FOG

Thomas Grzenkowski

sithgrze@stud.informatik.uni-erlangen.de

Die Weiterentwicklung von OOP zur Unterstützung von Entwurfsmustern und Aspekten erfordert die Unterstützung von Seiten der Sprache. Die Sprache vom Flexible Object Generator (FOG) ist eine Obermenge von C++ und unterstützt Metaprogrammierung und Reflexion zur Compilierungszeit. Die Syntaxgeneralisierung macht es z.B. in FOG möglich, dass Code für Interfaces und Implementierung nicht mehr doppelt abgelegt werden müssen. Weiterhin ermöglicht es Kompositionsregeln für wiederholte Deklarationen: Es können damit Klassen, Arrays, Aufzählungen (enumerations) und Funktionen erweitert werden. Dies ermöglicht das Weben, welches für wiederverwendbare Implementationen von Entwurfsmustern und für Aspekt-orientierte Programmierung benötigt wird.

Allerdings verlangt C und C++ die „One Definition Rule“ (ODR): Alles darf nur genau einmal definiert werden. Was zur Folge hat, dass die Funktionen und Klassen geschlossen sind. Dies verhindert jedoch in C und C++, dass man durch mehrmaliges definieren die bisherigen Definitionen erweitern könnte. Wünschenswert wäre jedoch, wenn man Klassendeklarationen auf diese Weise erweitern könnte. Darum bietet FOG die „Composite Definition Rule“ (CDR) an: Weitere Definitionen erweitern die bisherigen, sofern sie sich nicht widersprechen.

Um dies zu erreichen ist FOG als Präprozessor für C++ entwickelt worden. Er ist ein nicht kompatibler Ersatz für den von C nach C++ übernommenen Präprozessor cpp. Vor folgende Probleme stellt die Verwendung von cpp:

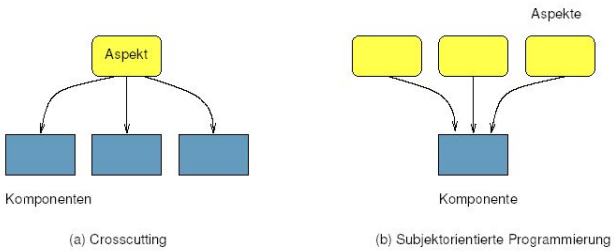
- Cpp sollte ersetzt werden anstatt beseitigt werden.
- Die Kompilierzeitprogrammierung ist notwendig, um Deklarationen zu konfigurieren.
- Muster und AOP erfordern das Weben.
- Die One Definition Rule muss umgangen werden.
- Introspection ist für einfache Anwendungen nützlich.
- Reflexion ist für hoch entwickelte Anwendungen fast unverzichtbar.
- Lexikalische Redundanz sollte beseitigt werden.
- Vorhersagbarer Code sollte automatisch zur Verfügung gestellt werden.
- Ein Konzept sollte sich durch einen einzelnen Aufruf realisieren lassen.
- Überlappende Deklarationen sollten erlaubt werden.

Der experimentelle Übersetzer heißt Flexible Object Generator (FOG). Die Spracherweiterung selber heißt auch FOG. Der Übersetzer bietet:

- Verbessert C++ (aufwärts kompatibel, ohne neue reservierte Wörter).
- Übernimmt die Umorganisierung des Quellcodes.
- Übernimmt den Aufbau der Deklarationen.
- Interpretiert Metaprogramme.

Herkömmliche Programmierung beschäftigt sicht mit der Entwicklung von Progammen passierend auf deren Verarbeitungsinstanz.
Metaprogrammierung beschäfigt sich mit dem programmieren auf Programminstanzen: die Klassen, Funktionen und deklarierten Variablen die ein Programm ausmachen.

Als Unterstützung für AOP verfolgt FOG den Ansatz von Subject Oriented Programming (SOP). Das heißt es ermöglicht mit Hilfe von CDR das Zusammensetzen von Klassen aus verschiedenen Fragmenten/Quellen, die z.B. verschiedene Sichtweisen (Subjects) auf die gleiche Klasse implementieren.



SOP verfolgt damit einen anderen Ansatz als die crosscutting-concerns in AOP. Wie SOP bietet auch FOG allerdings kein Konstrukt wie Pointcut in AspectJ. Damit ist das Weben von crosscuts nur sehr stark eingeschränkt möglich und damit ist FOG nur bedingt für AOP einsetzbar.

1. Beispiel – Monitor Aspekt

Es soll zu der Klasse Monitored ein Monitor Aspekt hinzugefügt werden.

monitor.fog:

```
using "monitor.inc";

class Monitored
{
	$DerivedMonitor::install();
private:
	int *_i;
public:
	int *f() { return _i; }
	const int *fc() const { return _i; }
	const volatile int *fcv() const volatile { return _i; }
	volatile int *fv() volatile { return _i; }
};
```

Mit **using** werden zusätzliche Dateien eingebunden: Es handelt sich dabei um ein verbessertes *#include*: FOG kümmert sich darum, dass keine Datei mehrfach eingebunden werden kann, und es damit nicht zu mehrfach Deklarationen kommen kann (Das was man normalerweise mit Hilfe eines *#ifdef, #define* und *#endif* händisch vermeidet).

Ein führendes **$** kennzeichnet Metaobjekte.

Der Monitor Aspekt wird nun durch den Aufruf der Metafunktion **$DerivedMonitor::install()** dazu gewoben. Die Metaklasse DerivedMonitor selber befindet sich in der eingebundenen Datei monitor.inc.

volatile: Funktion darf von mehreren Threads gleichzeitig genutzt werden.

monitor.inc

```
class DerivedMonitor : public Monitor
{
	export/interface Monitor;
	export/implementation Monitor;
};
```

export legt fest, dass fog nach dem präcompilieren das Interface und die Implementation von **DerivedMonitor** in der gleichen Datei wie **Monitor** ablegen soll.

monitor.inc – Fortsetzung:

```
auto Monitor::~Monitor()
{
	auto if (has_monitor)		// Avoid execution for Monitor and its derived classes.
	{
		auto for (iterator f = $functions(); f; ++f)
		{
			auto if (f->is_static())
				;
			else if (f->is_volatile())
				;
			else if (f->is_const())
			{
				$f->specifier() :{ entry { ReadOnlyLock aLock(_monitor); } };
			}
			else
			{
				$f->specifier() :{ entry { ReadWriteLock aLock(_monitor); } };
			}
		}
	}
}; // [[derived tree]];
```

auto deklariert Metavariablen und -funktionen.

Meta-Konstruktor (im Beispiel nicht definiert):
auto Monitor::Monitor() {...}
Zum definieren von „facilities“, welche später genutzt werden.

Meta-Destruktor:
auto Monitor::~Monitor() {...}
Zum erweitern von bereits existierenden Deklarationen.

Mit Hilfe vom **iterator** kann man schrittweise über eine Liste von Metaeigenschaften gehen. In diesem Beispiel: eine Liste der Funktionen (**$functions()**).

Zum Hinzufügen von Code an bereits bestehende Funktionen existieren fünf Regionen: **entry**, **pre body**, **post** und **exit**. Z.B.:

```
public bool Beispielklasse::Beispielfunktion()
:{              //: nicht vergessen
                entry { /* Code */ };
                exit { /* Code */ };
};              //; nicht vergessen
```

Die Angaben vor dem Doppelpunkt liefert uns im Monitorbsp. **$f->specifier()**.
(Achtung Syntax war in früheren Versionen anders! Leicht erkennbar an den Eckigen Klammern um die Regionen – z.B. [[entry]])

monitor.inc – Fortsetzung:

```
class Monitor
{
        auto number has_monitor = false;
public:
        class ReadOnlyLock
        {
        private:
                Monitor& _monitor;
        public:
                inline ReadOnlyLock(Monitor& aMonitor) : _monitor(aMonitor) { _monitor.enter_shared(); }
                inline ~ReadOnlyLock() { _monitor.leave(); }
        };
        friend class ReadOnlyLock;
        class ReadWriteLock
        {
        private:
                Monitor& _monitor;
        public:
                inline ReadWriteLock(Monitor& aMonitor) : _monitor(aMonitor) { _monitor.enter_exclusive(); }
                inline ~ReadWriteLock() { _monitor.leave(); }
        };
        friend class ReadWriteLock;
private:
        void enter_exclusive() { /* ... */ }
        void enter_shared() { /* ... */ }
        void leave() { /* ... */ }
};

auto declaration Monitor::install()
{
        class $This : auto $Dynamic
        {
                private $Dynamic _monitor;
                auto number has_monitor = true;
        };
}
```

Monitor::install() implementiert das Hinzufügen der Monitor-Klasse zu der aufrufenden Klasse (hier Monitored).

2. Beispiel – Marshalling Aspekt

Es soll zu einer Message-Klasse ein Marshalling-Aspekt hinzugefügt werden.

marshal.fog:

```
class Message
{
	// ...
};

class StockReport : public Message
{
	export/interface Message; export/implementation Message;
private:
	unsigned long _item_number;
	short _stock_level;
	// ...
};



using "marshal.inc";

class Message
{
	$Marshal::install();
};
```

Der Marshalling-Aspekt wird nun durch den Aufruf der Metafunktion **$Marshal::install()** dazu gewoben. Die Metaklasse Marshal selber befindet sich in der eingebundenen Datei marshal.inc. Sie fügt die Funktionen marshal und unmarshal hinzu:

- **public virtual size_t marshal(unsigned char dataBuffer[])**: Füllt dataBuffer[] mit dem Bytestrom und gibt die größe der Nachricht zurück.

- **public static !inline $Scope *unmarshal(unsigned char dataBuffer[])**: Der Funktion wird ein Bytestrom übergeben und gibt einen Pointer auf das erstellte Objekt zurück bzw. 0 bei einem Fehler.

marshal.inc:

```
auto class Marshal {};

typedef unsigned long size_t;

auto declaration Marshal::install()
{
	auto type MessageClass = $Scope;
	auto static statement switchBody =
	{
		default:
		return 0;
	}
	auto number byte_count = 0;
	protected enum MessageTypes {};
```

```
	public virtual size_t marshal(unsigned char dataBuffer[]) const
	:{
		derived(true) entry
		{
			unsigned char *p = dataBuffer;
			*p++ = MESSAGE_@Scope;
			*p++ = @byte_count;
		}
		derived(true) exit
		{
			return p - (dataBuffer+2);
		}
	};

	public static !inline $Scope *unmarshal(unsigned char dataBuffer[])
	{
		switch (dataBuffer[0])
			@switchBody;
	}

	public static @Scope *make(unsigned char dataBuffer[])
	:{ derived(true)
		{
			if (*p != @byte_count)
				return 0;
			else
				return new @{Scope}(dataBuffer);
		}
	};

	auto ${Scope}()
	{
		protected enum ${MessageClass}::MessageTypes { MESSAGE_$Dynamic };
		auto switchBody +=
			case MESSAGE_$Dynamic:
				return ${Dynamic}::make(dataBuffer);
		private inline/implementation ${Dynamic}(unsigned char dataBuffer[])
		{
			unsigned char *p = dataBuffer + 2;
		}
	}

	auto ~${Scope}()
	{
		auto for (iterator i = $variables(); i; i++)
			auto if (!i->is_static())
				$i->type().marshal($i->id());
	}
}
```

Für **marshal** wir hier nur das Framework definiert. Der Funktions-Body selber wird vom Meta-Destruktor definiert. Auch **make** wird erst vollständig vom Meta-Destruktor definiert. S.h. Code auf der folgenden Seite.

Mit **Scope** bekommt man Zugriff auf die Subklasse. Dabei liefert **@** statt **$** den Rückgabetyp.

switchBody wird am Anfang definiert, im Konstruktor erweitert und in der Funktion **unmarshal** verwendet.

marshal.inc – Fortsetzung:

```
auto declaration unsigned long::marshal(expression name)
{
	byte_count += 4;
//	using marshal
	public virtual size_t marshal(unsigned char dataBuffer[]) const
	{
		*p++ = ($name >> 24) & 0xFF;
		*p++ = ($name >> 16) & 0xFF;
		*p++ = ($name >> 8) & 0xFF;
		*p++ = $name & 0xFF;
	}
//	using ${Scope}
	private ${Scope}(unsigned char dataBuffer[])
	{
		{
			unsigned long temp = *p++;
			temp = (temp << 8) | *p++;
			temp = (temp << 8) | *p++;
			temp = (temp << 8) | *p++;
			$name = temp;
		}
	}
}

auto declaration short::marshal(expression name)
{
	auto byte_count += 2;
//	using marshal
	public virtual size_t marshal(unsigned char dataBuffer[]) const
	{
		*p++ = ($name >> 8) & 0xFF;
		*p++ = $name & 0xFF;
   }
//	using ${This}
	private ${Scope}(unsigned char dataBuffer[])
	{
		{
			unsigned long temp = *p++;
			temp = (temp << 8) | *p++;
			$name = temp;
		}
	}
}
```

mit Hilfe von **$i->type()** wird im Destruktor der Typ der aktuell zu bearbeitenden Variable ermittelt und abhängig davon die entsprechenden **marshal**-Funktion aufgerufen bzw. **${Scope}(unsigned char dataBuffer[])** fürs unmarshalling dazugewoben.

auto declaration unsigned long::marshal(expression name) für long
auto declaration short::marshal(expression name) für short

Literatur

- Homepage von SOP bei IBM Research: http://www.research.ibm.com/sop/
- „Program Instrumentation for Debugging and Monitoring with AspectC++“ von Daniel Mahrenholz, Olaf Spinczyk und Wolfgang Schröder-Preikschat: http://ivs.cs.uni-magdeburg.de/bs/papers/isorc02/isorc2002.pdf
- Aspektorientierung und Programmfamilien im Betriebssystembau – Dissertation von Olaf Spinczyk: http://ivs.cs.uni-magdeburg.de/bs/forschung/dissertationen/aspect-familie-os.shtml
- Homepage von FOG: http://www.computing.surrey.ac.uk/research/dsrg//fog/v2.0.4/
  (*Bei den Links auf der Seite ist evtl. vom Link .../fog/v/[htmldatei] das /fog/v herauszuschneiden, da man sonst nicht die neueste Version bekommt.*)