröder- Preikschat
- AspectC++, A Language Overview, Spinczyk
- Documentation: Ag++ Manual, Blaschke
- AspectC++ Language Reference, Urban, Spinczyk
- www.myhpi.de/~nicolai/