Kapitel 6
Objektfabriken

Als *Objektfabriken* oder *Object Factories* bezeichnet man Methoden, die Klassenentwicklung von der eigentlichen Anwendungsprogrammentwicklung zu trennen und zweckspezifisch nur Objekte der Klassen zu instanziieren, die für eine bestimmte Programmaufgabe benötigt werden. Dabei soll es den Anwendungsprogrammierer weder interessieren, wann die Klassen entwickelt werden (so lange sie sich an bestimmte Schnittstellenvorgaben halten), noch wo die Objekte lokalisiert sind. Es kann sich beispielsweise um Objekt handeln, die auf dem gleichen System existieren, oder um Objekte auf anderen Rechnern, die mit einem lokalen Stellvertreter über das Netz verbunden sind.
Objektfabriken gibt es in zwei Varianten:

(a) Laufzeitobjektfabriken, in denen die gewünschten Objekte während der Programmausführung eingebunden werden, was im Bedarfsfall auch in bereits laufenden Programmen ohne Neuinstallation erfolgen kann, und
(b) Compilezeitobjektfabriken, bei denen der Anwendungsentwickler vor der Übersetzung eines Programms entscheidet, welche Klassenkomposition er benötigt, um einen neue, nach Übersetzung aber auch nicht mehr veränderbare Programmversion zu erstellen.

Wir werden beide Varianten vorstellen.

6.1 Laufzeitobjektfabrik

6.1.1 Motivation

Welche Objekte innerhalb eines Programms benutzt werden können, wird in vielen Anwendungen bereits bei der Programmerstellung festgelegt: Durch Einbinden der *Header*–Dateien der verschiedenen Klassen während der Programmentwicklung sind diese dem Anwendungsprogramm direkt bekannt und es kann entsprechende Objekte erzeugen. Werden neue Klassen generiert, weil sich zum Beispiel neue Aufgabenstellungen ergeben haben, müssen in vielen Fällen die Anwendungsprogramme ebenfalls angepasst werden, um die neuen Klassen bekannt zu

G. Brands, *Das C++ Kompendium*, eXamen.press, 2nd ed.,
DOI 10.1007/978-3-642-04787-9_6, © Springer-Verlag Berlin Heidelberg 2010

machen und deren Funktionalitäten nutzen zu können. Werden in Dateien gesicherte Objektinhalte zurück gelesen, so muss dem Programm ebenfalls bekannt sein, zu welcher Klasse das Objekt gehört, damit die Daten korrekt gelesen werden können. Mit anderen Worten: auch Inhalt und Struktur einer Datei müssen zum Zeitpunkt der Programmentwicklung bereits festliegen. Dies führt zu Problemen bei der Weiterentwicklung von komplexerer Software, wie das folgende Beispiel demonstriert:

- Ein Softwareentwickler ***A*** entwickelt ein Anwendungsprogramm auf der Basis einer recht primitiven Klasse `base` und einiger von ihm selbst davon abgeleiteter Klassen `A_n:base` (*n=1, 2, ...*).

  Ein Anwender kann dieses Programm mit Objekten der Klassen `A_n:base` nutzen.
- Ein Softwareentwickler ***B*** entwickelt spezielle Klassen `B_m:base` und stellt die Header–Dateien nebst einer Laufzeitbibliothek zur Verfügung (*das heißt nicht die Softwarequellen*).

  Der Anwender kann das Programm weiterhin nur mit Objekten der Klassen `A_n:base` nutzen, auch wenn er die neuen Bibliotheken von ***B*** besitzt und am Einsatz interessiert ist. Um die neuen Funktionalitäten nutzen zu können, muss er zusätzlich zur Bibliothek des Herstellers ***B*** auch ein Upgrade der Software des Herstellers ***A*** erwerben, das die Kenntnis der Bibliothek mit Objekten des Typs `B_m:base` beinhaltet.[1]
- Um dem Anwender die neuen Funktionen zugänglich zu machen, erweitert der Softwareentwickler ***A*** sein Programm um die Bibliotheksfunktionen von ***B***, allerdings natürlich nur, sofern ausreichende Nachfrage besteht, dies in seinem Interesse liegt und er eine Vergütung erhält. Erst jetzt kann der Anwender, nach Erwerb und Installation des Upgrades, den neuen Umfang nutzen.[2]

Entwickler ***B*** ist damit vollständig abhängig von ***A***, es sei denn, er entwickelt ein vollständiges eigenes Anwendungspaket und tritt in Wettbewerb zu ***A***. ***A*** kann in seiner Schlüsselposition sogar darauf bestehen, dass ***B*** sein Produkt nur über ihn als Zwischenhändler vertreibt.

Der Anwender ist an die Entwicklungszyklen eines bestimmten Herstellers gebunden. Benötigte und existierende Anwendungen werden, falls überhaupt, nur mit einer zeitlichen Verzögerung verfügbar. Für den Anwender entstehen zusätzlich höhere Kosten, da er zu den neuen Funktionen auch ein Upgrade von ***A*** erwerben

---

[1] A muss sein Programm in Teilen anpassen mit den Header-Dateien von B neu übersetzen.

[2] An dieser Stelle könnte natürlich argumentiert werden, dass die Nutzung der `B`-Klassen problemlos möglich ist, wenn alle Methode bereits als virtuelle Methoden in `base` deklariert sind. Das ist zwar korrekt, aber die Anwendung verfügt über keine Mechanismen, Instanzen der `B`-Klassen zu erzeugen. Erst eine Modifikation der Anwendung durch A, die genau dies beinhaltet, erlaubt die Nutzung der `B`-Klassen.

muss. Hinzu kommt der Wartungsaufwand, der mit der Neuinstallation der Software verbunden ist, verbunden mit den Risiko, dass alte Funktionen nach dem Upgrade andere Eigenschaften aufweisen.

Das Kunststück besteht nun darin, die Entwicklungszyklen von ***A*** und ***B*** zu trennen und es dem Anwender zu ermöglichen, nach Bedarf Produkte zusammenzustellen. Zwar muss ***A*** immer noch bereit sein, die Standardschnittstelle offen zulegen, jedoch wird er es sich kaum leisten können, hier so grundlegend etwas zu ändern, dass Inkompatibilitäten entstehen, sobald andere Hersteller in größerem Umfang für die Anwender wichtige Produkte entwickelt haben, muss jedoch auch weniger fürchten, dass sein Part von anderen übernommen wird.

6.1.2 Die Basisklasse für Fabrikobjekte

Um mit Objekten arbeiten zu können, die zum Zeitpunkt der Anwendungsentwicklung noch unbekannt sind, ist es notwendig, dass

- alle Methoden virtuell sind,
- alle wesentlichen Methoden bereits in der Basisklasse vorhanden sind,[3]
- eine eindeutige Möglichkeit existiert, eine Klasse zu identifizieren und spezifizieren und
- Methoden existieren, die Objekte der gewünschten Klasse erzeugen können.

Beginnen wir also damit, die Komponenten einer Basisklasse für „Fabrikobjekte“ und den Umgang damit genauer festzulegen. Damit die beschriebenen Mechanismen zur Wirkung kommen können, ist ein durchgängiges Arbeiten mit Zeigerobjekten oder Referenzen notwendig. Das führt aber sofort zu zwei Problemen, die wir zunächst lösen müssen:

(a) Sowohl Konstruktor als auch `new`–Operator einer neuen Klasse stehen voraussetzungsgemäß nicht (*unbedingt*) zur Verfügung. Wir können ein Zeigerobjekt daher nicht auf herkömmliche Art durch `new` in der Anwendung erzeugen.
(b) Der Destruktor ist zur Übersetzungszeit ebenfalls nicht bekannt. Da Destruktoren aber nichts weiter als etwas ungewöhnlich benannte Methoden sind, müssen wir sie nur als `virtual` definieren, womit zumindest dieses Problem auch schon gelöst ist.

[3] Natürlich kann man weitere Methoden auch erst weiter oben in der Vererbungshierarchie ebenfalls als virtuelle Methoden einführen. Notwendig ist aber auf jeden Fall die Kenntnis der ersten Klasse, in der eine bestimmte Methode definiert wird, damit der Compiler sie in der Methodentabelle auch identifizieren kann.

Für die Erzeugung eines Objektes einer unbekannten abgeleiteten Klasse können wir eine Funktion verwenden, deren Adresse zur Laufzeit an beliebigen Programmpositionen bekannt gemacht werden kann und die Zugriff auf die notwendigen Konstruktoren und Operatoren zur Erzeugung eines Zeigerobjektes besitzt. Eine statische Funktion in einer Klasse erfüllt diese Anforderungen. Mit einigen sinnvollen Hilfsfunktionen erhält die Basisklasse damit folgende Komponenten:[4]

```
class FactoryObjectsBase:
              public virtual ObjectReferenceCounter {
protected:
     FactoryObjectsBase();
     FactoryObjectsBase(const
                  FactoryObjectsBase& obj);
     FactoryObjectsBase& operator=
                (const FactoryObjectsBase& ob);
     virtual FactoryObjectsBase* clone() const;
public:
     virtual ~FactoryObjectsBase();

     static FactoryObjectsBase* New(){
          return new FactoryObjectsBase();
     };

     FactoryObjectsBase * duplicate() const;
     ...
};//end class
```

An die Stelle der direkten Erzeugung eines Zeigerobjektes durch die Anweisung `new Constructor( )` tritt die Erzeugung mittels der statischen Methode `New( )`.

```
FactoryObjectsBase * fob;
// fob = new FactoryObjectsBase();
// wird ersetzt durch
fob = FactoryObjectsBase::New();
```

Um keine Zweifel an der Verwendung der Klassen aufkommen zu lassen, werden die Konstruktoren (*insbesondere der Kopierkonstruktor!*) als `protected` deklariert. Damit erreichen wir,

- dass keine Nicht–Zeiger–Variablen deklariert werden können,
- Zeigervariablen nur mit der neu definierten statischen Methode erzeugt werden können und

[4] Die Klasse erbt von der an anderer Stelle definierten Klasse **ObjectReferenceCounter**, um mit Mehrfachreferenzen arbeiten zu können. Ich spare aber im weiteren sämtlichen hiermit verbundenen Schreibaufwand ein. Notwendige Kodeergänzungen oder Korrekturen nehmen Sie bitte selbständig vor.

- globale Zuweisungsoperationen zwischen verschiedenen Objekten nicht möglich sind.[5]

Die automatische Verwaltung von Zeigervariablen mit Methoden für das Mehrfachreferenzenmodell ist mit kleinen Einschränkungen möglich. Die Variablendeklaration und Verwendung hat in der folgenden Weise zu erfolgen.

```
Ptr<FactoryObjectsBase>
     f(FactoryObjectsBase::New()); // ok
f=FactoryObjectsBase::New();       // ok

Ptr<FactoryObjectsBase> p;    // Compiler-Fehler !!
```

Aufgabe. Die Methoden `duplicate()` und `clone()` erlauben die Erstellung einer exakten Kopie eines Objektes. Auch hierbei greifen wir auf Ergebnisse aus dem letzten Kapitel zurück und nutzen den Formalismus der `CloneFactory`. Implementieren Sie diese Methoden.

In erbenden Klassen müssen die virtuellen Funktionen überschrieben und die Aufteilung der Methoden in öffentliche und geschützte Bereiche eingehalten werden. Wir unterstützen dies durch Makros für die Klassendefinition und die Implementation:

```
#define FactoryClassDefinition(Child,Parent) \
class Child: public Parent {\
public:\
    ~Child();\
    static FactoryBase* New();\
protected:\
    Child();\
    Child(Child const&);\
    Child(FactoryBase const&);\
    Child& operator=(Child const&);\
    FactoryBase* clone() const;

#define FactoryClassEnd     };

#define FactoryClassImplementation(Child,Parent) \
FactoryBase* Child::New() { return new Child();}\
```

[5] Das schränkt die Verwendung solcher Klassen etwas ein, da einige sonst mögliche Operationen mit Type-Cast-Operatoren unterbunden werden. Da das Typecasting programmiertechnisch aber einiger Aufmerksamkeit bedarf, trägt die Einschränkung zur Sicherheit bei. Globale Zuweisungen zwischen Objekten, deren genauer Typ zum Programmierzeitpunkt nicht bekannt ist, dürften ohnehin auch unter Verwendung einer TypeCast-Akrobatik nur in seltenen Fällen einen Sinn machen. Es ist allerdings darauf zu achten, dass in den erbenden Klassen die Einschränkung erhalten bleibt (*siehe weiteren Text*).

```
FactoryBase* Child::clone()const\
    { return new Child(*this); }
```

Die Implementation einer neuen Klasse wird hiermit recht einfach:

```
in der Header-Datei:
--------------------
FactoryClassDefinition(MyClass,FactoryObjectsBase)
    // Liste der neuen Klassenmethoden
FactoryClassEnd

in der CPP-Datei:
-----------------
FactoryClassImplementation
  (MyClass,FactoryObjectsBase)
// neue Klassenmethoden wie üblich
```

Damit haben wir zunächst den Grundstock für die Klassen in einer Fabrik geschaffen, der in den folgenden Kapitel sukzessive ergänzt wird. In dieser Form ist die Basis aber noch etwas „nackt“, da keine in Anwendungen nutzbaren virtuellen Methoden vorhanden sind. Die eigentlichen Basisklassen für Fabriken werden daher in der Regel von dieser Klasse erben und die Klasse selbst als Fabrikbasis kaum in Erscheinung treten. Allerdings wird dadurch das Problem, wenn man es nüchtern im Sinne der Einführung betrachtet, häufig auch nur verschoben. Wir werden daher noch einige Zusatzmechanismen für die universelle Nutzung erbender Klassen einbauen und auch noch einige andere Probleme lösen, die im ersten Moment vielleicht gar nicht als Probleme erkannt werden. Auch die Fabrik selber, also der Produktionsort für Objekte, steht noch aus.

6.1.3 Klassenidentifikation und Persistenzmodell

Der Titel dieses Kapitels mag aufmerksame Leser vielleicht etwas befremden, ist doch eine eindeutige Identifizierung von Objekten zur Laufzeit problemlos mittels des `typeid`-Operators möglich. Formal ist das natürlich korrekt, aber der `typeid`-Operator hat einige Nachteile:

- Die vom `typeid`-Operator gelieferten Klassennamen sind in der Regel schlecht lesbar. Zur Laufzeit nach Maßgabe des Anwenders generierte Objekte sollten jedoch durch einfach lesbare und klare Typnamen identifizierbar sein (*Nutzerfreundlichkeit*).
- Die vom `typeid`-Operator gelieferten Namen sind systemspezifisch, d.h. eine Bedienungsanleitung, die die Namen eines `typeid`-Operators auf einem Windows-System enthält, ist dann gegebenenfalls auf einem Linux-System nur eingeschränkt nutzbar, da die Bezeichnungen nun anders sind.

  Das ist nicht nur aus Anwendersicht unzumutbar. Wenn wir die Objektfabrik noch mit einem Persistenzmodell versehen wollen, also die Objekte in Dateien

oder auf Datenbanken ablegen und später bei einem erneutem Aufruf der Anwendung weiterverwenden wollen, haben sich möglicherweise die Namen verändert und die Objekte können nicht wiederhergestellt werden.
- In einem Persistenzmodell sind auch die Lebenszyklen von Typen zu berücksichtigen. Hat sich das Attributmodell einer Klasse in einer neuen Version verändert, so muss das beim Rücklesen älterer gesicherter Objektdaten berücksichtigt werden. Auch diese Information wird vom `typeid`-Operator nicht geliefert.

6.1.3.1 Klassenidentifikation

Wir bilden zunächst die normalerweise durch den `typeid`-Operator gelieferten Informationen in ein lesbares und systemunabhängiges Schema ab. Die Informationen für eine Identifikation müssen global und widerspruchsfrei sein. Wir verwenden dazu einen Parameterstring, der jeweils die vollständige Klassenhierarchie widerspiegelt. Dazu definieren wir zwei Methoden, die die Informationen statisch aus dem Klassennamen und dynamisch durch das Objekt liefern.[6]

```
virtual string Classname() const;
static  string classname();
```

Die statische Methode wird später für die Bedienung der eigentlichen Objektfabrik benötigt, kann aber auch in Vergleichen eingesetzt werden. Die Methoden sind in erbenden Klassen zu überschreiben und folgendermaßen implementiert:

```
class NeueKlasse: public AlteKlasse{
public:
    ...
    string  NeueKlasse::Classname() {
        return
           "<class>NeueKlasse"+AlteKlasse::Classname()+
           "</class>";
    }//end function

    string NeueKlasse::classname(){
        return NeueKlasse::Classname();
    }//end function
    ...
```

Die Identifikation einer Klasse oder eines Objektes einer Klasse erfolgt durch einen String der Form

[6] Die unterschiedlichen Schreibweisen sind zwar möglicherweise etwas unglücklich, aber nicht zu umgehen.

```
<class>NeueKlasse<class>AlteKlasse<class>...
<class>FactoryObjectsBase</class>...</class>
```

Sie können sich leicht davon überzeugen, dass diese Kennung in der Tat die Anforderungen erfüllt. Die Identifikation wird bei der Implementation festgelegt, so dass unterschiedliche Laufzeitvarianten nicht möglich sind. Da Identifikation und Klassenname identisch sind, sind Doppelbenennungen, wie sie bei einem Nummerierungssystem denkbar wären, nicht möglich. Gleichlautende Klassennamen sind recht unwahrscheinlich, aber nicht ganz auszuschließen. Würde allerdings versucht, Module mit gleichen Klassennamen in einer Anwendung zu verwenden, so findet bereits der Linker oder das Laufzeitsystem diesen Widerspruch und lässt eine Ausführung der Anwendung nicht zu.

> **Aufgabe.** Ergänzen Sie die Klassendefinition und die Makro-Definitionen aus dem letzten Teilkapiteln. Bei Verwendung der Makros braucht sich der Entwickler bei neuen Fabrikklassen um diese Details nicht mehr zu kümmern.

6.1.3.2 Identifikation des gespeicherten Zustands

Bei der Sicherung der Daten eines Objektes in eine Datei ist die Klassenkennung ebenfalls in die Datei zu schreiben, um beim Rücklesen der Daten wieder ein Objekt der Klasse erzeugen zu können. Bei näherer Betrachtung ist das aber nicht ausreichend. Erinnern Sie sich an Kapitel 2: Eine verbleibende Fehlerquelle beim Lesen von Daten ist die Sicherung mit einer Version einer Klasse und der Versuch des Rücklesens mit einer zweiten Version gleichen Namens, aber modifizierten Inhalts. Solche Fälle können auftreten, wenn alte Daten mit überarbeiteten Programmversionen gelesen werden sollen. Zur eindeutigen Festlegung der Klassenzugehörigkeit eines Objektes gehört daher zusätzlich die Angabe der Versionsnummer der Klasse, von der ein Objekt erzeugt wurde. In den Anfangskapiteln haben wir festgelegt

```
Version 1.2.3
        | | |
        | | Fehlerrelease
        | kompatible Erweiterung
        inkompatible Erweiterung
```

Das Konzept ist hier allerdings nur schlecht durchzuhalten, wie folgende Überlegungen zeigen:

- Eine Fehlerkorrektur muss sich nicht unbedingt in einer Änderung der Speicherstruktur niederschlagen. Gleichwohl kann es natürlich interessant sein, zu wissen, ob ein Objekt vor oder nach der Fehlerkorrektur gesichert wurde, auch wenn dies auf den Lesevorgang keinen Einfluss hat.

- Ein anwendungstechnisch inkompatible Erweiterung muss nicht in Bezug auf die Datensicherung inkompatibel sein, da sich möglicherweise die zu sichernden Attribute gar nicht geändert haben oder die neue Version eine Methode zum konvertieren der alten Daten besitzt.
- Die Änderungen betreffen möglichweise in der Klassenhierarchie recht tief gelegene Mutterklassen, während die eigentlichen Anwendungsklassen von der Änderung gar nicht betroffen waren und immer noch die gleichen Versionsnummern besitzen, obwohl sich die Datenstruktur insgesamt geändert hat.

Ähnlich wie die Klassenbezeichnung bauen wir daher die Versionsbezeichnung rekursiv auf, wobei wir nun das Tagkonzept einsetzen. Jede Klasse fügt in der Rekursion ihre Versionskennung hinzu, wobei es dem Entwickler einer speziellen Klasse überlassen bleibt, wie er die Versionen nun tatsächlich unterscheidet. Die Basisklasse besitzt folgende (hier zum besseren Lesen etwas aufbereitete) Form:

```
string FactoryBase::Version(){
  return "<class>FactoryBase
            <version>1</version>
            <class>MulRef
              <version>1</version>
           </class>
         </class>";
}//end function
```

Wir haben hier nur eine generelle Versionskennzeichnung vorgesehen. Falls weitere Unterscheidungen notwendig sind, kann das Tagkonzept bedarfsgerecht ergänzt werden. Mit dieser Vorgang muss nun das Persistenzmodell ebenfalls nur rekursiv arbeiten, um auf jeder Vererbungsstufe unabhängig von den anderen eine Kompatibilität feststellen zu können.

Aufgabe. Egänzen Sie das Implementationsmakro

```
#define FactoryClassImplementation (Child,Parent,clvers) \
\
string Child::Classname()\
{ return   " <class>"  #Child  +
           Parent::Classname()  +"</class> ";}\
string Child::classname() const \
{ return Child::Classname(); }\
```

durch die Makros für die Versionsinformationen..

6.1.3.3 Persistenzmodell

Für die Speicherung eines Objekts in einer Datei oder einer Datenbank wird der Objektinhalt rekursiv in Textform überführt. Wir sehen dazu zwei Methodengruppen vor:

```
class FactoryBase: virtual public MulRef {
public:
    ...
    virtual string key() const;
    virtual string to_string() const;
    virtual bool   from_string(string);
protected:
    ...
    string encode_data() const;
    bool decode_data(string,string);
    virtual bool check_compatibility();
};//end class
```

`to_string` wandelt die Objektdaten in einen String um, `from_string` ist für das Einlesen der Objektdaten von einem String zuständig. Die Konvertierung erfolgt in zwei Schritten:

(a) Die virtuellen Methoden `from_string` und `to_string` sorgen für die korrekte Durchführung der Rekursion, bei der jede Zwischenklasse selbst für die Versionsüberprüfung und Datenumwandlung zuständig ist. Die beiden Methode werden durch Makros implementiert, so dass der Anwendungsprogrammierer sich nicht um die Details der Rekursionsverfolgung kümmern muss.
(b) Die Methoden `encode_data` und `decode_data` für die Konversion durch und müssen für jede Klasse implementiert werden. Die Makrodefinitionen für die Methoden `from_string` und `to_string` sind so gestaltet, dass das Vergessen einer Implementation zu einem Compilerfehler führt.
(c) Die optionale Methode `check_compatibility` prüft, ob eine gelesene Version mit der Objektversion kompatibel ist.

Es fällt auf, dass die Methoden der Gruppe (b) nicht virtuell sind. In den Makros müssen daher Scope-Operatoren für die richtigen Methodenaufrufe sorgen. Die Makrodefinition aus der Versionsverwaltung wird hierzu erweitert, indem die Einträge hinter dem Versionstag durch

```
return "<class>... +
"<data>" + Child::encode_data() + "</data>" +
Parent::to_string + "</class>"
```

ersetzt werden. Die virtuellen Methoden sorgen beim Aufruf aus einer Anweisungszeile dafür, dass mit der höchsten Klassen begonnen wird. Im Inneren der Methoden wird rekursiv der Scope-Operator genutzt, so dass gezielt sämtliche Klassenmethoden nacheinander aufgerufen werden.

Aufgabe. Implementieren Sie auch das Makro für `from_string`. Berücksichtigen Sie neben den möglichen Fehlern bei der Tagidentifikation und der KLassenidentifikation auch die Methode `check_compatibility`

Die Methode `encode_data` muss lediglich für eine Umwandlung der Attribute in Strings sorgen (*ein Aufbau in Form eines internen Parameterstrings kann zur zusätzlichen Absicherung verwendet werden*), ist aber individuell zu implementieren und kann nicht durch ein Makro unterstützt werden.[7] Auch das Gegenstück zum Einlesen der Daten ist recht einfach zu Implementieren.

Die Speicherung der Daten beispielsweise in in einer Datenbank erfolgt in einer nochmals erweiterten Form, wozu wir nun die Stringtemplates implementieren, um an die bereits eingeführte Konvention der Stringkonvertierung anzuschließen.

```
template <> string to_string(FactoryBase const& f){
    return string("<object><classname>") +
           f.classname() "</classname>" +
           f.to_string()+"</object>";
}
template <>
bool from_string(FactoryBase const& f, string& s){
    string t,u;
    if(!extract_string(s,t,"<object>")
        return false;
    if(!extract_string(t,u,"<classname>")
        return false;
    if(u!=f.classname()) return false;
    return f.from_string(t);
}
```

Die Übergabe als Referenz (parallel können Sie auch Übergaben als Zeigervariablen oder Variablen mit automatischer Verwaltung von Zeigervariablen vorsehen) sorgt dafür, dass immer die zum jeweiligen Objekt gehörenden Methoden aufgerufen werden.

In den Templatespezialisierungen wird das Datenobjekt nochmals gekapselt und mit dem (wiederum gekapselten) Klassennamen versehen. Das sieht auf den ersten Blick wie eine Absicherungsparanoia aus, denn die Methode `from_string()` hat ja bereits die Aufgabe, die Kompatibilität zu überprüfen. Der Sinn lässt sich aber leicht nach Lektüre des folgenden Kapitels erkennen: beim Lesen gesicherter Daten ist das Trägerobjekt möglicherweise noch gar nicht bekannt, sondern muss erst ermittelt werden, damit die Objektfabrik eine Instanz erzeugen kann. Die vorgeschaltete Kapselung der Klassenbezeichnung ermöglich genau dies.

[7] Durch das Makro wird allerdings eine Fehlermeldung ausgelöst, falls Sie die Implementation vergessen sollten.

6.1.4 Die eigentliche Objektfabrik

Nachdem wir nun Möglichkeiten geschaffen haben, Objekte mitsamt ihrer Klassenzugehörigkeit zu speichern und zurück zu lesen, bleibt nun noch die Aufgabe zu bearbeiten, wie ein Programmteil von neuen Klassen erfährt und wie es Objekte von diesen Klassen erzeugen kann. Die Antwort ist relativ einfach und bereits in einem Nebensatz gegeben worden: vor Beginn der eigentlichen Arbeit, also beim Programmstart oder bei der Anbindung weiterer Bibliotheken an eine laufende Anwendung müssen sich alle in einem Laufzeitmodul präsenten Klassen bei einer zentralen Registrierung mit ihrem Klassennamen und der Adresse ihrer `New`-Methode anmelden. Für die Anforderung eines bestimmten Objektes genügt dann die Angabe des Klassennamens.

Um betriebssicher zu sein, muss eine solche Registratur automatisch funktionieren, das heißt die Klassen müssen sich ohne Aufwand für den Entwickler automatisch An- und Abmelden können. Für den An- und Abmeldevorgang definieren wir zunächst zwei Methoden:

```
typedef FactoryBase* (*New)();

bool    register_class(string s, New newAdr);
bool    unregister_class(string s);
```

Damit sich jede Klasse auch wirklich anmeldet, definieren wir einen Datentyp `AutoRegister`, von dem in jedem Implementationsmodul einer Fabrikklasse per Makro eine statische Variable deklariert wird:

```
template <class T> struct AutoRegister {
    AutoRegister(){
        register_class(T::Class(),&T::New);
    };
    ~AutoRegister(){unregister_class(T::Class());};
};//end class

// Makro-Bestandteil für die Implementation
#define FactoryClassImplementation(Child,...) \
\
static AutoRegister<child> Child ## _ar;
```

Beachten Sie den Aufbau der Makroerweiterung. `Child ##_ar` gibt jeder Variablen einen individuellen, von der Klasse abhängigen Namen, was die Definition mehrerer Klassen in einer Datei ermöglicht.

Statische modulinterne (*und dort „ lokal“ globale*) Variable werden vor dem Start der `main()` – Funktion initialisiert und in der gleichen Reihenfolge nach Verlassen der `main()` – Funktion wieder abgebaut, so dass sichergestellt ist, dass sich jede Klasse auch tatsächlich an- und abmeldet. Die Registrierung nehmen wir in einem `map`–Container vor. Hier stoßen wir allerdings auf ein Problem: Der Compiler initialisiert zwar alle statischen Variablen dynamisch zur Laufzeit, aber in welcher

Reihenfolge? Die Registervariable muss ja fertig initialisiert sein, sobald sie von der ersten Fabrikklasse zur Registrierung eines Namens und einer Funktion aufgefordert wird. Um hier sicher zu gehen, verwenden wir eine statisch zur Compilezeit initialisierte Zeigervariable (*die ist dann garantiert im richtigen Zustand während der dynamischen Initialisierung der Objekte*) und erzeugen das dazugehörende Zeigerobjekt dynamisch vor der ersten Nutzung:

```
map<string,New>* reg=0;

void Clear(){
    if(reg!=0){
        delete reg;
        reg=0;
    }//endif
}//end function

bool register_class(string s,New newadr){
    map<string,New>::iterator it;
    if(reg==0){
        reg=new map<string,New>();
        atexit(Clear);
    }//endif
    it=reg->find(s);
    if(it!=reg->end()){
        if(it->second==newadr)
            return false;
    }//endif
    for(it=reg->begin();it!=reg->end();++it){
        if(it->second==newadr){
            return false;
        }//endif
    }//endfor
    reg->insert(pair<string,New>(s,newadr));
    return true;
}//end function

bool unregister_class(string s){
    if(reg==0)
        return false;
    map<string,New>::iterator it;
    it=reg->find(s);
    if(it!=reg->end()){
        reg->erase(it);
        return true;
    }//endif
    return false;
}//end function
```

Die automatischen Registraturvariablen `AutoRegister` sorgen beim Ablauf des Destruktors dafür, dass der Registercontainer tatsächlich wieder geleert wird. Auch das Registraturobjekt selbst wird am Programmende wieder entfernt: die Methode `atexit()` ist eine Standard-C-Methode, die eine Funktionsadresse auf einen speziellen Stack schreibt. Dieser Stack enthält die Informationen zum Aufräumen sämtlicher Initialisierungsschritte, also auch die Adressen sämtlicher Destruktoren globaler Objekte. Durch den Eintrag der Methode `clear()` sorgen wir für das Löschen des Zeigerobjektes, nachdem sämtliche registrierten Klassen durch die `AutoRegister`-Variablen wieder in der Registratur gelöscht sind.

Dieses penible Aufräumen des Speicherplatzes am Ende des Programmes scheint etwas überflüssig, findet doch danach ohnehin nichts mehr statt. Stellen Sie sich aber einmal vor, das Ganze läuft in einer verteilten Umgebung und die Registratur befindet sich nicht auf Ihrem, sondern einem ganz anderen Rechner. Versäumnisse beim Abbau von Ressourcen wirken sich dann fatal aus. Wenn wir uns aber schon hier an sauberes Arbeiten gewöhnen, können solche Fehler im Ernstfall kaum passieren.

Die Erzeugung von Objekten beliebiger registrierter Klassen kann nun mit Hilfe der Klassenbezeichnung erfolgen:

```
FactoryBase* create_object(string s){
    if(reg!=0){
        map<string,New>::iterator it;
        it=reg->find(s);
        if(it!=reg->end()){
            return it->second();
       }//endif
    }//endif
    return 0;
}//end function

// Aufruf in den Funktionen:
MyClass* mf =dynamic_cast<MyClass*>
        (create_object(MyClass::Class()));
```

Damit lassen sich nun zur Laufzeit beliebige Objekte anwendergesteuert erzeugen. Einzige Voraussetzung ist die Registrierung einer entsprechenden Klasse.

> **Aufgabe.** Hilfreich sind weitere Methoden zur Feststellung, welche Klassen registriert sind, ob eine bestimmte Klasse registriert ist, wieviele Registrierungen vorliegen oder zur Erzeugung von Klassen nach Registrierungsindizes. Entwurf und Implementierung sei Ihnen überlassen.

Für die Arbeit mit persistenten Objekten ist auch eine Methode, die Objekte direkt aus einem Datensatz erzeugt, interessant. Als Modifikation der Methode `from_string` erhalten wir beispielsweise

```
bool object_from_string(string& s){
    string t,u;
    FactoryBase* obj=0;
    if(!extract_string(s,t,"<object>")
        return 0;
    if(!extract_string(t,u,"<classname>")
        return 0;
    obj=(FactoryBase*)create(u);
    if(obj==0) return false;
    if(!obj->from_string(t)){
        delete obj;
        obj=0;
    }
    return obj;
}
```

Damit hätten wir über die reine Objektfabrik hinaus auch eine Basis für eine Objektdatenbank geschaffen.

6.1.5 Benutzung neuer Methoden

Die Basisklasse ist naturgegeben mit einer recht kleinen Schnittstelle ausgestattet. Erbende Klassen werden weitere Methoden definieren und daher die eigentlichen Basisklassen für eine Objektfabrik darstellen. Unangenehm wird die Angelegenheit allerdings, wenn weitere Klassen in der Hierarchie ebenfalls mit eigenen Methoden aufwarten, die zu bedienen sind. Sofern das spezielle Programmteile betrifft, die mit den neuen Klassen zur Anwendung hinzukommen, lässt sich dies durch einen `typecast` erledigen:

```
void foo(Ptr<FactoryBase> const& p){
   if(dynamic_cast<SpecialClass*>(p()){
      dynamic_cast<SpecialClass*>(p())->sec_foo();
      ...
```

Funktioniert das nicht, wäre eine Erweiterung des Klassenmodells möglicherweise doch entgegen den mit einer Objektfabrik verbundenen Absichten mit einer Änderung des Anwendungskodes verbunden. Eine Möglichkeit, neue Methoden auf einem indirekten Weg in die Anwendung zu portieren, wäre daher als Alternative zu suchen.

Hierzu statten wir zunächst die Basisklasse mit zwei weiteren Methoden aus:

```
class FactoryBase: virtual public MulRef {
public:
    virtual string function_list() const;
            bool   action(string&);
```

Die Methode `function_list` dient dazu, der Umgebung die neuen Methoden bekannt zu machen. Mit „Umgebung“ ist der Anwender gemeint, denn die Entscheidung, bestimmte Klassenobjekte zu instanziieren und spezielle Methoden darauf auszuführen, kann nur von ihm getroffen werden. Die Methode liefert die Informationen in einer Parameterstring-Kodierung zurück:

```
<names>
    <1>fu_1</1>
    <2>fu_2...
       fu_n</n>
</names>,
<Fu_1>
   <parameter>
      <1>typ_1</1>
      <2>typ_2
         ...typ_m</m>
   </paramater>
   <return>typ_r</return>
   <help>..text..</help>
</fu1> ...
```

Der Name jeder Methode wird zunächst unter dem Bezeichner `names` bekannt gemacht, anschließend wird für jede Methode eine Liste der Übergabeparametertypen, der Rückgabetyp und ein Hilfetext mit beliebigem Inhalt ausgegeben. Der Hilfetext soll den Anwender über die Aufgabe und Wirkung der Methode informieren und hat programmintern sonst keine weiteren Wirkungen.

Die in den Listen spezifizierten Typen können einem Standardkanon wie

```
void, bool, char, int, double, string, ...
```

sowie der Klassenfabrik selbst entstammen (*welche Typen Sie berücksichtigen, sei Ihnen überlassen*). Wir beschränken uns auf die reinen Typbezeichnungen und unterscheiden nicht weiter zwischen

```
Standardkanon: TYP var, TYP& var, TYP const& var
Fabrikobjekte: TYP* var, TYP const* var
```

im Funktionskopf (*wie wir aber später sehen werden, empfiehlt es sich, die Typenliste in der Praxis zu beschränken*).

Der Aufruf einer Methode erfolgt durch die Methode `action`, die einen String als Übergabeparameter enthält. Dieser ist in der Form

```
<Function>fu_k
   <p>val_1</p>
   <p>val_2
      ...val_m</p>
</function>
```

kodiert und enthält bei Rückkehr die von der inneren Methode gegebenenfalls geänderten Daten

```
<result>
   <p>val_1</p>
   ...
   <r>val_r</r>
</result>
```

Die Datenübergabe ist also auf Strings beschränkt. Dies hat den Vorteil, dass falsche Datentypen, wie sie beispielsweise bei Übergabe durch `void`-Pointer mit anschließendem Typecast übergeben werden könnten, ausgeschlossen sind (*sichere Programmierung*). Nachteilig ist der höhere Ressourcenaufwand, wobei man allerdings berücksichtigen muss, dass es sich hier um eine Hilfskonstruktion handelt, um dauernde Programmänderungen zu vermeiden. Bei größeren Erweiterungen wird man vermutlich auch die Programme entsprechend anpassen und kann in diesem Rahmen sicher die neuen Funktionen auch direkt aufrufen.

Der Aufruf der Methoden erfolgt indirekt über die Methode `action(..)`, die im Übergabestring den Methodennamen und die Methodenparameter enthält, nach Rückkehr auch Rückgabeparameter und geänderte Daten. Diese Methode muss nicht überschrieben werden; es genügt, wenn sie in der Basisklasse implementiert ist und auf die Adressen der Funktionen zugreifen kann. An dieser Stelle müssen wir uns nochmals mit Funktionszeigern in C++ auseinandersetzen. Zeiger auf C-Funktionen werden bekanntlich in der Form

```
int (*fpointer)(string);
typedef double (*fpointertype)(double&)
```

als Variable oder Typen definiert, wobei Übergabe- und Rückgabetypen nach Bedarf anzupassen sind und die Definitionen für jeden Funktionstyp erneut auszuführen sind. Klassenmethoden in C++ sind, abgesehen von statischen Methoden, die wie C-Methoden behandelt werden, nicht in der gleichen Weise definierbar, da die Methoden an ein spezielles Objekt gebunden sind. Zusätzlich zu Übergabe- und Rückgabeparametern muss der über einen Zeiger aufgerufenen Methode auch noch mitgeteilt werden, wie sie den `this`-Zeiger auf die eigenen Attribute findet, was nur mit einer spezielle Syntax möglich ist. C++ definiert dazu spezielle Operatoren für die Definition

```
int (MeineKlasse ::* fpointer)(int);
```

die Initialisierung

```
fpointer = &MeineKlasse::meineFunktion;
```

und den Aufruf

```
MeineKlasse obj1, *obj2;
...
i=obj1.*fpointer(10);
j=obj2->*fpointer(20);
```

Die Operatoren `::*`, `.*` und `->*` sind extra für diesen Zweck reserviert. Mit den Funktionszeigern sind aber einige Besonderheiten verbunden, die mit objektorientiertem Arbeiten zu tun haben und am Besten an einigen Beispielen zu erläutern sind:

```
struct A { void f(void);};
struct B: public A { };
...
void (A::* fp)(void);  fp=&B::f;
A a; B b;
(a.*fp)(); (b.*fp)();
```

wird problemlos übersetzt, da die Methode `f()` in beiden Klassen nur eine Instanz hat, also nicht überschrieben wird. Überschreiben wir nun die Klasse, so kompiliert dieser Code nicht mehr, sondern wir müssen für jede Klasse einen eigenen Zeiger definieren:

```
struct A { void f(void);};
struct B: public A {void f(void); };
...
void (A::* fp1)(void);  fp1=&A::f;
void (B::* fp2)(void);  fp2=&B::f;
A a; B b;
(a.*fp1)(); (b.*fp2)(); (b.* fp1)();
```

Das Modell berücksichtigt allerdings virtuelle Vererbung.[8] Der Code

```
struct A { virtual void f(void);};
struct B: public A {void f(void); };
...
void (A::* fp)(void);
fp=&A::f;
A a; B b;
(a.*fp)(); (b.*fp)();
```

ruft die Methode `f()` in Abhängigkeit vom Bezugsobjekt korrekt auf.

In Summe entsprechen diese Kodierungsregeln dem, was man aus objektorientierter Sicht unter Berücksichtigung der strengen Typkontrollregeln von C++ auch erwarten sollte.

Halten wir fest: in Abhängigkeit von den Methodenschnittstellen erhalten wir verschiedene Zeigertypen, die wir ähnlich wie die Klassenschnittstellen registrieren müssen. Wir setzen hierzu eine Kombination aus verschiedenen Techniken ein. Für die Registrierung implementieren wir einen Container, der mit Zeigern arbeitet, d.h. unser SmartPointer kommt zum Einsatz. Alle Funktionszeigertypen besitzen

[8] Die Aussage ist compilerabhängig. GNU-Compiler stehen dem Standard in der Regel am Nächsten und unterstützen solche Eigenschaften, bei anderen muss man es ausprobieren.

eine gemeinsame Basisklasse, mit der die SmartPointer eingerichtet werden. Die speziellen Funktionszeigertypen erben von dieser Basisklasse und sind ihrerseits zweckmäßigerweise Templateklassen, um das Einrichtungsschema möglichst einfach zu gestalten. Solche Implementationsschemata für Klassenmethoden heißen Funktoren (*siehe Kapitel Transaktionsmanagement*). Im einzelnen erhalten wir so:

```
list<Ptr<FunctorBase> >* meth_list=0;
```

Hier werden im Fabrikmodul parallel zur Objektfabrik die Klassenmethoden gesammelt; die dort beschriebenen Implementationsschemata können hier fast nahtlos übernommen werden.

```
struct FunctorBase: public MulRef {
    string clname;
    string funame;
    virtual
    string execute(FactoryBase& obj, string& s)=0;
};//end struct
```

Die Basisklasse leiten wir im Interesse eines einfachen Funktionierens der SmartPointer von `MulRef` ab; mehr sollte man sich dabei allerdings nicht denken. Als Schlüsselbegriffe für den Zugriff auf die Methoden hinterlegen wir hier gleich den Klassen- und den Methodennamen. Die Methode `execute`, die für den Methodenaufruf zuständig ist, ist virtuell und wird bei den Zeigerklassen implementiert. Die übernimmt das Objekt, für das die Methode aufgerufen wird, sowie die Parameter, die an dieser Stelle noch in Stringkodierung übergeben werden.

Von dieser Klasse erben die speziellen Funktionszeigerklassen:

```
template <class T, class R, class P1>
struct Functor1: public FunctorBase {
    R (T::*function)(P1 p);

    Functor1(R (T::& fu)(P1)): function(fu) {}

    string execute(FactoryBase* obj, string& s){
        R r; T* t = dynamic_cast<T*>(obj);
        typename
        PtrTypeCheck<P1,void>::ValueType p1;
                if(t && from_string(p1,s)){
            r=(t->* function)(p1);
            s=to_string(p1);
            return to_string(r);
        }else{
            // Fehlerbehandlung
            return "";
        }//endif
    }//end function
};//end struct
```

Diese Templateklasse ist für eine Methode mit einem Übergabeparameter ausgelegt; sollen dies mehr oder weniger sein, so sind wie bei den TRACE-Klassen und anderen entsprechend weitere Klassenvereinbarungen notwendig, was aber im Grund nur unterschiedlichen Schreibaufwand darstellt.

In der execute-Methode werden nun die Strings in Daten der entsprechenden Datentypen konvertiert, so dass die neuen Methode nicht auf die Verwendung von Strings als Übergabeparameter angewiesen sind. Der dynamic_cast des Objektes ist notwendig, da die Aufrufe aus der Basisklasse erfolgen und hier nur die Klasse FactoryBase bekannt ist. Weiterhin ist zu beachten, dass die Datentypen in der Template-Parameterliste vollständig sein müssen, also TYP alleine nicht genügt, sondern auch anzugeben ist, ob es sich um Zeiger, Referenzen oder gar konstante Versionen davon handelt. Damit die lokalen Variablen korrekt erzeugt werden können, müssen wir unser Standardisierungsklasse PtrTypeCheck bemühen, die dies für uns erledigt.

Mittels einer Registrierungsmethode können nun die Methoden im gleichen Modul, in dem die Klassen registriert werden, angemeldet werden, beispielsweise durch

```
register_function(new
   Functor1<FC,int,int>(&FC::foo_1,
                        FC::Classname(),
                        "foo_1"));
```

Aufgabe. Die Implementation sei Ihnen überlassen, ebenso eine Erweiterung des AutoRegister-Modells für die automatische Anmeldung bei Start der Anwendung.

Es bleibt noch die Implementation der Methode action, die für die Ausführung der neuen Funktion sorgen soll. Diese kann in der Form

```
bool FactoryBase::action(string& s){
    PString ps(s); string t,u; int i;
    list<Ptr<FunctorBase> >::iterator it;
    ...
    for(it=meth_list->begin();
        it!=meth_list->end();it++){
        if((*it)->clname==classname() &&
           (*it)->funame==t){
            ps.pstr(t,s);
            u=(&it)->execute(this,s);
            ps.add_value(t,u);
            s=ps.str();
            return true;
        }//endif
    }//endfor
    return false;
}//end function
```

Wobei die Suche nach dem Funktionszeigerobjekt Ihnen überlassen bleibt. Zu beachten ist der Iteratorzugriff `(*it)->.. . Mit *it` wird das Ptr-Objekt zugänglich gemacht, und mit dem folgenden `->`-Operator auf die spezielle Methode oder das Attribut des Objektes im Ptr-Objekt zugegriffen.

6.1.6 Trennung von Anwendung und Bibliothek

6.1.6.1 Erzeugen einer Dynamischen Bibliothek

Wenn Sie die bisherigen Techniken in praktischen Versuchen erprobt haben (was ich doch schwer hoffe !), haben Sie vermutlich mit verschiedenen Header- und Objektdateien innerhalb einer Entwicklungsumgebung gearbeitet. Wenn eine Klasse dazukommt, muss bei dieser Vorgehensweise alles neu compiliert werden, was nun nicht gerade dem eigentlichen Ziel entspricht.

Der erste Schritt, Anwendungs- und Klassenentwicklung zu trennen, besteht zunächst einmal darin, die Klassen in Bibliotheken zu compilieren und diese später zur Anwendung dazu zu linken.

Wichtig. Die folgenden Ausführungen beziehen sich auf die Arbeit im Betriebssystem Windows; verwenden Sie ein anderes Betriebssystem, ändern sich möglicherweise einige Methodennamen, aber die grundsätzliche Vorgehensweise bleibt bestehen. Details und Beispiele sind aber in der Regel leicht zu finden.

Ein Projekt für eine Bibliothek, wobei wir zweckmäßigerweise gleich eine DLL (dynamic link library), also eine Bibliothek, deren Code erst zur Laufzeit des Programms angebunden wird, in Angriff nehmen, unterscheidet sich nur wenig von einem Anwendungsprogrammprojekt.[9] Wie dort wird zunächst eine Art `main`-Methode implementiert:

```
BOOL WINAPI DllMain(HINSTANCE hinstDLL,
          DWORD fdwReason, LPVOID lpvReserved){
    switch (fdwReason)
    {
        case DLL_PROCESS_ATTACH:
            // attach to process
            // return FALSE to fail DLL load
            break;

        case DLL_PROCESS_DETACH:
            // detach from process
            break;
```

[9] Die Alternative ist eine statische Bibliothek, deren Code während des Linkvorgangs in die Programmdatei eingefügt wird. Das nützt natürlich wenig, wenn wir zu einer tatsächlichen Trennung kommen wollen.

```
        case DLL_THREAD_ATTACH:
            // attach to thread
            break;

        case DLL_THREAD_DETACH:
            // detach from thread
            break;
    }
    return TRUE; // succesful
}
```

Diese hat die Aufgabe, Initialisierungen und Bereinigungen beim Einbinden oder Abschalten einer Bibliothek vorzunehmen. Eine Bibliothek kann in einen einzelnen Prozess oder einen Prozess mit mehreren Threads eingebunden werden. Unterstützt eine Bibliothek beispielsweise keine Threads, so kann sie durch den Rückgabewert FALSE eine Aktivierung unterbinden.

In der Regel wird man hier nicht viel Einfügen müssen. Handelt es sich aber beispielsweise um eine Bibliothek, die auf der lokalen Maschine nur Schnittstellen vorsieht, während der eigentliche Laufzeitkode auf einer anderen Maschine liegt, kann die Initialisierung beispielsweise im Aufbau der Netzwerkverbindung und der Anbindung an die entfernte Anwendung bestehen.

Im Weiteren werden die Klassen und ihre Methoden in fast gewohnter Form implementiert:

```
void DLL_EXPORT foo(void)
{
    cout << "Hier ist Ihre DLL" << endl;
}
```

Die Header-Dateien enthalten die Spezifizierung, was mit dem Makro `DLL_EXPORT` gemeint ist:

```
#include<windows.h>
#ifdef BUILD_DLL
    #define DLL_EXPORT __declspec(dllexport)
#else
    #define DLL_EXPORT __declspec(dllimport)
#endif

#ifdef __cplusplus
extern "C"{
#endif

void DLL_EXPORT foo(void);

#ifdef __cplusplus
}
#endif
```

Die Spezifikation `_declspec(dllexport)` sorgt dafür, dass der Compiler bestimmte Informationen, die normalerweise im Link-Prozess entfernt werden, im Ausführungskode der DLL landen, um eine Anbindung an das Anwendungsprogramm zu ermöglichen. Da sich C und C++ Anwendungen hinsichtlich der Schnittstellendefinitionen auf Maschinenebene unterscheiden, wird zusätzlich festgelegt, dass einheitlich die C-Konvention verwendet wird (die notwendigen Anpassungen nimmt der Compiler automatisch vor). `_declspec(dllimport)` sorgt nun wieder dafür, dass der Linker die Methoden nicht statisch einbindet, sondern sie in eine Umgebung stellt, die bei Bedarf aktiviert wird.

Um Klassen in einer DLL exportieren zu können, werden diese wie gewohnt mit Header- und Objektdatei implementiert, ohne dass das DLL_EXPORT-Makro verwendet wird. Einzige verbindliche Nebenbedingung: die Methoden der Objekte, die im Anwendungsprogramm genutzt werden sollen, müssen sämtlich als `virtual` deklariert werden. Die Header-Dateien werden in der DLL-Headerdatei außerhalb der `extern C`-Anweisung eingebunden:

```
...
    #define DLL_EXPORT __declspec(dllimport)
#endif

#include "MyClass.h"

#ifdef __cplusplus
extern "C"{
```

Exportiert werden nur Erzeugungsmethoden für ein Objekt. Die exportierte DLL-Methode wird hierdurch beispielsweise zu

```
void* DLL_EXPORT foo(void){
    return static_cast< void*>(new MyClass());
}
```

Im Arbeitsprogramm können Objekte der Klasse durch den Aufruf

```
MyClass* p = static_cast<MyClass*>(foo());
```

erzeugt und völlig normal genutzt werden.

Wieso das funktioniert, lässt sich durch ein wenig Überlegung schnell herausfinden. Die Methode `foo()`, die einen Zeiger auf das Objekt erzeugt, wird vom Anwendungsprogramm direkt aufgerufen, und damit der Compiler und das Laufzeitsystem das problemlos hinbekommen, müssen die notwendigen Linkinformationen in der DLL hinterlegt sein, und diese Methodik ist nur für C definiert, was somit `_declspec(..)` und `extern C` notwendig macht.

Das Objekt selbst wird im Kontext der DLL erzeugt; das Anwendungsprogramm erhält lediglich einen Zeiger darauf, der wiederum einen Zeiger auf die Adresse der Methodentabelle des Objektes (im Adressraum der DLL) enthält. Der Austausch von Übergabeparametern und Rückgabewerten erfolgt über den Stack. Es besteht

somit überhaupt kein Problem, das Objekt völlig normal zu bedienen, da keinerlei Namensauflösungen zwischen den Programmteilen notwendig sind.[10]

Aufgabe. Im ersten Anlauf, den Sie nun realisieren können, wird eine DLL-Datei erzeugt (nebst einigen anderen Dateien, die der Compiler ebenfalls benötigt), die Sie im Projekt für die Anwendung in die Liste der benötigten Bibliotheken eintragen müssen. Die Header-Dateien werden normal in das Anwendungsprogramm integriert.

Teilen Sie Ihre bisherigen Versuche in Grundprogramm und Klassenbibliothek auf. Sie werden feststellen, dass noch einige zusätzliche Handgriffe notwendig sind, um alles wieder zum Laufen zu bewegen. Wir klären das im nächsten Teilabschnitt.

6.1.7 Dynamische Einbindung einer DLL

Wir haben damit die Klassenentwicklung von der Anwendungsentwicklung abgekoppelt, aber nicht die Anwendungsentwicklung von der Klassenentwicklung, denn DLL muss während des Link-Vorgangs eingebunden werden, und möglicherweise sind sogar die Headerdateien noch zu berücksichtigen. Es besteht aber auch die Möglichkeit, eine DLL zu einem laufenden Programm durch folgende Ergänzung hinzuzufügen:[11]

```
typedef void* (*Foo)();
Foo foo;
HINSTANCE hDll=LoadLibrary("....dll");
if ( hDll == NULL ) error("DLL not found");
foo = (Foo) GetProcAddress(hDll, "foo");
```

Der Rest funktioniert wie oben beschrieben, und wenn die DLL nicht mehr benötigt wird, wird sie mit dem Befehl

```
FreeLibrary(hDll);
```

wieder entfernt.

Wie kann man diese Dynamik nun nutzen? Sowohl die Einbindung der DLLs auch auch der Zugriff auf Methoden erfolgt durch Text, so dass sich Übergabeparameter beim Programmstart, Konfigurations- oder Skriptdateien oder Dialoge mit dem Bediener anbieten. Für die Verwendung mit der Objektfabrik definiert man zweckmäßigerweise einen Satz von Standardmethoden, die in jeder DLL, die man für einzelne Klassengruppen konstruieren kann, die gleichen Bezeichnungen

[10] Aber nochmals ausdrücklich: die Methoden müssen `virtual` sein. Nicht virtuelle Methoden können so nicht bearbeitet werden, da das Anwendungsprogramm deren Adresse nicht kennt.

[11] Auch dies bezieht sich wieder auf Windows als Betriebssystem und Entwicklungssysteme mit GNU-Compilern. Für andere Compiler/Entwicklungssysteme müssen Sie die Details ggf. selbst in Erfahrung bringen, wenn sich mit diesen Programmzeilen nichts tut.

besitzen. Intern können die Methodenadressen auf Feldern von Funktionszeigern abgelegt und jede DLL durch ein Containerobjekt verwaltet werden.

Aufgabe. Implementieren Sie eine Verwaltung aktiver DLLs

Bei Ihren Versuchen werden sie festgestellt haben, dass bei der Initialisierung der Objektfabrik Probleme auftreten. Programm und DLL initialisieren sich jeweils unabhängig voneinander, was dazu führt, dass die Klassen der DLLs nicht im Programm registriert werden. Die Registrierung kann jeweils nur lokal erfolgen, d.h. jede DLL hat intern ihr eigene Objektfabrik.

Um weiterhin eine zentrale Registrierung zu besitzen, verlagert man zweckmäßigerweise die Verwaltung der DLLs ebenfalls in das Modul der Objektfabrik. Die Objektregistrierungen der DLLs müssen die Methoden zum Ermitteln der Klassennamen exportieren, so dass nach Anbinden einer neuen DLL die dort registrierten Klassen in die zentrale Liste übernommen werden können. Zusätzlich ist in der Verwaltung eine Kennung notwendig, ob die Objekte im Zentralprogramm oder einer der DLLs erzeugt werden. Bei Erzeugung in einer DLL ist deren Erzeugungsmethode wie oben im Beispiel beschrieben aufzurufen und das erhaltene Objekt durchzureichen.

Aufgabe. omplettieren Sie die Objektfabrik im beschriebenen Sinn. Ich unterstelle, dass weitere Kodebeispiele nicht notwendig sind.

6.2 Compilezeit-Objektfabriken

Hinter dieser Objektfabrik steckt eine andere Problematik als hinter Laufzeitfabriken. Viele Anwendungen werden je nach Kundenanforderungen und deren Bereitschaft, Lizenzgebühren zu bezahlen, in unterschiedlichen Qualitätsstufen ausgeliefert. Die Qualitätsstufen kann man auf Klassenimplementationen abbilden, was mit folgender Hierarchie pro Klasse verbunden sein kann:

```
class ProtoType {
   ...
   virtual void f()=0;
   ...
};

class SimpleType: publich ProtoType { ... };
class Sophisticated: public ProtoType { ... };
```

In der Anwendung wird grundsätzlich mit dem virtuellen Typ `ProtoType` gearbeitet, und es ist an allen Stellen, an denen Objekte erzeugt werden, dafür zu sorgen, dass der jeweils zur bestellten Version passende Datentyp verwendet wird.

Aufgabe. Das Problem lässt sich – eine Ableitung von `ProtoType` von `FactoryBase` vorausgesetzt – mit der vorhandenen Fabrik lösen. Entwerfen Sie ein Arbeitsmodell.

Umfassen die Varianten mehr als zwei Möglichkeiten und eine Vielzahl von nicht oder nur entfernt zusammengehörenden Klassen, möglicherweise sogar unterschiedliche Anzahlen von Klassen, so ist ein Management ohne Hilfsmittel recht mühsam.

Eine andere Motivation kann darin liegen, sehr häufig (neue) Klassen mit einem variablen Attributspektrum einsetzen zu müssen. Nun ist das Einrichten neuer Klassendefinitionen eine relativ aufwändige Angelegenheit, so dass Techniken, die aus einer Liste von Typen ein Objekt mit entsprechenden Attributinstanzen zu erzeugen vermögen, für Softwarekonstrukteure in einem sehr dynamischen Umfeld recht interessant werden.

Wir werden hier zunächst Klassen- oder Typlisten nebst einfachen Verwaltungsmöglichkeiten entwickeln und anschließend untersuchen, wie dies Typlisten im Rahmen einer Objektfabrik genutzt werden können.

6.2.1 Typlisten

Eine Typliste ist eine Sammlung verschiedener Datentypen, auf die mit Standardtypbezeichnungen zugegriffen werden kann. Eine denkbare Lösung für die Implementation einer Typliste ist

```
struct Nulltype {};

template <class T1=Nulltype,
          class T2=Nulltype, ...,
          class Tn=Nulltype> struct TypeList {
    typedef T1 type_1;
    typedef T2 type_2;
    ...
    typedef Tn Type_n; };
```

mit der Implementation

```
typedef TypeList<int,double,int> myList;
```

Alle nicht belegten Typen sind auf den Spezialtyp **Nulltype** reduziert, so dass jederzeit festgestellt werden kann, welche Typen in einer speziellen Liste tatsächlich benutzt werden.

Dieser Aufbau einer Typliste hat jedoch verschiedene Nachteile.

- Sie muss von vornherein mit so vielen Templateparametern definiert werden, wie im Extremfall benötigt werden.
- In der Regel ist man nicht nur an den Listen, sondern auch an Manipulationen der Listen interessiert, und dies stößt bei dieser Form der Definition ebenfalls an Grenzen.

Wir wählen daher eine andere, beliebig erweiterbare Form, die auch rekursiven Manipulationsalgorithmen entgegenkommt, nämlich die einer verketteten Liste. Das Knotenelement der Liste enthält nur zwei Templateparameter

```
template <typename T1, typename T2=Nulltype>
struct Typelist {
    T1 first;
    T2 second;
};//end struct
```

Die Knotenelemente werden zu einer Art verketteten Liste verbunden, wobei jeder Typelisteintrag einen Typ sowie den Folgeknoten trägt bzw. beim letzten Element den **Nulltype**. Die Generierung erfolgt mit Hilfe geschachtelter Makros

```
#define TYPELIST_1(T1) \
            Typelist<T1,Nulltype>
#define TYPELIST_2(T1,T2) \
            Typelist<T1, TYPELIST_1(T2) >
#define TYPELIST_3(T1,T2,T3) \
            Typelist<T1, TYPELIST_2(T2,T3) >
...
```

Zwar müssen auch hier so viele Makros konstruiert werden, wie Typen in einer Liste auftreten können, jedoch ist dieses Modell leicht erweiterbar. Eine Typliste wird mit Hilfe dieser Makros einfach durch

```
typedef TYPELIST_n(int,double,...) MyList;
```

erzeugt, besitzt formal den Aufbau

```
struct Typelist {
   int  first;
   Typelist {
      double   first;
      Typelist {
         ...
         Nulltype second;
         ...
      }       second
   }    second
};
```

und enthält mit `Nulltype` als Abschluss genau einen Typ mehr, als man tatsächlich benötigt. Warum `Nulltype` in jedem Fall als Abschlussdefinition (man könnte ja auch `Typelist <int, double>` implementieren, wenn man nur diese beiden Typen benötigt) und wie geht man nun mit so einer geschachtelten Struktur um?

6.2.2 Zugriff auf einen Typ in der Liste

Während der Zugriff auf einen Typ im ersten Entwurf recht einfach wäre – man greift einfach auf

```
typename myList::type_x var;
```

zu, ist der direkte Zugriff im bevorzugten Modell zunächst formal recht umständlich. Um mit dem dritten Typ in der Liste ein Objekt zu erzeugen, wäre die Anweisung

```
typename myList::second::second::first var;
```

notwendig.

Wie an der Konstruktion zu erkennen ist, endet eine Typliste immer mit dem Hilfstyp `Nulltype`. Dies erlaubt es uns, auf der Liste Algorithmen mit einem Abbruchkriterium ablaufen zu lassen. Die Anzahl der in der Liste vorhandenen Typen ist durch ein rekursives Template feststellbar; und mit der gleichen Technik auch auf über einen Index auf einen Typ zugegriffen werden.

```
template <class TList> struct Sizeof;

template <> struct Sizeof<Nulltype> {
    enum { value=0 };};

template <class T, class U>
struct Sizeof<Typelist<T,U> > {
    enum { value = 1 + Sizeof<U>::value };
};
```

Der Trick ist leicht zu erkennen, oder? Ist der Templateparameter der Klasse `sizeof` ein Knotentype, wird rekursiv ein Untertyp mit dem zweiten Knotenparameter definiert. Die Kette bricht ab, wenn der `Nulltype` durch die Spezialisierung erkannt wird, und gewissermaßen im Rückwärtsgang wird nun in der Konstanten mitgezählt, um wie viele Rekursionen es sich gehandelt hat.[12]

Man mag sich zunächst etwas wundern, wieso es sinnvoll sein kann, zur Compilezeit die Größe einer ebenfalls zur Compilezeit definierten Typliste festzustellen, denn die sollte doch eigentlich bekannt sein. Im nächsten Abschnitt werden wir jedoch einige Manipulationsalgorithmen für Typlisten angeben, an deren Ende die Größe einer aus verschiedenen Bereichen zusammengeführten Typliste nicht unbedingt trivial und bekannt sein muss.[13]

---

[12] Falls die Liste nicht ordnungsgemäß durch einen `Nulltype` abgeschlossen wird, existiert nur die leere Definition der Klasse, die mangels der Konstanten zu einem Compilerfehler führt.

[13] Diesen einfachen Algorithmus im Hinterkopf können Sie ja mal überlegen, ob die Anzahl der Typen in einer linearen Liste ebenfalls ermittelt werden kann. Ein entsprechender Algorithmus ist auf jeden Fall alles andere als elegant.

Der indizierte Typezugriff erfolgt ebenfalls rekursiv mit Hilfe eines Zahlentemplateparameters:

```
template <class T, int i> struct TypeAt;

template <class T, class U, int i>
struct TypeAt<Typelist<T,U>,i> {
    typedef typename TypeAt<U,i-1>::type type;
};//end struct

template <class T, class U>
struct TypeAt<Typelist<T,U>,0> {
    typedef T type;
};//endif
```

Auch dieser Trick ist sicher leicht zu durchschauen. Da eine Typliste als Templateparameter übergeben wird, wählt der Compiler für die Auswertung die erste Spezialisierung aus, die in der Rekursion nun den zweiten Templateparameter der Typliste einsetzt. Dieser ist in der Regel gemäß Konstruktion ebenfalls wieder eine Typliste, so dass wiederum eine der Spezialisierungen vom Compiler ausgewählt wird. Der Zugriff auf einen bestimmten Typ in der Liste via

```
typename TypeAt<myList,2>::type  var;
```

ist nun strukturell nicht komplizierter als der im linearen Listenmodell, jedoch wesentlich sicherer. Wird versucht, über das Ende der definierten Liste zuzugreifen, so erhält man einen Compilerfehler, da das nun vom Compiler ausgewählte allgemeine Template leer ist.

6.2.3 Algorithmen auf Typlisten

Da Typlisten im Prinzip nichts anderes sind als die Compilerversion verketteter Listen, kann man so ziemlich jeden Algorithmus, der auf diesem Datentyp operiert, auch auf Typlisten loslassen; lediglich das Ergebnis sollte natürlich irgendeinen Sinn im Zusammenhang mit Datentypen ergeben.

Die Algorithmen müssen lediglich rekursiv formuliert werden, wobei Sie den Trick bereits kennen gelernt haben: es existiert eine Spezialisierung des allgemeinen (und meist zum Abfangen von grundsätzlichen Fehlern leeren) Templates für Typlisten, die rekursiv das Template mit dem zweiten Templateparameter der Typliste aufrufen.

Optimierungen, die bei Laufzeitalgorithmen eine beträchtlich Rolle spielen, sind hier uninteressant, da Programmübersetzungen ja nicht den produktiven Arbeitsteil darstellen.[14]

---

[14] Bei größeren Anwendungen kann es durchaus etliche Stunden dauern, ein Programm fertig zu übersetzen, weshalb dies nicht selten in die Nachtstunden verlegt wird. Das Ergebnis liegt dann meist nebst einigen automatischen Testläufen, ob die Software nun tatsächlich das macht, was sie

6.2.3.1 Position eines Typs

Eine erste interessante Frage ist die, ob ein bestimmter Typ in einer Typliste vorkommt. Wir erweitern die Fragestellung ein wenig, indem wir nach dem Index eines Typen in der Liste fragen:

```
template <class TList, typename T> struct IndexOf;
```

Der Algorithmus besitzt zwei Abbruchkriterien.[15] Ist der gesuchte Typ `T` nicht in der Liste vorhanden, endet die Rekursion wieder bei `Nulltype`, und wir initialisieren eine Zählvariable mit -1.

```
template <class T> struct IndexOf<T,Nulltype> {
    enum { value=-1 };
};//end struct
```

Stimmt der gesuchte Typ mit dem ersten der Typliste überein, initialisieren wir mit Null, wobei auf `first` aber gar nicht zugegriffen werden muss, weil der Typ ja bereits in der Templateparameterliste steckt. Die Spezialisierung für diesen Fall ist damit

```
template <class T, class U>
struct IndexOf<Typelist<T,U>,T> {
    enum { value=0 };
};//end struct
```

Die dritte allgemeine Spezialisierung enthält drei Templateparameter und muss die Rekursion fortsetzen, wobei bei der Rückkehr wieder die Rekursionstiefe gezählt wird. Allerdings darf nun `value` nicht inkrementiert werden, wenn festgestellt wird, dass die nächste Rekursionsstufe den Wert `-1` liefert. In der Implementation muss daher eine Fallunterscheidung eingebaut werden, die zunächst prüft, welcher Wert aus der Rekursion zurückgeliefert wird. Um die Ruksursion nicht mehrfach durchlaufen zu lassen, optimieren wir ein wenig mit Hilfe eines Zwischenzählers:

```
template <class T1, class U, class T>
struct IndexOf<Typelist<T1,U>,T> {
private:
    enum { tmp = IndexOf<U,T>::value };
public:
    enum { value = tmp==-1 ? -1 : 1+tmp };
};//end struct
```

---

soll, Morgens bei Arbeitsbeginn der Programmierer vor. Um so wichtiger ist bei solchen Projekten natürlich, sicher zustellen, dass sich der Compiler nicht nach einigen Stunden mit einem Compilerfehler verabschiedet, dessen Ursache in 10 Minuten zu beseitigen ist. Aber das ist ein anderes Thema.

[15] Natürlich eigentlich drei, denn eine falsch definierte Typliste ist ja auch eine mögliche Option. Wie zuvor führt dieser Fall, wie eine Analyse zeigt, auf einen Compilerfehler, während die beiden diskutierten Fälle das natürlich nicht machen.

Da der Compiler die jeweils beste Spezialisierung auswählen muss, bricht der Algorithmus ab, sobald der gesuchte Typ gefunden ist. Allerdings interessiert sich der Algorithmus nicht dafür, ob der Typ mehrfach auftaucht; es wird immer die erste Position geliefert.

Aufgabe. Entwerfen Sie einen Algorithmus, der zählt, wie oft ein vorgegebener Typ in der Liste auftritt.

6.2.3.2 Zusammenfügen von Typlisten

Eine weitere Aufgabe kann darin bestehen, aus zwei Typlisten eine einzige zu machen. Das Ergebnis ist allerdings nicht eine der alten Listen, die erweitert wurde, sondern eine neue Typliste, die den Inhalt der beiden Eingabelisten enthält. Jede Typliste ist eine vollständig eigenständige Klasse, die auch nach der Verwendung in einem Algorithmus weiter existiert und weiter verwendet werden kann. Jede „verändernde“ Operation führt zu einem weiteren neuen Typ. Speicherplatzsparende Algorithmen wie in der Laufzeitversion, in der alte Inhalte einfach überschrieben werden, existieren in der Compilervariante nicht.

Wir implementieren für die Erweiterung die Klasse

```
template <class TList, class U> struct Append;
```

In der Hauptrekursion gehen wir davon aus, dass der erste Templateparameter eine Typliste ist. Der zweite Parameter wird dann an den zweiten Templateparameter der Liste weitergereicht. Als Ergebnis der Rekursion definieren wir einen internen Typ `type`.

```
template <class T, class U, class V>
struct Append<Typelist<T,U>,V> {
    typedef Typelist<T,
                typename Append<U,V>::type> type;
};//end struct
```

Die Nutzung des Algorithmus besteht im Aufruf

```
typedef TYPELIST_n(..) List1;
typedef TYPELIST_m(..) List2;
typedef Append<List1,List2>::type AppList;
```

Wie man sieht, ist der Ergebnistyp eine Typliste, die aus dem ersten Typ als neuem ersten Typ und dem Ergebnis der Verknüpfung mit dem zweiten Typ und der zweiten Liste als zweitem Typ besteht. Die Rekursion durchläuft also zunächst die erste Typliste, und wir müssen eine Spezialisierung angeben, wenn wir auf den Fall

```
template <class T> struct Append<Nulltype,T>
```

treffen. Bei näherer Überlegung stoßen wir auf drei mögliche Fälle:

(1) `T` ein einfacher Datentyp. Das Ergebnis ist in diesem Fall eine einfache Typeliste

```
template <class T> struct Append<Nulltype,T> {
   typedef TYPELIST_1(T) type;
};//end struct
```

(2) `T` ist eine Typliste, die nun selbst zum Ergebnistyp wird

```
template <class U, class V>
struct Append<Nulltype,Typelist<U,V> >{
   typedef Typelist<U,V> type;
};//end struct
```

(3) `T` ist vom Typ `Nulltype`, d.h. an der ersten Liste ändert sich nichts.

```
template <> struct Append<Nulltype,Nulltype>{
   typedef Nulltype type;
};//end struct
```

Aufgabe. Erweitern Sie die `Append`-Klasse um eine weitere Spezialisierung, die auch beim ersten Templateparameter einen einfachen Typ zulässt.

Mit der letzten Spezialisierung erzeugt die `Append`-Klasse immer eine gültige Typliste, auch wenn einer der beiden Parameter nur ein einfacher Typ ist. Ist eine der beiden Listen eine leere Liste, so wird dies durch den Type `Nulltype` und nicht etwa durch `Typelist<Nulltype,Nulltype>` angegeben. Die Zusammenlegung zweier leerer Listen ergibt sinnvollerweise wieder eine leere Liste, also `Nulltype`. Überzeugen Sie sich noch einmal davon, dass sämtliche bislang vorgestellten Algorithmen in diesem Sinn korrekt funktionieren.

6.2.3.3 Entfernen von Typen aus einer Typliste

Auf vergleichbare Art lassen sich einzelne Typen aus einer Klassenliste entfernen.

```
template <class T, class U> struct Erase ;
```

Auch hier müssen wir als interne Typdefinition eine neue, reduzierte Liste erstellen. Der erste Templateparameter ist wieder eine Typliste, und die Rekursion ist abzubrechen, wenn der erste Typ der Typliste mit dem zu löschenden übereinstimmt. Der Rückgabetyp besteht dann im zweiten Typ der Typliste, der entweder eine weitere Typliste oder `Nulltype` ist. Im Negativfall ist der erste Parameter in die neue Liste zu übernehmen, gefolgt vom Ergebnis des Löschvorgangs auf den zweiten `Typelist`-Parameter.

```
template <class T, class U, class V>
struct Erase<Typelist<T,U>,V> {
    typedef Typelist<T,
                typename Erase<U,V>::type> type;
```

```
};//end struct

template <class T, class U>
struct Erase<Typelist<T,U>,T>{
    typedef U type;
};//end struct
```

Abzubrechen ist wieder, wenn das Ende der Liste erreicht ist, ohne den zu löschenden Typ zu finden.

```
template <class T> struct Erase<Nulltype,T> {
    typedef Nulltype type;
};//end struct
```

Wie leicht nachzuprüfen ist, wird der Algorithmus lediglich dann vom Compiler nicht akzeptiert, wenn der erste Templateparameter keine Typliste ist. Das Ergebnis ist entweder eine Typliste oder `Nulltype` als Stellvertreter für eine leere Typliste.

Die Löschfunktion lässt sich einfach auf das Löschen sämtlicher Vorkommen eines Typs in einer Liste erweitern. Wir können alles kopieren und müssen lediglich die Trefferfunktion anpassen:

```
template <class T, class U>
struct EraseAll<Typelist<T,U>,T>{
    typedef typename EraseAll<U,T>::type type;
};//end struct
```

Die letzte Klasse kann nun wiederum eingesetzt werden, um alle Doppeleinträge in einer Liste zu entfernen.

```
template <class T> struct Noduplicates;
```

Wir benötigen dazu einen Zwischentyp, der die Anwendung von `EraseAll` mit dem Kopftyp des aktuellen Knotens als zu entfernender Typ auf den Rest der Liste darstellt. Auf diesen kann wiederum die Entfernung weiterer Typen rekursiv fortgesetzt werden.

```
template <class T, class U>
struct Noduplicates<Typelist<T,U> >{
private:
    typedef typename EraseAll<U,T>::type tmp;
public:
    typedef Typelist<T,
         typename Noduplicates<Tmp>::type> type;
};//end struct

template <> struct Noduplicates<Nulltype> {
    typedef Nulltype type;
};//end struct
```

Der Compiler kann den `public`-Teil erst bearbeiten, wenn er den `private`-Teil gelöst hat, womit die korrekte Funktion des Algorithmus sichergestellt ist.

Aufgabe. Verwandt mit der Aufgabe, einen Typ zu Löschen, ist die Aufgabe, einen Typ durch einen anderen Typ zu ersetzen. Implementieren Sie auf der Grundlage von **Erase** und **EraseAll** die Klassen **Exchange** und **ExchangeAll**, die dies machen.

6.2.3.4 Sortierung von Typlisten

Wenn man sich an das Spektrum der Standardalgorithmen erinnert, bleiben noch Sortieralgorithmen auf Typlisten zu implementieren. Ein Sortieralgorithmus erzeugt eine neue Typliste, die die gleichen Typen wie die Ausgangsliste enthält, jedoch in einer anderen Reihenfolge. Da zwischen Typen keinerlei natürlich definierte Ordnungsrelation besteht, müssen wir diese selbst definieren, können sie aber dann auch so gestalten, wie es für unsere Bedürfnisse am Besten passt.

Für einen Sortieralgorithmus nehmen wir den Bubblesort-Algorithmus zum Vorbild, den wir folgendermaßen aufbauen:

(a) Eine rekursive Klasse prüft den Inhalt einer Typliste gegen eine vorgegebene Klasse und liefert als Rückgabetyp den Typ mit der höchsten (oder niedrigsten) Bewertung gemäß unserer Ordnungsrelation zurück. Dies ist unser Bezugstyp.
(b) In der Typliste wird der Bezugstyp gegen den ersten Typ mit Hilfe des Austauschalgorithmus ausgetauscht.
(c) Der Bezugstyp ist der Kopftyp einer neuen Typliste. Der Schwanztyp ist das Ergebnis der Sortierung des Schwanztypen der Ausgangstypliste, d.h. wir beginnen mit dem zweiten Typ wieder bei a)

Bringen wir das zunächst in eine Klassenform. Das Sortierkriterium definieren wir gleich ebenfalls als Templateparameter, so dass wir es beliebig austauschen können

```
template <class T,
          template <class,class> class ordnung>
struct TypelistSorter;
template <class T, class U,
          template <class,class> class ordnung >
struct TypelistSorter<Typelist<T,U>,ordung> {
private:
  typedef typename ordnung<U,T>::type tmp;
public:
  typedef Typelist
    <tmp,
     typename TypelistSorter
       <typename Exchange<U,tmp,T>::type,
        ordnung>::type
    >  type;
};//end struct
```

```
template <> struct TypelistSorter<Nulltype>{
    typedef Nulltype type;
};//end struct
```

Die Algorithmen `TypelistSorter` und `Exchange` haben wir geschachtelt. Eine Trennung in eine weitere Zwischenklasse für das Ergebnis von `Exchange` ist an dieser Stelle nicht notwendig. Beachten Sie dabei aber, dass `Exchange` nur den ersten gefundenen Austauschtyp austauschen darf und dann abbrechen muss. Sind mehrere gleiche Typen in der Typliste vorhanden, liegen diese nach Abschluss der Operation hintereinander im Ergebnis.

Für die Konstruktion von Ordnungsrelationen wird eine Selektionsklasse benötigt, die aus zwei Klassen eine aufgrund einer logischen Bedingung auswählt. Die Typselektion haben wir bereits an anderer Stelle hinreichend verwendet, so dass diese Implementation recht schnell vonstatten geht:

```
template <class A, class B, bool>
struct Selectfirst {
    typedef A type;
};//end struct

template <class A, class B>
struct Selectfirst<A,B,false>{
    typedef B type;
};//end struct
```

Ordnungsrelation sind ebenfalls Klassen, die auf Typlisten operieren, denn gemäß a) müssen sie ja die Restliste vollständig durchsuchen. Als Beispiel für ein Ordnungskriterium betrachten wird eine Sortierung nach Vererbung. Bei der automatischen Zeigerverwaltung haben wir bereits eine Klasse konstruiert, die ein Vererbungsverhältnis zwischen zwei Klassen feststellen kann. Diese nutzen wir nun, um den logischen Parameter für die Klasse SelectFirst zu generieren. Die folgende Ordnungsrelation zieht die Klassen an der Spitze der Vererbungshierarchie nach vorne.

```
template <class T, class V>
struct LastInHierarchie;

template <class T1, class T2, class V>
struct LastInHierarchie<Typelist<T1,T2>,V> {
public:
  typedef typename
    LastInHierarchie
      <T2,
       typename Selectfirst
         <T1,V,
          FirstInheritsFromSecond<T1,V>::yes
```

```
          >::type
      >::type type;
};//end struct

template <class V> struct LastInHierarchie<Nulltype,V> {
    typedef V type;
};//end function
```

Der Compiler kann in jedem Schritt zunächst die Ordnungsrelation auf der aktuellen Ebene überprüfen und mit dem Ergebnis die Rekursion fortsetzen.

Aufgabe. Weitere Ordnungsrelationen können beispielsweise eine Trennung der Typen nach Klassen und Standardtypen oder eine absolute Reihenfolge der Typen gemäß einer vorgegebenen Typreihenfolge erzeugen. Hierzu wird der Ordnungsrelation eine Vergleichstypliste zugewiesen, und die Sortierung erfolgt beispielsweise mit Hilfe von `IndexOf` eines Typs in dieser Vergleichsliste. Implementieren Sie einen entsprechende Ordnungsrelation.

Aufgabe. Konstruieren Sie einen Algorithmus, der eine Typliste anhand einer vorgegebenen Klasse in zwei Teillisten zerlegt. Eine der Teillisten soll alle Klassen umfassen, die in einem Vererbungsverhältnis mit der Vergleichsklasse stehen, die andere enthält die restlichen Typen der ursprünglichen Liste.

Um sich den Aufwand zu verdeutlichen, die diese Algorithmen im Compiler verursachen, können Sie ja einmal die formale Struktur für eine Typliste mit zwei bis drei Einträgen konstruieren. Grob formuliert wird der Zeitliche Aufwand $O(n^2)$ des Laufzeitsortieralgorithmus im Compileralgorithmus in einen entsprechenden Speicherbedarf für die Zwischentypen übertragen. Unter diesen Gesichtspunkten kann man durchaus Verständnis dafür entwickeln, dass zwischen der Festlegung des Sprachstandards und seiner Regeln und der Marktreife der ersten Compiler, die tatsächlich alles unterstützen konnten, mehr als fünf Jahre lagen.

6.2.4 Arbeiten mit Typlisten

Typlisten und die auf ihnen definierten Algorithmen mögen ja hinsichtlich des Auslotens von Compileralgorithmen ganz interessant sein, aber bis jetzt hat die ganze Arbeit keinerlei Spuren im Laufzeitkode hinterlassen. Was können wir also mit Typelisten in der Entwicklung konkreter Programme anfangen? Die Eingangs diskutierte Möglichkeit, die Typlisten nach Bedarf auszutauschen und in Anwendungsprogrammen mit festdefinerten Instanziierungen mit `TypeAt` zu arbeiten, rechtfertigt nicht gerade den danach betriebenen Auswand der Entwicklung weiterer Compileralgorithmen.

6.2.4.1 Typlisten als Attributlisten

Nehmen wir an, wir wollen eine Klasse mit einer aufgabenbezogenen Anzahl von Attributen verschiedenen Typs erzeugen – oder besser: durch den Compiler erzeugen lassen, indem wir die Attributtypen durch eine Typliste vorgeben. Diese Liste ist rekursiv abzuarbeiten, bis der Abschlusstyp `Nulltype` erreicht ist.

Bei der rekursiven Bearbeitung stellt sich allerdings ein Problem: Attributnamen sind nicht wie Funktionsnamen überladbar. Wir können also nicht im Rahmen der Rekursion auf irgendeine Art und Weise fortlaufen Attribute zu einer Klassen hinzufügen, da uns der Compiler dies übelnimmt (*das gleiche Problem hätten wir sogar mit Funktionsnamen, wenn der gleiche Funktionstyp mehrfach in einer Liste auftritt*). Wir müssen dem also beim Entwurf der Rekursion von vornherein Rechnung tragen.

C++ liefert uns aber glücklicherweise eine Möglichkeit, mehrfach mit gleichen Bezeichnungen klarzukommen, vorausgesetzt, in der Geschichte ihrer Erzeugung gibt es Unterschiede: Mehrfachvererbung. Erbt eine Klasse von zwei verschiedenen Klassen, die Attribute gleichen Namens besitzen, so regt sich der Compiler nicht über die Namensgleichheit auf, sondern verlangt lediglich bei der Referenzierung der Attribute die Angabe, welche Mutterklasse gemeint ist. Wir müssen die Rekursion also so aufbauen, dass eine Mehrfachvererbung mit unterschiedlichen Klassen erfolgt, die eine Auflösung der gleichen Attributnamen erlauben. Die folgende Klasse erfüllt dies Anforderungen:

```
template <class T> struct GenericClass;

template <class T1, class T2>
struct GenericClass<Typelist<T1,T2> >:
    public GenericClass<T1>,
    public GenericClass<T2> {
};//end struct

template <class T> struct GenericClass{
    T value;
};//end struct

template <> struct GenericClass<Nulltype> {};
```

Sie enthält via Vererbung von jedem Typ der Typliste ein Objekt mit Namen `value`.

Allerdings ist der Zugriff auf die Attribute recht eingeschränkt. Der Scope-Operator fällt aus, da die Struktur nicht bekannt ist.[16] Eine erste Möglichkeit liefert ein Typecast:

[16] Bei selbstdefinierten Typlisten ist die Struktur natürlich schon bekannt, aber man muss sich bei Umtypisierungen in der Liste peinlich genau an eine bestimmte Reihenfolge der Typen halten und der Code wird durch die Scopeoperatoren sehr aufwändig, so dass insgesamt wenig für die Programmierung gewonnen ist. Spätestens dann, wenn die Compileralgorithmen nicht nur esotherischen Wert besitzen sollen, sondern auch benutzt werden, trifft die obige Aussage aber zu.

```
typedef TYPELIST_3(char,int,double) tl1;

GenericClass<tl1> gf3;
static_cast<GenericClass<int>& >(gf3).value=3;
```

Der Compiler kann aufgrund des Typecast das passende Attribut herausfiltern. Achten Sie dabei auf den Typecastparameter ! Es heißt `GenericClass<int>&` und nicht `GenericClass<int>` ! Der Grund liegt darin, dass das Objekt der Oberklasse, auf das der Cast angewandt wird, weniger Informationen enthält als das Objekt der Unterklasse, das hier herausgefiltert werden soll. Der Compiler weiß bei der Konstruktion des Castaufrufes (noch) nicht, was im Weiteren noch so alles passiert, insbesondere ob Informationen über das Unterklassenobjekt noch benötigt werden, für deren Speicherung im Oberklassenobjekt gar kein Platz vorgesehen ist (*beispielsweise könnte das Unterklassenobjekt über eine virtuelle Methodentabelle verfügen, die bei der weiteren Bearbeitung des Befehls verwendet werden muss*). Um dieses Problem zu umgehen, erzeugt der Compiler bei einem `GenericClass<int>`-Aufruf mittels des Kopierkonstruktors von `GenericClass` ein temporäres Objekt und kopiert den Inhalt des eigentlichen Objektes darauf. In der Folge wird in der Anweisungszeile

```
static_cast<GenericClass<int> >(gf3).value=3;
```

der zugewiesene Wert auf das temporäre Objekt kopiert und nicht auf das eigentliche Objekt. Erst die Hinzunahme des Referenzoperators im Typecast sorgt nun dafür, dass dies nicht geschieht, sondern tatsächlich auf das Unterattribut der Hauptklasse zugegriffen wird. Man muss sich allerdings darüber im Klaren sein, dass nun gewisse andere, erst zur Laufzeit auswertbare Sachen nicht mehr funktionieren.

Aber auch die Zugriffsmöglichkeit mittels Typecast ist darauf beschränkt, dass `int` genau einmal in der Typliste auftritt. Bei mehreren `int` Typen ist der Compiler genauso überfordert wie bei Fehlen des Typs und mehreren kompatiblen anderen Typen. Bevor wir allerdings die Zugriffskonstrukte einfacher machen und Möglichkeiten schaffen, über den Index anstelle des Typs zuzugreifen, erweitern wir zunächst das Modell.

6.2.4.2 Typlisten in Klassenmodellen

Nehmen wir an, wir wollen nicht (*nur*) auf Attribute zugreifen, sondern wären im Besitz einer Templateklasse, die einzeln mit jeder Klasse der Typliste instanziiert werden soll. Als Beispiel können Sie sich die `Ptr`-Klassen vorstellen, die im Grunde auch nicht vielmehr bewirkt, als eine Klasse mit verschiedenen Attributen zu erzeugen, allerdings die Nutzung virtueller Methoden der Attribute erlaubt. In komplexeren Fällen – die eigentliche Zielrichtung dieser Operation – stellen die Templateklassen selbst Methoden oder auch Attribute zur Verfügung, die alle unter dem Dach der Hauptklasse versammelt werden.

Diese Templateklasse fügen wir als Template-Template-Parameter hinzu:

```
template <class T, template <class> class Modell> struct
Generic;

template <class T1, class T2,
          template <class> class Modell>
struct Generic<Typelist<T1,T2>,Modell>:
    public Generic<T1,Modell>,
    public Generic<T2,Modell> {
};//end struct

template <class T, template<class> class Modell>
struct Generic: public Modell<T> {
};//end struct

template <template <class> class Modell>
struct Generic<Nulltype,Modell> {};
```

Die Arbeitsweise lässt sich folgendermaßen verstehen: die generische Klasse erhält eine Typliste und eine Policy-Klasse als Parameter und erbt ihrerseits von zwei eigenen Template-Instanzen, die von den beiden Typen des ersten Typlistenknotens definiert werden. Während der zweite Typ als Typliste rekursiv das Ganze wiederholt oder als **Nulltype** die Rekursion beendet, greift der erste Typ auf eine von der Policy-Klasse erbende Spezialisierung zu. Die Policy-Klasse kann nun eine Instanz eines Objektes des jeweiligen Typs enthalten, so dass bei Instanziierung der **Generic**-Klasse mit einer Typliste tatsächlich von jedem Typ ein Attribut erzeugt wird. Die Policy-Klasse kann aber auch, je nach Bedarf in der Anwendung, weitere Aufgaben übernehmen.

Nehmen wir als Beispiel an, eine Policy-Klasse **Holder** definiere ein Attribut mit dem Namen **_val**. Vorausgesetzt, jeder Typ tritt in der Typliste nur einmal auf, kann der Zugriff auf die Objekte mittels eines TypeCast erfolgen:

```
typedef TYPELIST_3(int,double,string) myList;
Generic<myList,Holder> obj;
static_cast<Holder<string> >(obj)._val = "Hallo";
```

Da `Holder<string>` als Teilklasse nur einmal in **obj** vorhanden ist, kann der Compiler den **static_cast** ausführen und das passende Attribut bereitstellen.

Damit haben wir das gleiche Ergebnis erreicht wie im ersten Beispiel, nur dass in der Anwendung nun nicht auf die generische Klasse Bezug genommen wird, sondern auf die Policy-Klasse. Allerdings ist diese Zugriffsmethode technisch noch ebenso unbefriedigend wie zuvor.

Anstelle des aufwändigen `static_cast`-Zugiffs, der immer noch einen Klassennamen enthält, schaffen wir uns zunächst eine einfachere Möglichkeit mit Hilfe einer Methode `Bind`, die nur noch das generische Objekt und den Attributdatentyp enthält und folgende Form besitzen soll

```
Bind<string>(obj)._val= "Hallo"
```

Der Rückgabewert der Methode **Bind** muss in diesem Fall eine Referenz auf ein `Holder<string>`-Objekt sein (wir simulieren mit der **Bind**-Methode somit exakt die Funktion des **static_cast**), und sofern der Compiler mehrere **Bind**-Methoden kennt, kann er die passende heraussuchen. Wir müssen ihn nur noch überreden, diese **Bind**-Methoden (bei Bedarf) selbst zu erzeugen.

Hierzu erweitern wir die **Generic**-Klasse(n) um eine interne öffentliche Hilfsklasse, die den Datentyp der Policy-Klasse enthält:

```
template < ... > class Generic {
public:
   template <class Z> struct Binder {
       typedef Modell<Z> type;
   };//end struct
   ...
```

Die `Bind`-Methode lässt sich damit folgendermaßen definieren, zuzüglich einer zweiten Version für den **const**-Fall):

```
template <class T, class U>
typename U::template Binder<T>::type& Bind(U& obj)
{ return obj;}
```

Wie funktioniert nun die Auflösung des Rückgabetyps? Das Objekt **obj** ist vom Typ `Generic<myList,Holder>`, der hier den Template-Parameter **U** spezifiziert. Der Funktionsaufruf selbst muss einen Zugriff auf ein Objekt des Typs **string** ermöglichen, das hier den Parameter **T** spezifiziert. Der Zugriff wird, wie beim **static_cast**, dadurch ermöglicht, dass die zugehörende Teilklassen aufgelöst wird. Hierzu sind die verschiedenen **Holder**-Instanzen der **Generic**-Klasse zu durchsuchen, bis der passende Typ, in diesem Fall `Generic<myList, Holder>::Holder<string>` gefunden ist.

Der ungewohnte Ausdruck `U::template Binder<T>` hat dabei den gleichen Hintergrund wie das dem Programmierer geläufigere Konstrukt `typedef typename U::type`. Da der Compiler zunächst nur eine Syntaxprüfung des Quelltextes durchführt, die tatsächlichen Zusammenhänge aber erst im zweiten Durchlauf feststellt, ist im Standard festgelegt, dass eine Konstruktion mit dem Scope-Operator im Sinne von **Klasse::Attribut** oder **Klasse::Methode** zu interpretieren ist. Bei einem **typedef** führt dies zu einem Syntaxfehler, da Attribute nicht in Typdefinitionen verwendet werden dürfen, in einem Template-Ausdruck führt dies ebenfalls zu einem Syntaxfehler, da die `< >` - Klammern des Templateausdrucks als Vergleichsoperatoren interpretiert werden. Durch die Schlüsselworte **typename** und **template** an den kritischen Stellen wird der Compiler über die korrekte Bedeutung des Scope-Operators informiert und kann nun die richtige Syntaxprüfung vornehmen.

In Summe ist das zwar nun aus Sicht der Programmiertechnik schon ein recht befriedigender Zwischenstand, denn für den Zugriff auf ein Attribut eines beliebigen Objektes benötigen wir nun nur noch das Objekt selbst sowie den Typ des gewünschten Attributs, aber nicht mehr die komplette Klassendefinition des Objektes (die darf der Compiler nun selbst herausfinden). Aber ...

6.2.4.3 Indizierte Zugriffe auf die Modelle

... die vorgestellten Zugriffsoperationen funktionieren nur, sofern der benötigte Typ nicht mehrfach in der Typliste vorhanden ist, und bei Verwendung kompatibler Typen kann es passieren, dass der von Compiler ausgewählte exakte Typ gar nicht der ist, den der Anwendungsprogrammierer im Visier gehabt hat, oder der Compiler gar mit einem Compilerfehler die Mitarbeit verweigert. Derartige Unstimmigkeiten können durch einen indizierten Zugriff umgangen werden:

```
Field<2>(obj)._val="Hallo";
```

Wie indizierte Zugriffe zu realisieren sind, haben wir bei `TypeAt` bereits herausgefunden. Wir erweitern zur Übertragung der ermittelten Techniken auf das Klassenmodell die Klassendefinitionen ein weiteres Mal:

```
typedef Modell<T1> first;
typedef Generic<T2,Modell> second;
```

Die erste Definition ist bei beiden Spezialisierungen der Generic-Klasse notwendig, die zweite nur bei der ersten.

> **Aufgabe.** Wie bei `TypeAt` kann nun eine Klasse `GenTypeAt` konstruiert werden, die den Typ liefert. Dies sei Ihnen überlassen.

Der Rückgabetyp des Funktionsaufrufs der Methode **Field** liegt damit fest. Etwas mehr Kopfzerbrechen macht allerdings die Ausführung der **Field**-Methode selbst. Methoden können nämlich nicht, wie Klassen, mittels ganzzahliger Template-Parameter rekursiv gemacht werden. Wir können hier nur den Umweg gehen, in der Rekursionstiefe Funktionen mit verschiedenen (rekursiven) Parametertypen zu konstruieren. Dies führt schließlich zu der Konstruktion

```
template <int i,class T>
typename GenTypeAt<T,i>::type& Field(T& obj) {
  return Indexer(obj,
        Type2Type<typename GenTypeAt<T,i>::type>(),
                 Int2Type<i>());
}//end function
```

mit der rekursiven Hilfskonstruktion

```
template <class T, class R, int i>inline
R& Indexer(T& obj, Type2Type<R> tt, Int2Type<i>){
    typename T::second& sobj=obj;
    return Indexer(sobj,tt,Int2Type<i-1>());
}//end function

template <class T, class R>
inline R& Indexer(T& obj, Type2Type<R>, Int2Type<0>){
    typename T::first& sobj=obj;
    return sobj;
}//end function
```

Die beiden Hilfstypen

```
template <class T> struct Type2Type {
    typedef T type;
};//end struct

template <int i> struct Int2Type {
    enum { value=i };
};//end struct
```

spielen dabei eine wesentliche Rolle. `Int2Type` erzeugt in jeder Rekursionsstufe einen anderen Datentyp und zwingt den Compiler so, jeweils eine neue überladene Methode zu konstruieren, mit der die Rekursion fortgesetzt werden kann. Dabei ist `Int2Type` ein leerer Datentyp, von dem nach dem Übersetzen des Programms in Maschinenkode nichts mehr übrig bleibt, obwohl er als Übergabeparameter in der Schnittstelle der Funktion vorhanden ist.[17] Die Rekursion endet dann bei der Spezialisierung `Int2Type<0>`, womit diese rekursive Typisierung die Rolle des ganzzahligen Parameters in den klassengestützten Compileralgorithmen übernommen hat.

`Type2Type` erlaubt das Durchreichen des gewünschten Ergebnistyps in Form eines leeren Objektes durch alle Rekursionsstufen, verdient aber noch eine kleine Erläuterung. Wir hätten auch eine Definition in der Form

```
R& Indexer(T& obj, R, Int2Type<0>)
```

vornehmen können, also nach Auflösung des Rückgabetypen ein Objekt dieses Typs durch die Rekursion durchreichen können, was ebenfalls zu einer korrekten Auflösung führt, da ja schon `Int2Type` die notwendigen neuen Typen liefert. Allerdings ist **R** in unserem Beispiel zumindest vom Typ `Holder<string>` und kann in anderen Anwendungen noch wesentlich komplexer ausfallen. In jeder Rekursionsstufe müsste nun ein Objekt dieses Typs generiert werden, das aber innerhalb der Methode gar nicht verwendet wird. Der Compiler kann die Erzeugung eines nicht verwendeten Objektes auch bei aktivierter Optimierung aber nicht einfach unterdrücken, da zum Einen die Optimierungsarbeit durch die vielen Zwischenstufen überaus komplex ist und zum Anderen im Konstruktor und Destruktor eines temporären Objektes im Hintergrund ja möglichweise noch weitere, für die Gesamtfunktion wesentliche Abläufe stattfinden. Das Ganze wäre also im Regelfall mit einem ziemlichen Aufwand verbunden, sofern nicht gesperrte Konstruktoren überhaupt eine Übersetzung unterbinden.

Der Typ `Type2Type` erlaubt nun ebenfalls die problemlose Weitergabe des Rückgabetyps, ist aber im Gegensatz zu `R` ein leerer Datentyp ohne Attribute,

---

[17] Der Typ enthält in dieser Definitionsvariante seine Definitionszahl als Konstante, die jedoch für diese Anwendung nicht benötigt wird und auch fortgelassen werden könnte. In anderen Anwendungsfällen ist der Rückgriff jedoch durchaus brauchbar, weshalb er hier berücksichtigt wurde. Unabhängig davon besitzt die Klasse keinerlei Attribute oder virtuelle Definitionen, und folglich gibt es für den Compiler bei einem Aufruf auch nichts auf den Stack zu übertragen.

belastet also wie `Int2Type` das Laufzeitsystem nicht, obwohl formal Objekte durchgereicht werden.

Fassen wir zusammen. Typlisten bieten die Möglichkeit, Typsammlungen für verschiedene Anwendungen auf einfache Art und Weise zusammenzustellen und zu verwalten. Die `Generic`-Klassen ermöglichen es, Anwendungsklassen durch Auflistung der Attributtypen zu generieren oder eine Template-Klasse mit einem ganzen Satz von Typen zu instanziieren. Die Template-Methoden `Bind` und `Field` erlauben es schließlich, gezielt auf bestimmte Instanzen der Typen in der Liste zuzugreifen, wobei die Feinheiten der Instanzen in den `Generic`-Klassen durch zusätzliche `Modell`-Klassen gesteuert werden können.

Zugegeben, das Ganze ist recht abstrakt, und wenn man wie bisher nur sehr eng an verschiedenen Aufgabenstellungen entlang programmiert hat, fällt einem vermutlich so schnell nichts Konkretes ein, bei dem diese Techniken sinnvoll eingesetzt werden könnte. Deshalb noch ein ...

6.2.5 Beispiel: Compiletime-Objektfabrik

Als praktisches Anwendungsbeispiel entwerfen wir nun eine zur Laufzeitfabrik vergleichbaren Struktur, einer Compilezeitfabrik. Einsatzziel einer solchen Fabrik ist eine Anwendung, die in einer Light-Version und einer Professional-Version erscheinen soll, was bei vielen Anbietern von Software eine normale Marketingstrategie ist. Beide Versionen sollen aktuellen Trends folgen, insbesondere soll die Light-Version, die zum Einsatz der Vollversion anregen soll, letztere auch bei neuen Versionen widerspiegeln. Ein gleichzeitige Pflege von zwei kompletten Programmversionen ist allerdings viel zu kostspielig. Und hier kommt nun eine Compilezeit-Objektfabrik zum Einsatz.

Für den Softwarehersteller ist das zunächst ein Designproblem. Eine Anwendung besteht aus einem Programm, das die Hauptabläufe beinhaltet, sowie einer ganzen Reihe von Objekten, die im Rahmen der Abläufe erzeugt und aufgerufen werden. Das Steuerprogramm ist in allen Programmversionen das Gleiche, die Klassen, von denen die Objekte abgleitet werden, sind aber von Version zu Version auszutauschen.

Das Designmodell sieht also so aus, dass das Steuerprogramm unter Nutzung einer Klassenbibliothek mit den Typen

```
class:  A, B, C, ..
```

implementiert wird, in den verschiedenen Versionen aber die erbenden Klassen

```
A_light, B_light, ...
A_prof,  B_prof , ...
```

für die Erzeugung der Objekt einzusetzen sind, wobei unerheblich ist, welche Funktionalitäten die Grundtypen beinhalten oder `A_prof` von `A_light` erbt usw.

Um die passenden Objekt zu erzeugen, verwendet das Steuerprogramm eine Instanz einer Fabrikklasse. Diese verfügt über eine Template-Methode `Create`, die ein passendes Objekt erzeugt.

```
Factory<..> fo;
...
A* a= fo.Create<A>();
...
```

Die Fabrikklasse wird mit einer für die Programmversion notwendigen Typliste erzeugt:

```
typedef TYPELIST_n(A_prof,B_prof,C_light,...) tl;
Factory<tl> fo;
```

Die Typen in der Typliste sind mit den Typen, mit denen die `Create`-Methode instanziiert werden, nicht verträglich, so dass unsere bisherigen Algorithmen für Typlisten nicht greifen. Wir erweitern daher unsere Algorithmenliste um einen, der feststellt, welche Typ in der Liste von Typ der `Create`-Funktion erbt. Wir erreichen dies wieder durch einen internen Auswahltyp

```
template <class TL, class A> struct InheritsFrom;

template <class T1, class T2, class A>
struct InheritsFrom<Typelist<T1,T2>,A> {
private:
    template <class U1, class U2, class B, bool>
    struct temp {
        typedef typename
                 InheritsFrom<U2,B>::type type;
    };//end struct

    template <class U1, class U2, class B>
    struct temp<U1,U2,B,true> {
        typedef U1 type;
    };//end struct
public:
    typedef typename
       temp<T1,T2,A,
            FirstInheritsFromSecond<T1,A>::yes
           >::type type;
};//end struct
```

Die Klasse liefert als Ergebnis den ersten Typ in der Liste, der mit dem übergebenen Typ in einem Vererbungsverhältnis steht, oder einen Compilerfehler, wenn kein passender Typ in der Liste gefunden wird.

Aufgabe. Die Ermittlung des Kindtyps in der Typliste führt zu der Einschränkung, dass gemischte Komplexitäten nicht möglich sind. Mit der Liste

`TYPELIST_n(A,A_light,B_prof,A_prof,...)`

würden `Create<A>()` immer ein Objekt des Typs `A` erzeugen. Implementieren Sie Kontrollklassen zur Überprüfung, ob innerhalb einer Liste keine weiteren Vererbungsbeziehungen vorliegen. Sollten zwei Klassen in einer Liste in einer Vererbungsbeziehung untereinander stehen, wobei die Reihenfolge gleichgültig ist, soll diesmal ein Compilerfehler ausgelöst werden.

Das Designmodell muss die Eindeutigkeit der Typlisten bezüglich der Vererbungshierarchien sicherstellen, wenn dieses Fabrikmodell verwendet werden soll.

Die Erzeugung der Objekte ist nun sehr einfach:

```
template <class T> struct InherictanceFactory{
    template <class A>
    A* Create() const {
        typedef typename
               InheritsFrom<T,A>::type type;
        return new type();
    }//end function
};//end class
```

Treten in der Typliste mehrere Klassen auf, die voneinander erben, und ist dies vom Design her nicht vermeidbar, so ist das Fabrikmodell nicht brauchbar. Es lässt sich jedoch durch ein Modell ersetzen, dass mit zwei Typlisten arbeitet, von denen eine die Basisklassen und die zweite die zu verwendenden Klassen enthält. Mit den bekannten Techniken entwirft man nun schnell folgende Fabrikklasse für diesen Fall. `TM` ist die Modelltypliste, `TI` die Liste der tatsächlich zu implementierenden Typen.

```
template <class TM, class TI>
struct CorrespondingFactory {
  template <class A>
  A* Create() const {
    typedef typename
      TypeAt<TI,IndexOf<TM,A>::value>::type type;
    return new type();
  }//end function
};//end class
```

Dieses Modell ist allerdings auch nur für den Fall geeignet, dass ein gegebene Basisklasse immer durch eine bestimmte abgeleitete Klasse zu ersetzen ist. Verschiedene Ersetzungsmodelle lassen sich nur noch durch ein Indexsystem realisieren, was allerdings das Problem mit sich bringt, dass obsolete Klassen nicht einfach aus der Typliste gestrichen werden dürfen, wenn nicht das gesamte Steuerprogramm angepasst werden soll. Sollte dieser Anwendungsfall also tatsächlich eintreten, so sollte man besser von einem zu behenden Designfehler ausgehen.

Aufgabe. Eindeutige Zuordnungen lassen sich auch in langen Typlisten sicherstellen, wenn aus der Modellliste Doppeleinträge entfernt und anschließend die Längen der Modell- und der Implementationsliste verglichen werden. Entwerfen Sie ein Kontrollelement für die Fabrikklasse.

6.3 Applets und Sandbox in C++

6.3.1 Das Sandbox-Konzept

Das letzte Kapitel zum Thema „Objektfabrik“ schweift ein wenig ab. Zwar werden auch hier einer fertigen Programmumgebung Objekte mit neuen Eigenschaften hinzugefügt, jedoch sollen diese nicht aus der fertigen Umgebung heraus bedient werden, sondern sie bringen ihre Bedienumgebung in der einen oder anderen Form mit und sollen bereits bestehende Ressourcen nutzen. Gewissermaßen stellt die fertige Umgebung mehr oder weniger eine Laufzeitbibliothek oder Laufzeitumgebung dar, aus der sich das neue Objekt bedient bzw. bedienen muss. Dabei soll aber sichergestellt werden, dass bestimmte Objekte auch nur bestimmte Methoden der Laufzeitumgebung nutzen können und der Zugriff auf nicht freigegebene Methoden verhindert wird.

Wenn man diese Randbedingungen genauer analysiert, stellt man schnell fest, dass es sich hier weniger um C++, sondern mehr um Java-Philosophie handelt. Wir analysieren hier, wie diese Java-Spezifika in C++ zu realisieren sind, müssen aber von vornherein bemerken, dass es in C/C++ Umgebungen in der Regel nicht möglich ist, das Javakonzept konsequent so umzusetzen, dass ein Betrug ausgeschlossen ist. Das Ziel ist hier also zunächst zu erkennen, wie Java seine Ziele erreicht, aber möglicherweise stellen sich dem Einen oder Anderen von Ihnen ja doch in der Zukunft auf anderen Gebieten Aufgaben, bei denen man sinngemäß vorgehen kann. Doch zunächst einmal zu Java.

Die Programmiersprache Java wurde ursprünglich entwickelt, um Programme über das Internet auf beliebige Zielsysteme zu übertragen und dort laufen zu lassen. Das ist im Allgemeinen ein Problem, da man als Empfänger eines Programms nicht sicher sein kann, dass der Absender es auch ehrlich meint. Dieses Risiko sollte mit dem Java-Konzept ausgeschaltet werden, indem die übertragenen Programme in einer abgesicherten Umgebung laufen, in der sie keinen schlimmen Unfug anstellen können.

Grundlage eines Sicherheitskonzepts ist zunächst eine Analyse, was unter „schlimmen Unfug“ zu verstehen ist. In erster Linie gilt es zu verhindern, dass auf Dateien des Zielsystems zugegriffen werden kann, neue Dateien erzeugt werden, Datenübertragungsverbindungen eingerichtet oder Programme gestartet werden. Alle Klassen, die Gefahren einer Manipulation des Zielsystems oder der Kompromittierung vertraulicher Daten beinhalten, sind von den Java-Entwicklern an einen Sicherheitsmanager gebunden worden, den sie vor Ausführung einer kritischen Aktion um Erlaubnis fragen müssen. Wird diese verweigert, wird die Aktion nicht ausgeführt und ggf. eine Ausnahme geworfen.

Der Sicherheitsmanager wiederum ist ein System-Singleton, in dem Sperrvermerke notiert werden. Um eine geordnete Systemfunktion zu gewährleisten – Java hat sich ja gewissermaßen zu einer Eier legenden Wollmilchsau entwickelt, mit der alles gemacht werden kann, und die lokalen Java-Programme dürfen natürlich keiner Funktionsbeschränkung unterliegen – , müssen alle externen Programmmodule von der Klasse „Applet“ erben. Das Laufzeitsystem sorgt dafür, dass nur solche externen Module ausgeführt werden, die von Applet erben, und Applet selbst sorgt als Basisklasse dafür, dass der Sicherheitsmanager in den geblockten Modus umgeschaltet wird und der Anwender während des Ablaufs des Applets keine Möglichkeit hat, dies zu ändern.

6.3.2 Sandbox in C++ Umgebungen

Wir stellen hier nun ein vergleichbares Konzept für C++ vor. Der (externe) Anwender kann wahlweise

- Quellkode liefern, der lokal mit übersetzt wird,
- Objektkode, der zusammen mit eigenen C++ Modulen vom Linker zu einem fertigen Programm verbunden wird, oder
- Bibliotheksmodule, die zur Laufzeit hinzugelinkt werden.

Da C++ nicht wie Java von vornherein auf die Einschränkung der Rechte bestimmter Anwendergruppen eingerichtet ist, sind einige Randbedingungen zu schaffen bzw. einzuhalten. Hierzu gehört zunächst, dass alle Klassen mit kritischen Operationen, beispielsweise `stream`-Klassen, die Dateizugriffe erlauben, überschrieben werden müssen. Die aufgerufene Funktion darf nur dann ausgeführt werden, wenn der Sicherheitsmanager dies zulässt:

```
void SecureAction::action(char const* s) const {
    if(!SecurityManager::is_locked()){
        // Ausführung der normalen Funktion
    }else{
        // Ausnahme oder Abbruch
    }//endif
}//end function
```

In der Compilerumgebung für die Entwicklung des (externen) Anwenderkodes dürfen nur die Headerdateien der überschriebenen Klassen bekannt sein, nicht jedoch die normalen Bibliotheksklassen. Für C++ ist die Entfernung der Header in der Regel ausreichend, da der Compiler in diesem Fall die Namen nicht so umwandeln kann, dass eine korrekte Einbindung der Bibliotheken erfolgt.

Diese Vorgabe ist jedoch nicht geeignet, auch sauberen Code zu erhalten, denn der (externe) Programmierer kann auf `extern`-deklarierte Klassen und Methoden zurückgreifen, die vom Compiler nicht überprüft werden, den Linke aber möglicherweise zum Einbinden der unzulässigen Klassen und Methode veranlassen. Stellt er

nicht nur Quellkode, sonder Objektkode oder Bibliotheksmodule zur Verfügung, besteht auch die Möglichkeit, dass er unzulässige Aufrufe direkt selbst einbindet.

Man kann somit nicht verhindern, dass sich der externe Entwickler nicht an die Regel hält. Eine Absicherung ist nur möglich, indem man

- auf dem eigenen System nur spezielle Bibliotheken zur Verfügung stellt, die keine kritischen Methoden enthalten, sowie
- den erhaltenen Code auf unzulässige Systemaufrufe überprüft und diese entfernt.

Beides ist natürlich sehr aufwändig und behindert auch die Entwicklung anderer Anwendungen, was der tatsächlichen Installation einer solchen Sicherheitslösung mit C++ Mitteln enge Grenzen setzt.

Um die Kontrolle beim System zu belassen, ist die `main`-Funktion ebenfalls Bestandteil der Systemmodule. Stellt der Anwender eine weitere `main`-Funktion in seinem Quellkode zur Verfügung, unterbricht der Linker aufgrund der hierdurch auftretenden Mehrdeutigkeiten die Ausführung. Die Anwenderfunktionalität kann so in beliebige Entwicklerkontexte eingebunden werden, ohne dass der Anwender diese umgehen kann.

6.3.3 Die Applet-Basisklasse

Damit der Anwender seinen Code einbinden kann, muss er seine Klassen von der Klasse `Applet` erben lassen. `Applet` stellt im Konstruktor Registrierfunktionen zur Verfügung. Diese bleiben jedoch für den Anwender unsichtbar, da auch `Applet` nur eine Header-Datei zur Verfügung stellt, der Code jedoch in einem Modul gekapselt ist.

```
class Applet{
public:
    Applet(int priority=0);
    virtual ~Applet();

    virtual void run(string& param)=0;
    void re_run(int priority);
private:
    Applet();
    Applet(Applet const&);
    static bool action(string& param);
    static void re_run_all();
    friend int main(int argc, char *argv[]);
};//end class
```

Der Anwender kann beliebig viele Klassen von `Applet` ableiten und muss jeweils lediglich die Methode `run(string&)` überschreiben. Von jeder Klasse muss er in

seinem Modul so viele statische Variable erzeugen, wie er für seine Aufgaben benötigt. Die Reihenfolge der Variablen kann über den Parameter `priority` gesteuert werden.

```
class MyApplet: public Applet {
public:
    MyApplet(): Applet(0){...}
    ~MyApplet(){...}
    void run(string& pa){...}
} myApp1 ;
```

Im Modul der Klasse `Applet` verwaltet ein Singelton-Containerobject die einzelnen Variablen. Jeder Eintrag des Containers enthält einen Zeiger auf das Anwenderobjekt sowie die zugehörige Priorität.

```
struct AppObject {
    int priority;
    Applet* app;
    AppObject(int p, Applet* a){...}
    AppObject(AppObject const& ao){...}
    ~AppObject(){}

    bool operator==(AppObject const& ao) const {
        return app==ao.app;
    }//end function

    bool operator<(AppObject const& ao) const {
        return priority<ao.priority;
    }//end function
};//end struct

typedef vector<AppObject> AppContainer;
```

Der Konstruktor von `Applet` registriert das Objekt im Container, der Destruktor löscht es wieder, so dass auch temporäre Objekte korrekt verwaltet werden (*bzw. Versuche, mit ungültigen Zeigern zu arbeiten, abgefangen werden*).

```
Applet::Applet(int priority){
    appContainer().push_back(
              AppObject(priority,this));
    sort(appContainer().begin(),a
         ppContainer().end());
}//end function

Applet::~Applet(){
    appContainer().erase(
        find(appContainer().begin(),
             appContainer().end(),
             AppObject(0,this)));
}//end destructor
```

Die Methode `action` wird in der `main`-Funktion aufgerufen und arbeitet die Applets des Containers ab (*Aufruf der Methode* `run`(. .)). Dazu wird über den Sicherheitsmanager die Ausführung der abgesicherten Funktionen und die Priorität auf einen Wert <0 zurückgesetzt. Applet mit negativen Prioritäten werden nicht ausgeführt, so dass jedes Applet in der Regel nur einmalig ausgeführt wird. Mittels der Methode `re_run`(. .) kann sich das Applet jedoch erneut mit einer bestimmten Priorität reaktivieren. Es kann jedoch nur sich selbst reaktivieren, nicht andere Applets im Container. Die `main`-Funktion kann jedoch mittels der Methode `re_run_all`() alle Applets in der zuvor gültigen Priorität beliebig oft reaktivieren. Die Methode `run`(. .) besitzt einen `string`-Parameter mit Referenzübergabe. Hiermit können beliebige Daten zwischen den Applets, zwischen Applet und System oder zwischen Applet und Laufzeitumgebung ausgetauscht werden (*siehe z.B. Parameterstrings*).

6.3.4 Der Security-Manager

Der Sicherheitsmanager ist ähnlich konzipiert wie der Ausnahmemanager (`class NotifyHandler`).

```
class SecurityManager {
public:
    enum Lockmode { unlock , lock ,
                    lock_question ,
                    lock_unrevoke };
    SecurityManager(Lockmode lm);
    ~SecurityManager();
    static bool is_locked();
private:
    SecurityManager();
    SecurityManager(SecurityManager const&);
    Lockmode old_mode;
    bool mode_set;
};//end class
```

In einer modulinternen statischen Variablen ist die jeweilige Freigabe abgelegt. Sie wird bei Erzeugen eines Manager-Objektes im Objekt abgespeichert und durch die neue Freigabe ersetzt. Zu Beginn eines Blockes wird ein Manager-Objekt erzeugt, bei Verlassen des Blockes wird automatisch der alte Zustand wiederhergestellt (*siehe Ausführungsmethode für alle Applets als Beispiel*).

```
bool Applet::action(string& param){
    AppContainer::iterator it;
    bool active=false;
    SecurityManager
            sm(SecurityManager::lock_unrevoke);
```

```
    try{
        for(it=appContainer().begin();
               it!=appContainer().end();it++){
            if(it->priority>=0){
                it->priority-=10001;
                it->app->run(param);
                if(it->priority>=0)
                    active=true;
            }//endif
        }//endfor
    }catch(extended_exception& e){
        active=false;
    }//endtry
    return active;
}//end function
```

Die Header-Datei des Sicherheitsmanagers kann dem Anwendermodul zur Verfügung gestellt werden, so dass auch aus dem Anwendermodul eine Freigabe von Sicherheitsfunktionen angefordert werden kann. Der Modus `lock_unrevoke` sorgt dafür, dass weitere Instanzen des Sicherheitsmanagers an der Blockierung der Sicherheitsfunktionen nichts ändern können, der Modus `lock_question` dient dazu, im Rahmen einer Anfrage an das System oder die Laufzeitumgebung unter gewissen Voraussetzungen eine Freigabe zu genehmigen (*Java erlaubt dies durch interaktive Anfrage an den Bediener ebenfalls*). Soll zwischen verschiedenen Sicherheitsstufen für unterschiedliche abzusichernde Objekte unterschieden werden, so ist die Liste der Zustände entsprechend zu erweitern.

Außerhalb der `main`-Methode ist der Status in der Regel `lock_unrevoke`. Dies ist notwendig, da der Anwender natürlich auch innerhalb von Konstruktoren und Destruktoren grundsätzlich die Möglichkeit hat, abgesicherte Methoden aufzurufen. Die Konstruktoren von statischen Objekten werden jedoch vor Aufruf der `main`-Methode ausgeführt, die Destruktoren nach deren Verlassen. Werden in diesen Programmbereichen auch statische Objekte im System erzeugt, die sichere Objekte benötigen, so kann entweder der Sicherheitsmanager grundsätzlich vor dem Anwender verborgen werden (*Header-Datei nicht zugänglich*) oder eine zusätzlich verborgene Schnittstelle für Systemfunktionen implementiert werden, die einen Zugriff auf abgesicherte Methoden auch im Status `lock_unrevoke` erlaubt.

6.3.5 Aufrufe und Probleme

Wie wird nun ein Appletkode aktiviert? Wir können hier mehrere Methoden vorsehen.

- Für eine bedarfsgesteuerte Aktivierung hinterlegt der Entwickler Erzeugungsmethoden für seine Objekte in einer Objektfabrik, die die Klasse `Applet` als Basisklasse besitzt.

  Die eigentliche Anwendung prüft nun, ob für bestimmte Aufgaben Appletklassen in der Objektfabrik vorhanden sind und erzeugt sich bei Bedarf die passenden Objekte. Diese Überprüfung kann

  - interaktiv mit dem Anwender durch eine entsprechende Menueschnittstelle oder
  - durch ein Skript, das von der Anwendung interpretiert wird,

  ausgeführt werden. Bei Java-Anwendungen, die in einem Browser ausgeführt werden, übernehmen dies oft HTML-Elemente, die bei Aktivierung bestimmte Java-Objekte ausführen.
- Bei einer permanenten Aktivierung hinterlegt der Entwickler nicht die Erzeugungsmethoden in einer Objektfabrik, sondern fertige Objekte in einem Objektcontainer. Diese werden von der Anwendung bei Ereignissen angesprochen, und die Applets müssen untereinander ausmachen, wer für die Behandlung des Ereignisses zuständig ist. Mechanismen für diesen Fall behandeln wir in einem späteren Kapitel.

Wie man sieht, bedarf es neben dem Appletkode selbst noch einiger weiterer Arbeit, um Anwendungen mit sinnvollen Funktionen zu erzeugen. Auf konkrete Aufgaben sei an dieser Stelle verzichtet.

Probleme können durch das Zeitverhalten der Appletobjekte auftreten, da diese die Kontrolle nach einer gewissen Zeit wieder an das Hauptprogramm zurückgeben müssen, das aber möglicherweise nicht oder nicht rechtzeitig tun. Ein Ausweg aus Blockaden durch Fremdobjekte sind Threads, jedoch muss dann auch der Appletkode damit kompatibel sein. Diese Mechanismen werden wir in einem späteren Kapitel untersuchen.

Auch Ausnahmen können Probleme verursachen, wenn von der Anwendung andere Konzepte verfolgt werden als vom Applet. Hier sei auf das bereits durchgearbeitete Kapitel über Ausnahmen verwiesen.

Größter Problempunkt dürfte allerdings der freie Speicherzugriff in C/C++ Programmen sein, der im Gegensatz zu Java grundsätzlich nicht verhindert werden kann (zumindest nicht durch die heute zur Verfügung stehenden Möglichkeiten, die das Betriebssystem bietet). Dabei geht es hier nicht um den Ausschluss von Fehlern aufgrund schlampiger Programmierung (das bekannteste Pro-Java-Argument), sondern um vorsätzliche Aktionen eines böswilligen Applet-Entwicklers.