Betriebssystemtechnik

Operating System Engineering (OSE)

“Synthesizing Objects“ [1]

(U. Eisenecker und K. Czarnecki)

Motivation

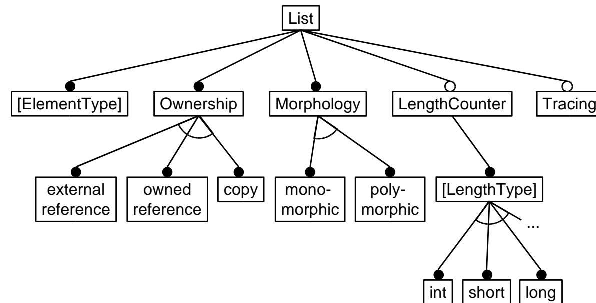
- Produktlinienkomponenten müssen wiederverwendbar sein
  - statisch/dynamisch konfigurierbar
  - generisch formuliert
- Welche Programmiersprachenmechnismen für die generische Programmierung können genutzt werden?
  - **C++ *Templates***!
- Wie beschreibt man Variabilität auf einem problemadequaten abstrakten Niveau?
  - **Merkmalmodelle**!

→ **Merkmal**- getriebene Generierung mittels *Templates*

© 2005 Olaf Spinczyk

Beispielszenario - Merkmale

- eine Familie von Listen- Klassen



© 2005 Olaf Spinczyk

Beispielszenario - Architektur

- ... in Form einer (erweiterten) funktionalen Hierarchie
  - optionale Merkmale werden optionale obere Schichten
  - weitere Variabilität durch Konfigurierbarkeit der Schichten
  - Basisschicht mit Konfigurationswissen

Layer	Class
Tracing Layer	`TracedList`
Counter Layer	`LenList`
BasicList	`PtrList`

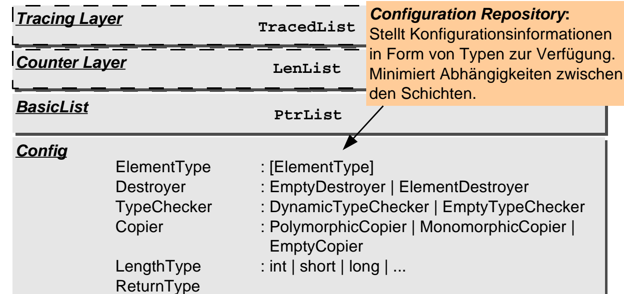
Config

ElementType	: [ElementType]
Destroyer	: EmptyDestroyer | ElementDestroyer
TypeChecker	: DynamicTypeChecker | EmptyTypeChecker
Copier	: PolymorphicCopier | MonomorphicCopier | EmptyCopier
LengthType	: int | short | long | ...
ReturnType	

© 2005 Olaf Spinczyk

Beispielszenario - Architektur

- ... in Form einer (erweiterten) funktionalen Hierarchie
  - optionale Merkmale werden optionale obere Schichten
  - weitere Variabilität durch Konfigurierbarkeit der Schichten
  - Basisschicht mit Konfigurationswissen



PtrList (1) – Daten und Typen

```
template <class Config_>
class PtrList {
public:
  // Config als Member-Typ zur Verfügung stellen:
  typedef Config_ Config;
private:
  // Weitere Typen aus dem Configuration Repository:
  typedef typename Config::ElementType ElementType;
  typedef typename Config::SetHeadElementType SetHeadElementType;
  typedef typename Config::ReturnType ReturnType;

  // Diese Typen repräsentieren Policies!
  typedef typename Config::Destroyer Destroyer;
  typedef typename Config::TypeChecker TypeChecker;
  typedef typename Config::Copier Copier;
private:
  // Datenelemente
  ElementType *head_; // jede Liste speichert einen Zeiger auf 1 Element
  ReturnType *tail_;  // weitere Elemente stecken in der Liste tail_

public:
  // Methoden (siehe nächste Folien)
  ...
```


PtrList (2) - Methoden

```
  // Konstruktor: erzeugt eine neue Liste
  PtrList(SetHeadElementType &h, ReturnType *t=0) : head_(0), tail_(0) {
    setHead(h);
  }

  // Destruktur: ggf. das referenzierte Element löschen
  ~PtrList() { Destroyer::destroy (head_); }

  // Initialisierung des Elements: ggf. Typprüfung und Kopie
  void setHead(SetHeadElementType &h) {
    TypeChecker::check(h);
    head_ = Copier::copy(h);
  }

  // Setter und Getter Funktionen
  ElementType &head() { return *head_; }
  void setTail(ReturnType *t) { tail_ = t; }
  ReturnType *tail() const { return tail_; }
};
```

- wozu eigentlich `SetHeadElementType&`?
  - je nachdem, ob die Elemente kopiert werden, ist das `ElementType&` oder `const ElementType&`

Policies (1) - `Destroyer`

```
template <class ElementType>
struct ElementDestroyer {
  static void destroy(ElementType *e) {
    delete e; // Heap speicher wird freigegeben, Destruktor ausgeführt
  }
};
```

```
template <class ElementType>
struct EmptyDestroyer {
  static void destroy(ElementType *e) {
    // NICHTS!
  }
};
```

- der `ElementDestroyer` muss verwendet werden, wenn die Liste Elemente vor dem Einfügen kopiert
- Aufrufe von `EmptyDestroyer::destroy()` kann der Optimierer komplett eliminieren!

Policies (2) - `TypeChecker`

```
template <class ElementType>
struct DynamicTypeChecker {
  static void check(const ElementType &e) {
    // Typüberprüfung mittels RTTI zur Laufzeit (Debug-Modus)
    assert(typeid(e)==typeid(ElementType));
  }
};
```

```
template <class ElementType>
struct EmptyTypeChecker {
  static void check(const ElementType &e) {
    // NICHTS
  }
};
```

- der `DynamicTypeChecker` sorgt dafür, dass alle Elemente der List den selben Typ haben (*monomorphic list*)
- ... und noch eine Klasse, die **NICHTS** tut.

Policies (3) - `Copier`

```
template <class ElementType>
struct PolymorphicCopier {
  static ElementType *copy(const ElementType &e) {
    return e.clone(); // Kopieren mittels einer virtuellen Methode
  }
};
```

```
template <class ElementType>
struct MonomorphicCopier {
  static ElementType *copy(const ElementType &e) {
    return new ElementType(e); // Kopieren mittels Copy-Konstruktor
  }
};
```

```
template <class ElementType>
struct EmptyCopier {
  static ElementType *copy(const ElementType &e) {
    return &e; // liefert einfach das Original
  }
};
```

Zwischenfazit – *Traits* & *Policies*

- unser `PtrList` *Template* ist hochgradig generisch:
  - variable Aktivitäten werden durch *Policies* realisiert
  - der Zugriff auf die richtige ***Policy***- Klasse wird über das *Configuration Repository* gesteuert: eine ***Traits***- Klasse
- was sind *Traits* und *Policies*?

> - ***Traits*** *represent natural additional properties of a template parameter*
> - ***Policies*** *represent configurable behavior for generic functions and types*

D. Vandevoorde and M. Josuttis
C++ Templates: The Complete Guide

`LenList` – optionale Wrapper Klasse 1

```
template <class BaseList>
class LenList : public BaseList { // Template Parameter wird Basisklasse
public:   // Ermittlung des Configuration Repository
  typedef typename BaseList::Config Config;
private: // Ermittlung weiteree Typen aus dem Configuration Repository:
  typedef typename Config::ElementType ElementType;
  typedef typename Config::SetHeadElementType SetHeadElementType;
  typedef typename Config::ReturnType ReturnType;
  typedef typename Config::LengthType LengthType;
private: // Datenelemente
  LengthType length_; // Anzahl der Elemente in der Liste

public:   // Methoden: Erweiterungen zur Verwaltung der Länge ...
  LenList(SetHeadElementType &h, ReturnType *t=0) : BaseList(h,t),
    length_(computedLength()) {}

  void setTail(ReturnType *t) {
    BaseList::setTail(t);
    length_ = computedLength();
  }

  const LengthType &length() const { return length_; }

  LengthType computedLength() const { return tail()? tail()->length()+1 : 1; }
};
```

`TracedList` – optionaler Wrapper 2

```
template <class BaseList>
class TracedList : public BaseList { // Template Parameter wird Basisklasse
public:  // Ermittlung des Configuration Repository
  typedef typename BaseList::Config Config;
private: // Ermittlung weiteree Typen aus dem Configuration Repository:
  typedef typename Config::ElementType ElementType;
  typedef typename Config::SetHeadElementType SetHeadElementType;
  typedef typename Config::ReturnType ReturnType;

public:  // Methoden: Wrapper zur Ausgabe des Funktionsnamens
  TarcedList(SetHeadElementType &h, ReturnType *t=0) : BaseList(h,t) {}

  void setHead(SetHeadElementType &h) {
    cout << "setHead(" << h << ")" << endl;
    BaseList::setHead(h);
  }
  ElementType &head() {
    cout << "head()" << endl;
    return BaseList::head();
  }
  void setTail(ReturnType *t) {
    cout << "setTail(t)" << endl;
    BaseList::setTail(t);
  }
};
```


Konfigurierung – Variante 1

- ... erfolgt durch Anlegen eines *Configuration Repository*

```
struct TracedCopyMonoPersonLenListConfig {

  // typedefs, die von den Komponenten benötigt werden
  typedef Person                              ElementType;
  typedef const ElementType                   SetHeadElementType;
  typedef int                                 LengthType
  typedef ElementDestroyer<ElementType>       Destroyer;
  typedef DynamicTypeChecker<ElementType>     TypeChecker;
  typedef MonomorphicCopier<ElementType>      Copier;

  // PtrList wird in LenList und TracedList verpackt
  typedef
    TracedList<
      LenList<
        PtrList<TracedCopyMonoPersonLenListConfig> > > ReturnType;
};

// und nun noch ein kürzerer Name...
typedef TracedCopyMonoPersonLenListConfig::ReturnType MyList1;
```


Konfigurierung – Variante 1

- ... erfolgt durch Anlegen eines *Configuration Repository*

```
struct TracedCopyMonoPersonLenListConfig {

  // typedefs, die von den Komponenten benötigt werden
  typedef Person                              ElementType;
  typedef const ElementType                   SetHeadElementType;
  typedef int                                 LengthType
  typedef ElementDestroyer<ElementType>       Destroyer;
  typedef DynamicTypeChecker<ElementType>     TypeChecker;
  typedef MonomorphicCopier<ElementType>      Copier;

  // PtrList wird in LenList und TracedList verpackt
  typedef
    TracedList<
      LenList<
        PtrList<TracedCopyMonoPersonLenListConfig> > ReturnType;
};

// und nun noch ein kürzerer Name...
typedef TracedCopyMonoPersonLenListConfig::ReturnType MyList1;
```

Weg der Konfigurations-informationen

Konfigurierung – Variante 2

- ... oder z.B. so:

```
struct RefPolyPersonListConfig {

  // typedefs, die von den Komponenten benötigt werden
  typedef Person                              ElementType;
  typedef ElementType                         SetHeadElementType;
  typedef EmptyDestroyer<ElementType>         Destroyer;
  typedef EmptyTypeChecker<ElementType>       TypeChecker;
  typedef EmptyCopier<ElementType>            Copier;

  // PtrList wird direkt verwendet
  typedef PtrList<RefPolyPersonListConfig> ReturnType;
};

// und nun noch ein kürzerer Name...
typedef RefPolyPersonListConfig::ReturnType MyList2;
```

Zwischenfazit

- mit Hilfe von unterschiedlichen *Configuration Repositories* lassen sich 24 verschiedene Listentypen erzeugen
- wer erstellt die *Configuration Repositories*?
  - der Listen- Entwickler
    - bei 24 Varianten gerade noch machbar, bei tausenden nicht!
  - der Anwender der Listen
    - mühsam und fehlerträchtig
    - viel wissen über interne Strukturen nötig
- Antwort:
  - Konfigurierung durch den Anwender auf abstrakter Ebene mit Hilfe der Merkmale
  - das *Configuration Repository* muss generiert werden; das ist die Aufgabe des ***Configuration Generator***
    - *Template* Metaprogrammierung!

Der *Configuration Generator* (1)

- ... erwartet Merkmale als Eingabe:

```
enum Ownership { ext_ref, own_ref, cp };
enum Morphology { mono, poly };
enum CounterFlag { with_counter, no_counter };
enum TracingFlag { with_tracing, no_tracing };
```

- ... basiert auf einem kleinen Hilfs- *Template*:

```
template <bool cond, class ThenType, class ElseType>
struct IF {
  typedef ThenType RET;
};

// Spezialisierung für false
template <class ThenType, class ElseType>
struct IF<false, ThenType, ElseType> {
  typedef ElseType RET;
};
```

- ... und wird so benutzt: **ganz einfach!**

```
typedef LIST_GENERATOR<Person, ext_ref, poly>::RET MyList3;
```

Der *Configuration Generator* (2)

```
template <
  class ElementType_ = int, Ownership ownership = cp,
  Morphology morphology = mono, CounterFlag counterFlag= no_counter,
  TracingFlag tracingFlag = no_tracing, class LengthType_ = int>
class LIST_GENERATOR {
public:
  typedef LIST_GENERATOR<ElementType_, ownership, ...> Generator;
private:
  // Konfigurierungslogik wird durch IFs ausgedrückt...
  typedef IF<ownership==cp || ownership==own_ref,
             ElementDestroyer<ElementType_>,
             EmptyDestroyer<ElementType_> >::RET Destroyer_;
  typedef IF<morphology==mono,
             DynamicTypeChecker<ElementType_>,
             EmptyTypeChecker<ElementType_> >::RET TypeChecker_;
  ...
  typedef PtrList<Generator> List;
  typedef IF<counterFlag==with_counter, LenList<List>, List>::RET
    List_with_counter_or_not;
  typedef IF<tracingFlag==with_tracing,
              TracedList<List_with_counter_or_not>,
              List_with_counter_or_not>::RET
    List_with_tracing_or_not;
  // öffentliche Typen ... (nächste Seite)
```

Der *Configuration Generator* (3)

```
public:
  // öffentliche Typen
  typedef List_with_tracing_or_not RET;

  struct Config {
    typedef ElementType_        ElementType;
    typedef SetHeadElementType_ SetHeadElementType;
    ...
    typedef RET                 ReturnType;
  };
};
```

- da `Config` jetzt eine generierte Klasse im Generator ist, muss die `PtrList` Klasse minimal angepasst werden.

```
template <class Generator>
class PtrList {
public:
  typedef typename Generator::Config Config;
  // der Rest wie gehabt ...
};
```

Zusammenfassung

- C++ *Templates* bilden das notwendige Handwerkszeug, um mit C++ generisch zu programmieren
- Merkmalmodelle bilden das Handwerkszeug, um Variabilität von Produktlinen zu beschreiben
- in *„Synthesizing Objects“* verbinden U. Eisenecker und K. Czarnecki beide Techniken
  - Trennung der Belange durch *Configuration Repositories* und *Configuration Generators*
  - Anwendung der *Template* Metaprogrammierung
- der Ansatz erfordert, dass man die Nachteile der Programmierung mit C++ *Templates* in Kauf nimmt
  - Fehlermeldungen, Übersetzungszeiten, Syntax, ...

Literatur

[1] K. Czarnecki und U. Eisenecker, *Synthesizing Objects*. Concurrency: Practice and Experience, John Wiley & Sons, Ltd.