Kapitel 11
Koordination von Abläufen

11.1 Grafische Anwenderschnittstellen

Bislang sind wir in den Programmbeispielen mit altertümlichen Texteingabefenstern (Shell) ausgekommen, um Informationen an eine Anwendung zu übergeben bzw. die berechneten Daten zurück zu erhalten. Grafische Anwenderschnittstellen sind dank ihrer Erklärungs- und Bedienmöglichkeiten für den Anwender wesentlich angenehmer, bringen im Gegenzug allerdings eine Reihe technischer Probleme mit sich. So ist die Programmlogik, der die Bedieneinheit für die Anwenderschnittstelle folgt, in der Regel völlig von der Rechenlogik verschieden. Die einzelnen Bedienelemente sind bereits ziemlich komplex, können aus unterschiedlichen Datenquellen versorgt werden und unterliegen einer wechselnden Rangordnung.

Da nun auch die Anwendung ihre komplexe Logik besitzt und ein Mischen der beiden Komponenten zu hoher Fehleranfälligkeit und schlechter Wartbarkeit der Anwendungen führt, geht das Bemühen moderner Technologien dahin, die Komponenten zu trennen und die Informationen zwischen ihnen durch spezielle eigenen Einheiten zu vermitteln.[1]

Wir sehen uns hier exemplarisch einige der Techniken an. Es sei angemerkt, dass dieses Kapitel eher einen Lehrcharakter besitzt, da man in der Praxis zu einem der vorhandenen Frameworks für die Erstellung von grafischen Anwenderschnittstellen greifen wird und sich dann an die dort vorgegebenen Mechanismen halten muss. Auf die rein computergrafische Seite der Schnittstelle werden wir in einem anderen Kapitel genauer eingehen. Weiterhin sind solche Anwendungen in der Praxis mit Multi-Threading-Umgebungen, also parallel nebeneinander laufenden Programmteilen, versehen. Auch dies werden wir hier zunächst nicht berücksichtigen.

[1] Ein Beispiel für solche Technologien ist das MVC-Konzept (Model-View-Controller) in der Webprogrammierung, auf das wir hier aber nicht näher eingehen können.

G. Brands, *Das C++ Kompendium*, eXamen.press, 2nd ed.,
DOI 10.1007/978-3-642-04787-9_11, © Springer-Verlag Berlin Heidelberg 2010

11.1.1 Bildschirmobjekte und Ereignisse

Wir untersuchen nun exemplarisch anhand einer einfachen Textverarbeitung, wie eine Anwendung konstruiert werden kann. Als Beispiel für den Bildschirmaufbau kann der unten stehende „Screen Shot“ des Programms „Acrobat Reader“ dienen. Hier wird aber schon wesentlich mehr Funktionalität angeboten, als wir in unserer Beispielanwendung berücksichtigen werden. Die folgenden Seiten mögen Ihnen vielleicht etwas trivial erscheinen, sind aber für die Entwicklung technischer Modelle Voraussetzung.

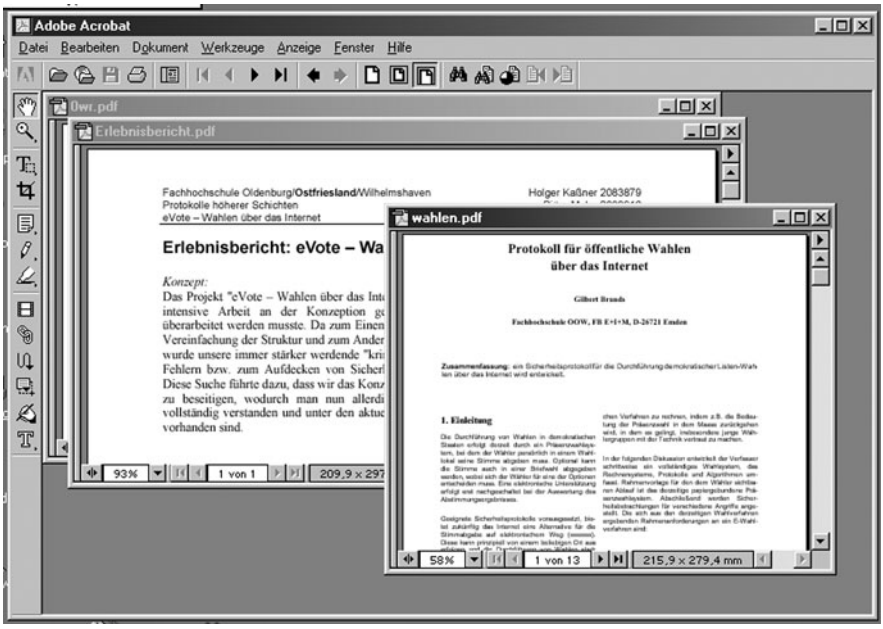
11.1.1.1 Die Anwendersicht

Die gesamte Dialogschnittstelle besteht aus einem großen Rahmen, in dem sich (*fast*) alles abspielt. In den meisten Anwendungen haben nur wenige der „inneren“ Objekte das Recht, bei ihrer Darstellung seine Grenzen zu überschreiten, und wir werden diese Fälle hier auch nicht betrachten.[2] Innerhalb des Rahmens ordnen wir zwei Objektgruppen beziehungsweise Objekte an:

- Am oberen Rand des Rahmens befindet sich eine Menüleiste, in der unterschiedliche Arbeitsfunktionen angewählt werden können. Für unser Beispiel beschränken wir sie auf das Öffnen und Schließen von Dokumenten und das Verändern der Textformatierung. Die Menüleiste ist immer sichtbar und wird durch Anklicken mit der Maus aktiviert (*erweiternd können auch Tastenkombinationen vorgesehen werden*). Nach Aktivierung kann einer der Menüpunkte angewählt und bedient oder eines der Objekte im unteren Bereich aktiviert werden, wobei die Aktivierung der Menüleiste wieder verloren geht.

  Bei Anwahl eines der Menüpunkte wird ein weiteres Fenster im Arbeitsrahmen geöffnet. Ist eine solche Funktion erst einmal angewählt, muss sie vollständig bearbeitet werden, das heißt eine Aktivierung eines anderen Objektes im Arbeitsrahmen ist nicht möglich. Nach Beenden der Funktion wird die Aktivität an eines der Dokumentenfenster übergeben.
- Unterhalb der Menüleiste ist Raum für die Darstellung einer beliebigen Anzahl von Dokumenten, deren Rahmen einander überdecken können, jedoch nicht über den Hauptrahmen hinausragen oder die Menüleiste verdecken. Zwischen den Dokumentensichten kann jederzeit durch Anklicken sichtbarer Teile teilverdeckter Fenster gewechselt werden, wobei sich die Darstellungsreihenfolge der Objekte ändert. Andere Ereignisse (*Zeicheneingabe, Formatierungsfunktion im Menü*) wirken sich jeweils auf das oberste vollständig dargestellte Dokument aus. Ein neues Dokumentenobjekt wird jeweils als oberstes Objekt eingefügt.

[2] Wie aus der folgenden Vorgehensweise ersichtlich wird, lassen sich solche Fälle durch Erweiterungen der Attributsätze realisieren, basieren also auf Verfeinerungen der Grundgedanken und nicht auf neuen Überlegungen.



Wir beschränken die Eigenschaften unserer Beispielanwendung auf diese wenigen Punkte. Ihnen ist die Funktionsweise einer Textverarbeitung hinreichend bekannt, um bei Bedarf Erweiterungen vornehmen und die Untersuchungen ausdehnen zu können. Sie werden dabei feststellen, dass die Beziehungen zwischen Objekten oder Objektgruppen als binäre Relationen der Art „*wenn ..., dann ...*“ ohne eine Angabe eines Globalbildes der Anwendung beschrieben werden. Für die Entwicklung eines Systems gilt es nun, die Relationen widerspruchsfrei zu halten und die Objekte anhand ihres Relationenmodells durch ein sinnvolles hierarchisches Klassenmodell zu strukturieren. Weitere Relationen und Klassen sind jederzeit ergänzbar, das heißt das System lebt.

11.1.1.2 Grundeigenschaften und Basismodell

Für unsere Untersuchung halten wir zunächst fest:

(a) Eine derartige Anwendung ist kein arbeitsintensives Programm, das nach einem mehr oder weniger fest vorgegebenen Schema ununterbrochen Daten bearbeitet. Im Gegenteil wird die Anwendung die meiste Zeit gar nichts tun, sondern darauf warten, das etwas geschieht, das heißt der Anwender eine Taste oder eine Tastenkombination drückt oder etwas mit der Maus oder einer sonstigen

Eingabeeinheit veranstaltet. Auf solche Ereignisse gilt es schnell zu reagieren[3] und dann wieder auf die nächste Aktion zu warten.

(b) Es ist weitgehend offen, welche Komponenten in der Anwendung zusammen wirken müssen. Das Menü ist zwar immer vorhanden, kann jedoch nach einigen Erweiterungen umgestaltet werden; die Dokumentenanzahl kann zwischen Null und irgendeiner großen Zahl schwanken; die Dokumente können deutsch, chinesische oder, bei einer etwas abstrakteren Betrachtungsweise, Bilder, Tondokumente oder Filme sein. Wir haben also kein monolitisches Programm vor uns, sondern eine Sammlung von Objekten, von denen jedes wissen muss, was es unter bestimmten Bedingungen zu tun hat.

(c) Es ist weitgehend offen, welches aus einer großen Liste möglicher Ereignisse als nächstes eintritt und wer sich darum kümmern muss. Es muss nur sichergestellt sein, dass ein Ereignis als „erledigt“ markiert und gelöscht wird, bevor das nächste Ereignis eintritt.

Um die Textverarbeitung zum Funktionieren zu bringen, müssen wir ein Organisationsschema konstruieren, wie die Objekte anzuordnen sind, sowie ein Regelwerk entwerfen, das für jedes Objekt festlegt, wie es sich im Fall eines Ereignisses zu verhalten hat. Bei diesem Regelwerk darf es keine oder allenfalls nur eine sehr untergeordnete Rolle spielen, welche anderen Objekte im System vorhanden sind.

Bringen wir zunächst Ordnung in die Menge der Objekte, die alle von einer Basisklasse erben (*hierbei kann es sich um eine Spezialisierung des Basisobjektes der Objektfabrik handeln. Wir legen dies aber nicht weiter fest*). In der Basisklasse werden einige Attribute und (*meist virtuelle*) Methoden festgelegt, die alle in einer Anwendung möglicherweise auftretenden Objekte besitzen müssen. Eines der zentralen Attribute ist beispielsweise eine Kennung, ob das Objekt aktiv ist und auf bestimmte Ereignisse reagieren darf oder sich nur passiv im Hintergrund darstellt. Zentral wird auch eine Methode benötigt, sich selbst darzustellen. Da jedes Objekt andere Vorstellungen hat, wie es sich präsentiert, ist diese Methode virtuell. Konsequenterweise operieren solche Anwendungen mit Zeigervariablen. Auf der Basisklasse baut sich eine ausgedehnte Klassenhierarchie auf, die eine Abstraktion des Relationenmodells darstellt.

Es hängt stark vom Abstraktionsgrad des Klassendesigns ab, was in einer einzelnen Klasse an Attributen und Methoden übrig bleibt und wie viele Stufen die Vererbungshierarchie aufweist. Klassenhierarchien grafischer Entwicklungssysteme besitzen eine ziemliche Ausdehnung, deren systematische Darstellung meist ein Buch mit dem gleichem Umfang wie diesem notwendig macht.[4] Wir werden aber ein wesentlich einfacheres Bild verwenden, um einen einfachen Zugang zu den Funktionsmechanismen zu erhalten.

---

[3] Schnell bedeutet hier „schnell im Sinne des Anwenders“. Da einige zehntel Sekunden für die Leistungsfähigkeit der Maschinen fast schon geologische Zeiträume darstellen, muss das nicht „schnell im Sinne der Maschine“ bedeuten.

[4] Siehe zum Beispiel Richard Kaiser, C++ mit dem Borland C++ Builder, Springer-Verlag 2002

Wir entwickeln Klassen- und Objektmodell parallel. Wie die Klassen bilden auch die Objekte in der Anwendung eine Hierarchie. Das Basisobjekt, von dem alle anderen abhängen, ist der Arbeitsrahmen, in dem sich alle Aktionen abspielen. Er kommuniziert mit dem Betriebssystem und besitzt eine Liste der „inneren Objekte“. Dieses Schema ist rekursiv, das heißt auch die inneren Objekte besitzen meist eigene „innere Objekte“, beispielsweise die Untermenues zum Öffnen und Schließen von Dokumenten. Wir legen deshalb ein Listenattribut für verwaltete innere Objekte direkt in der Basisklasse `FrameBase` an. Dabei verwenden wir die Methoden aus dem Kapitel „Objektfabriken“ für eine effektive Objektverwaltung

```
class FrameBase : public FactoryObjectsBase {
...
protected:
    list<Ptr<FrameBase*> objList;
};//end class
class MainFrame: public FrameBase {..}
```

Auf die Objekte in einer Liste kann nur über das Hauptobjekt zugegriffen werden, das damit auch in der Lage ist, Filterfunktionen durchzuführen, das heißt bestimmte Ereignisse gar nicht erst an die inneren Objekte weiter zu leiten, oder dafür zu sorgen, dass bestimmte Arbeitsschritte erst abgeschlossen werden, bevor die Kontrolle zurück gegeben wird, oder Nachrichten das inneren System nicht verlassen, da sie draussen nichts zu suchen haben. Jedes Objekt besitzt über die Objektfabrik eine eindeutige Identifikationsnummer.

In diesem einfachen Modell werden wir uns darauf beschränken, dass nur `MainFrame` Einträge in der Liste besitzt. Da in der Praxis Fenster mit und ohne solche Nebenbedingungen vorkommen, wird in komplexeren Bibliotheken mindestens eine Zwischenklasse existieren, die als Listenträger fungiert, während die eigentliche Basisklasse weitgehend leer ist.

11.1.2 Ereignisketten

In unserem Modell enthält die Liste von `MainFrame` das Menueleistenobjekt und je nach Arbeitsstand mehrere Dokumentenobjekte. Die Objektzusammensetzung kann sich dynamisch verändern. Die Synchronisation der Objekttätigkeiten wird intern nachrichtenorientiert organisiert. Darunter wollen wir verstehen, dass Vorgänge nicht durch eine bilaterale Kommunikation zwischen verschiedenen Objekten abgewickelt, sondern Neuigkeiten öffentlich bekannt gemacht werden und Objekte, die sich zuständig fühlen, selbständig darauf reagieren (*oder eben niemand, wenn kein zuständiges Objekt existiert*). Sehen Sie sich zum Verständnis dieser Vorgehensweise noch einmal unsere Eigenschaftsliste (a) – (c) einer Anwendung an. Aufgrund der angestrebten Dynamik ist es nicht möglich, jedem Objekt in der Liste ein neu hinzugekommenes bekannt zu machen und dabei noch zu entscheiden, ob ein Objekt sich überhaupt für bestimmte andere Klassen von Objekten interessiert.

Da niemand weiß, wer gerade vorhanden ist, ist auch keine Kommunikation der Art „*Du, Objekt 4711, übernimm mal!*“ möglich, sondern nur ein „Broadcast“ der Art „*Ich bin nicht mehr zuständig!*“.

Der Anstoß für eine Aktion erfolgt extern: Wird eine Taste auf der Tastatur gedrückt oder mit der Maus eine Aktion durchgeführt, so wird dies dem `MainFrame`–Objekt vom Betriebssystem mitgeteilt. Nachdem das `MainFrame`–Objekt beziehungsweise der jeweilige Listenträger überprüft hat, ob es selbst tätig werden muss, gibt es die Nachricht an alle Objekte in seiner Liste weiter, ohne selbst die Entscheidung zu treffen, wer für die Bearbeitung zuständig ist (*in diesem Fall wären wir nämlich wieder bei dem Problem angelangt, dass das Trägerobjekt sehr viel über die Umgebung wissen muss, was die Flexibilität des Gesamtsystems stark einschränkt*).

Erkennt ein Objekt seine Zuständigkeit, so kann es die Nachricht löschen oder neue Nachrichten erzeugen. Beispielsweise kann ein Dokumentenobjekt erkennen, dass es das bisherige Objekt als oberstes Fenster ablösen soll. Neben einer Anpassung des eigenen Zustands ist eine Nachricht an alle anderen Dokumentenobjekte notwendig, dass sich die Reihenfolge der Fenster geändert hat und die betroffenen Objekte ebenfalls ihren Zustand anpassen müssen (*wir untersuchen das noch genauer*). Das äußere Ereignis löst somit im Extremfall eine regelrechte Nachrichtenkaskade aus, die so lange von den Objekten der Anwendung bearbeitet wird, bis alle Nachrichten gelöscht sind.

Jedes Objekt geht bei der Auswertung einer Nachricht nur nach eigenen in der Klasse definierten Regeln vor. Vom Gesamtregelwerk müssen wir daher neben der Widerspruchsfreiheit (*es dürfen nicht zwei Objekte in Konkurrenz treten, ohne dass diese aufgelöst werden kann*) auch Rekursionsfestigkeit fordern.[5] Jede in das System eingespeiste Nachricht muss natürlich auch wieder verschwinden. Da nicht gesichert ist, dass ein Objekt die Bearbeitung übernimmt und die Nachricht löscht, legen wir als Arbeitsregel fest:

(a) Die Objekte besitzen für einen Zyklus eine konstante Reihenfolge. Alle Nachrichten werden nacheinander von allen Objekten in dieser Reihenfolge bearbeitet.
(b) Nachrichten werden spätestens dann gelöscht, wenn sie unverändert wieder bei dem erzeugenden Objekt eintreffen.
(c) Ein Zyklus ist abgelaufen, wenn alle im Rahmen eines Ereignisses erzeugten Nachrichten gelöscht sind.

[5] Widersprüche sind bei entsprechender Sorgfalt vermeidbar, bei Rekursionen ist das in komplexen Systemen oft weniger einfach. Trotz eindeutiger Regeln spielen sich bestimmte Objekte unter gewissen Umständen dann möglicherweise gegenseitig den Ball zu, was das Betriebssystem mit „Programm reagiert nicht, Task beenden?“ quittiert oder zu allgemeiner Funkstille führt. Der Kontrollaufwand zum Finden solcher Zyklen steigt proportional zur Anzahl der möglichen Kombinationen. Über diese Problematik haben wir bereits in den Anfangskapiteln diskutiert.

Beschreiben wir einen Arbeitsablauf nach diesen Regeln zunächst einmal phänomenologisch, bevor wir auf die Details eingehen. Als Beispiel diene ein Ereignis zum Aktivieren eines bestimmten Dokumentenfensters, typischerweise ein Mausereignis an einer bestimmten Bildschirmposition. Jedes Dokumentenfenster kann nun prüfen, ob die Position innerhalb des eigenen Rahmens liegt. Dabei können drei Fälle eintreten:

- Kein Fenster ist zuständig, so dass die Nachricht unbearbeitet die Liste wieder verlässt und vom Hauptmenue gelöscht wird.
- Das Ereignis liegt im Zuständigkeitsbereich eines Fensters, jedoch können auch andere zuständig sein. Die Nachricht wird durch eine Anmeldung modifiziert und nach verlassen der Liste aufgrund der Veränderung erneut an die Liste übergeben.
- Ein zuständiges Fenster erkennt, dass keine weiteren Fenster zuständig sein können, und löscht die Nachricht. Das Hauptmenue muss nicht weiter tätig werden.

Die Nachrichten verwalten wir mit Objekten von Nachrichten–Klassen, die wir ebenfalls von einer Basisklasse erben lassen. Jedes Objekt trägt eine Kennung, von wem es erzeugt (*oder gegebenenfalls verändert*) wurde. Damit lässt sich Arbeitsregel (b) problemlos realisieren

```
struct EventBase {
    enum EventType   {...}  eventType;
    ObjID  gen_from;
};//end structure
```

Von der Basisklasse ausgehend können verschieden Nachrichtentypen unterschieden werden, die über das Attribut `eventType` identifiziert werden können und eigene spezialisierte Klassen sind, beispielsweise Nachrichten über:

- Mausereignisse
- Tastaturereignisse
- Zeitereignisse
- Systemereignisse (*Signale*)
- innere Ereignisse (*unter Umständen viele verschiedene Typen. Hier können sich die Objekte gegenseitig Mitteilungen machen*)
- ...

Nach der Definition einer Nachrichtenklasse können wir die Klassenvereinbarungen für `FrameBase` und `MainFrame` erweitern. Aus unserer Ablaufbeschreibung entnehmen wir, dass wir innerhalb einer Gruppe gleichartiger Objekte gewisse Auszeichnungen vornehmen müssen. Die Dokumentenfenster überlappen einander, und zum Erkennen, welches Fenster ein anderes überdeckt, ist Attribut notwendig, das die Ebene angibt, in der sich das Objekte befindet. Vor einem Objekt in Ebene a besitzen dann alle Fenster der Ebenen $1,2,\ldots a-1$ Vorrang. Die Ebenenkennung bezieht sich aber nur auf Objekte, die sich im gleichen Rahmen aufhalten. Für das

Menü und die Fenster zum Öffnen und Schließen von Dokumenten benötigen wir keine Ebenen. Wir müssen aber unterscheiden können, ob gerade ein Dokument oder ein anderes Objekt bedient wird, das heißt wir benötigen ein Attribut für den primären „Fokus“ des nächsten Ereignisses. Wir ergänzen die Objektklassen entsprechend dieser Überlegungen und fügen noch ein Attribut hinzu, das das Löschen von Objekten steuert, sowie eine Methode zur Auswertung der anstehenden Meldungen. Während das Einfügen neuer Objekte in die Liste relativ problemlos ist, findet das Löschen von Objekten sinnvollerweise nach Abschluss eines Meldezyklus statt, um einen Ausfall von Regel (2) (*Meldung noch da, aber Erzeuger nicht mehr*) auszuschließen.

```
class FrameBase {
...
    typedef list<Ptr<EventBase*> > evList;
    int layer;
    bool hasFocus;
    bool kill;
    virtual void actEvent(evList& ev);
};//end class
```

Untersuchen wir nun die einzelnen Arbeitsfälle, an denen wir auch die weiteren Ergänzungen einführen können.

11.1.2.1 Tastaturereignisse

Allgemeine Bearbeitung eines Tastaturereignisses: Das `MainFrame`–Objekt erhalte vom Betriebssystem Mitteilungen über „normale“ Tastaturereignisse, das heißt ohne Involvierung von Funktions–, Alt– oder Strg–Tasten. Eine generelle Regel lautet dann, dass für die Auswertung solcher Ereignisse das Objekt mit `hasFocus==true` zuständig ist. Die Nachricht wird in die Meldeliste übernommen und allen Objekten zugeleitet.

```
void MainFrame::main(){
    evList evl;
    list<Ptr<FrameBase*> >::iterator it;
    do{
        EventBase * ev = SysGetEvent();
        ev->gen_from=objID;
        if(ev!=0){
            ev->gen_from=objID;
            evl.push_back(Ptr<EventBase*>(ev));
        }//endif
        while(!evl.empty()){
            for(it=objList.begin();
                !evl.empty()&&
                  it!=objList.end();++it)
```

```
                objList->actEvent(evl);
            actEvent(evl);
        }//endwhile
        remove_if(objList.begin(),
      objList.end(),KillCond());
    }while(!kill)
}//end method
```

Die Auswertung wird in der Methode `actEvent(..)` ausgeführt, die in jeder Objektklasse zu überschreiben ist. Neben den notwendigen Aktionen ist die Methode für das Löschen oder Einfügen von Nachrichten in die Nachrichtenliste zuständig sowie für das Setzen der Löschkennung. Das Löschen eines Objektes wird anschließend vorgenommen, wobei ich Ihnen zur Übung die Implementation der Klasse `KillCond()` überlasse, die die Löschbedingung überprüft (*siehe Kap. 4.6.2*). In dem angenommenen Fall eines „normalen“ Tastaturereignisses lautet die Auswertungsregel in `actEvent(..)`:

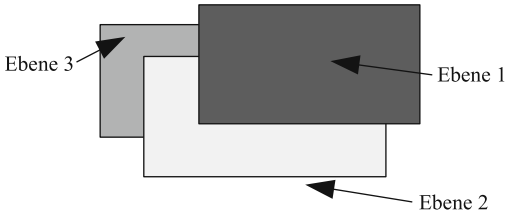
```
bool kill_message=false;
et=evl.begin();
while(et!=evl.end()){
    if(et->gen_fom==objID){
        et=evl.erase(et);
        continue;
    }//endif
    switch(et->eventType){
    ...
    case keyboard_normal:
        if(hasFokus){
            ...   // Zeichen darstellen usw.
            kill_message=true;
        }//endif
        break;
    ...
    }//endswitch
    if(kill_message){
        et=evl.erase(et);
        kill_message=false;
    }else{
        ++et;
    }//endif
}//enwhile
```

Objekte ohne Fokus gehen kommentarlos über die Nachricht hinweg, das zuständige führt die notwendigen Aktionen aus und löscht die Nachricht. Zuständig ist im Zweifelsfall das erzeugende Objekt. Der Leser verifiziere, dass das Tastaturereignis

vom `MainFrame`–Objekt gelöscht wird, falls kein Objekt vorhanden ist, das den Fokus besitzt.

11.1.2.2 Mausereignisse

Mausereignis zum Wechsel des Fokus: Nun erhalte das `MainFrame`–Objekt vom Betriebssystem ein Mausereignis gemeldet. Von dem Ereignis können sich nun ein oder gleich mehrere Objekte angesprochen fühlen. Um das zu verstehen, betrachten wir das folgende Schema.



Alle Objekte besitzen einen rechteckigen Rahmen, innerhalb dessen sie ihr Innenleben präsentieren. Die Dokumentenfenster können dabei überlappen, und auch Fenster, die durch das Menü geöffnet wurden, können in diesem Bereich liegen. Ein Masuereignis besteht nun aus den Koordinaten, an denen sich der Mauszeiger aufhält, sowie der gedrückten Maustaste.

```
struct MouseEvent: public eventBase {
    int x,y;    // Koordinaten des Mauszeigers
    int key;    // gedrückte Maustaste
    int layer;  // Ebenennummer des Objektes
};//end struct
```

Die Objekte können prüfen, ob das Mausereignis innerhalb ihres Rahmens stattgefunden hat. Da sie nichts von den anderen Objekten wissen, können sie jedoch nicht feststellen, ob sie in dem Bereich überhaupt sichtbar und damit tatsächlich zuständig sind. Die Regeln, nach denen nun vorzugehen ist, sind also etwas komplexer und auch je nach Objekttyp unterschiedlich. Überprüfen Sie die Wirkung der folgenden Festlegungen:

(a) Das Objekt ist zuständig besitzt den Fokus. Es ist damit automatisch zuständig und kann die angeforderte Aktion ausführen und die Meldung löschen.
(b) Das Objekt ist nicht zuständig, aber vom Typ „Dokument öffnen/ schließen“ (*und besitzt den Fokus*). Das Objekt muss vollständig bearbeitet werden. Unterbrechungen sind nicht zulässig. Es löscht die Meldung, macht aber sonst nichts.
(c) Das Objekt ist nicht zuständig, besitzt aber den Fokus. Es löscht den Fokuseintrag, macht aber sonst nichts (*im System ist nun kein Objekt mehr mit einem Fokus vorhanden*).

(d) Das Objekt ist zuständig und besitzt den Fokus nicht. Nun sind mehrere Fälle zu unterscheiden:

1. `event->layer < object->layer`: Es erfolgt keine Reaktion
2. `event->layer > object->layer`: Die eigene Ebenen- und die Objektkennung wird in die Meldung eingetragen.
3. `event->gen_from == objec->objID`: Das Objekt darf den Fokus übernehmen. Es ändert seine Ebenen auf Eins und sendet eine Ebenensynchronisation.

```
struct TakesFokus: public eventBase {
    int oldLayer;
}//end struct
```

Aufgrund dieser Nachricht müssen alle Objekte mit

```
object->layer < event->oldLayer
```

ihre Ebenennummer um eine Einheit erhöhen.
Alternativ zum Dokumententeil kann die Menueleiste betroffen sein:

4. Dem Objekt ist keine Ebene zugeordnet. Es setzt seine Fokuskennung und trägt seine Objektkennung ein beziehungsweise löscht die Meldung, wenn die Objektkennung bereits die eigene ist.

Überprüfen Sie, dass mit diesem Regelwerk das eingangs beschriebene Funktionsschema erfüllt wird. Wenn die Verteilung der Objekte neu gemischt ist, muss hinsichtlich der Bildschirmpräsentation noch etwas geschehen: (*mindestens*) alle Objekte, für die sich etwas geändert hat, müssen neu dargestellt werden. Die simpelste Möglichkeit ist ein ereignisgesteuerter Aufruf aller Objekte in abnehmender Ebenenfolge, wobei sich jedes Objekt auf dem Bildschirm komplett darstellt und die einander überlappenden Bereiche in der richtigen Reihenfolge überschrieben werden. In den heutigen Rechnersystemen existieren hierzu aber auch eine Reihe hardwaregestützter Möglichkeiten, in dem komplette Abbilder der Objekte in speziellen Speichern für verschiedene Ebenen halten werden. Objekte, die sich ändern (*auch solche, die nicht sichtbar sind*), stellen sich auf diesen Speichern neu dar, und für den Bildschirmaufbau werden die Puffer unabhängig von internen Aktualisierungen durch Hardwareeinheiten übereinander gelegt. Das funktioniert nahezu ohne Zeitverzögerung und macht weitere Ereignisse unnötig.

11.1.3 Änderung des Objektbaumes

Erzeugung und Freigabe von Objekten: Wir betrachten hierzu das Öffnen eines Untermenuefensters. Weitere Aktionen sind ähnlich abzuwickeln, so dass wir auf eine Diskussion verzichten können. Die Aktion wird durch ein Mausereignis im Menü angestoßen. Das Menueobjekt erzeugt ein Untermenueobjekt und sendet dies per Nachrichtenobjekt an das `MainFrame`–Objekt zum Eintrag in die List.

```
struct NewObject: EventBase {
    Ptr<FrameBase*> obj;
};//end struct
```

Um das Objekt zu aktivieren, das heißt ihm den Fokus zu übergeben, kann beispielsweise zusätzlich anschließend ein fiktives Mausereignis erzeugt werden, das nach dem Ablaufmodell (B) das Objekt aktiviert. Das anschließende Einfügen oder Löschen eines Dokuments lässt sich ähnlich abwickeln, und der Leser entwerfe Ablaufschemata dazu.

11.1.4 Das Gesamtdesign

Spätestens bei der nächsten Erweiterung wird man vermutlich in ein Redesign einsteigen, wenn der erste Entwurf sich tatsächlich an unser aus didaktischen Gründen sehr simples Modell angelehnt hat. Die Unterscheidung zwischen Menü- und Dokumentenfenster lässt sich viel klarer unter Verwendung von Subframes bewerkstelligen, die Untermenues wird man dem Menueobjekt zuordnen und nicht dem `MainFrame`–Objekt, die Meldungskaskaden sind in diesem Fall anders anzulegen, um die Arbeit in Subframes zu unterstützen, und die Verwendung eines virtuellen Mausereignisses zur Übergabe des Fokus ist nun auch nicht gerade elegant. Spätestens hier tritt dann auch die Notwendigkeit von Fenstern auf, deren Größe oder Position sich nicht am Eigentümer orientiert oder die Eigenschaften von enthaltenen Objekten koordinieren. Denken Sie an Auswahlknöpfe in Dialogen, die häufig in Gruppen angeordnet sein müssen, in denen jeweils nur eine Möglichkeit aktiviert sein darf (*und muss„ Aktiv“ ist eine andere Eigenschaft als „ Fokus“!*).

Das „Nachbessern“ ist hier aber keine ehrenrührige Angelegenheit (*vergleiche auch die Anmerkungen zum „ extreme programming“ an anderer Stelle*). Wir haben es hier nicht mit Programmmonolithen zu tun, der am Reißbrett durchgeplant wird, sondern mit einem oder weniger „sozialen“ Modell, in dem Objekte, die wenig miteinander zu tu haben, miteinander auskommen müssen. Wir haben nur wenige globale Regeln eingeführt, die beachtet werden müssen. Bei einer sukzessiven Vorgehensweise wird immer nur das umgesetzt, was neu benötigt wird, und es müssen dabei längst nicht alle Schrauben neu angezogen werden. Da die Ereignisse unabhängig voneinander und jeweils nur an wenigen Stellen von Objekten ausgewertet werden, lässt sich die Einhaltung der Regeln für jede Klasse recht gut kontrollieren, und gerade die Vielzahl unterschiedlicher Ereignisse erlaubt eine recht gute Trennung der Reaktionen der Objekte. Obwohl es also beim ersten Anblick des Klassenschemas eines größeren Entwicklungssystems fast unmöglich erscheint, bei der Vielzahl der verschiedenen Klassen einen Überblick über die innere Konsistenz des Systems zu bewahren, ist dem nicht so. Das Ganze ähnelt eher einem biologischen Evolutionsmodell: sind am Anfang noch wenige Klassen vorhanden, so sind Änderungen (*das heißt neue Arten/Gattungen/Familien*) mit größeren Auswirkungen verbunden, aber alles kann noch bequem überschaut werden; ist das Modell schon sehr weit entwickelt, fällt das Auftreten neuer Arten nur noch lokal auf.

11.1.5 Grafische Anwendungsentwicklung

Werfen wir nun einen Blick in Richtung auf ein grafisches Entwicklungssystem, das nicht nur auf einfache Art die Erstellung der Arbeitsoberfläche erlaubt, sondern zusätzlich auch die Verknüpfung von Datenfeldern untereinander oder mit Datenbanken. Dialogapplikationen mit teilweise sogar recht komplexen Abhängigkeiten oder Verwaltungsvorgängen lassen sich mit solchen Systemen weitgehend entwickeln, ohne dass der Entwickler auch nur eine Zeile Code zu schreiben braucht oder lediglich einige Fragmente einfügen muss, wobei das Entwicklungssystem dafür sorgt, dass diese an der richtigen Stelle landen. Auch hier erweist sich das Ereigniskonzept als wesentliche Komponente für die Flexibilität des Entwicklungssystems.

Bei der grafischen Anwendungsentwicklung werden zunächst die Bausteine wie Rahmen, Listen, Eingabefelder, Auswahlknöpfe usw. am Bildschirm aus einer Werkzeugleiste zusammengefügt. Das Entwicklungswerkzeug erzeugt aus diesem grafischen Bild automatisch Programmkode, allerdings in der Regel mit einem wichtigen Unterschied zu unserer Vorgehensweise im ersten Teilkapitel. In unserem Modell haben wir die Fensterobjekte mit Listen für die enthaltenen Objekte ausgestattet, das heißt das Innenleben ist völlig variabel. In Verbindung mit der Objektfabrik genügt dann als Ergebnis einer grafischen Schnittstellenkonstruktion eine Textdatei, nach der die Objektfabrik nach einigen Erweiterungen (*es müssen eine Reihe von Parametern an die Objekte übergeben werden*) die Anwendung erzeugt. Grafische Entwicklungswerkzeuge machen das aber nur noch an bestimmten Stellen, etwa bei den Dokumenten, die nur in Form einer Liste angelegt werden können (*wobei die Art Listenelemente aber wesentlich stärker eingeschränkt ist*). Alle fest definierten Bildschirmobjekte werden direkt aufgebaut, das heißt die enthaltenen Objekte sind Attribute des Trägers.

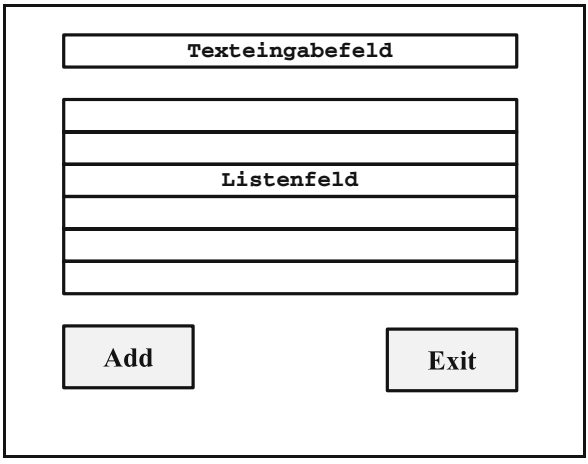
```
class MainFrame{
...
    TextField_1 tf_1;     // feste Texte im Fenster
    TextField_2 tf_2;
    TextEntry   te_1;     // Eingabe von Texten
    TextEntry   te_2;
    Canvas      cv_1;     // Feld mit Auswahlknöpfen
    Button      exit;     // Schließen des Fensters
};//end class
```

Die Methoden zur Ablaufsteuerung greifen dann auch direkt auf die Attribute zurück, das heißt der erzeugte Code ist umfangreicher als der von uns erzeugte, Kontrollen und Ablaufsteuerung werden teilweise direkt implementiert.

Warum haben wir uns mit einer andere Implementationsform beschäftigt, als die grafischen Entwicklungswerkzeuge verwenden? Die Antwort liegt in der Variabilität unserer Methode. Denken Sie beispielsweise an eine technische Anwendung, in der in einem Fenster Maschineneigenschaften beschrieben werden und aufgrund einer besonderen Kundenanforderung oder einer weiteren Maschine ein neues

Dialogelement notwendig wird. In unserem Modell erhält das spezielle Maschinenobjekt einen weiteren Eintrag in seiner Inhaltsliste und stellt sich anschließend korrekt mit der neuen Eigenschaft dar, während die vorhandenen Objekte und auch die Trägerobjekte nicht verändert werden müssen. In der fest definierten Umgebung eines grafischen Entwicklungswerkzeuges sind aber je nach Grundkonzept erheblich Änderungen notwendig. Im schlimmsten Fall entsteht eine neue Anwendung, die mit vorherigen (*an andere Kunden ausgelieferten*) nicht mehr kompatibel ist. Unser Modell ist somit eine Erweiterung, die an bestimmten Positionen zum Erreichen einer Laufzeitdynamik eingesetzt werden kann, worauf ich bereits in der Einleitung hingewiesen habe. Es müssen „nur“ einige dafür in Frage kommende Klassen der Bibliothek erweitert werden. Das setzt natürlich schon eine gute Kenntnis der Funktionsweise der Bibliothek voraus. Die Anzahl der Anwendungen, die für solche Verfahren in Frage kommen, übersteigt denn erfahrungsgemäß die Anzahl der Entwickler, die sich mit solchen Techniken auseinandersetzen. Falls Sie es einmal mit solchen Anwendungen zu tun bekommen, dürfte sich der Exkurs in diesem Kapitel wohl auszahlen, auch wenn hier keine intensiven Übungen durchgeführt werden.

Doch zurück zu der weiteren grafischen Anwendungsentwicklung: Ist das Layout fertig, werden im zweiten Schritt Abhängigkeiten wie „*das Fenster darf nur geschlossen werden, wenn die Eingabepositionen A, B und K ausgefüllt sind*“eingefügt. Auch dieser Schritt wird auf der grafischen Entwicklungsoberfläche erledigt, indem der Entwickler die Felder durch Graphen miteinander verbindet und in Hilfsfenster definiert, was zu geschehen hat. Die Vorgehensweise und die Aktivität des Entwicklungssystems sei wieder an einem Beispiel vorgestellt.

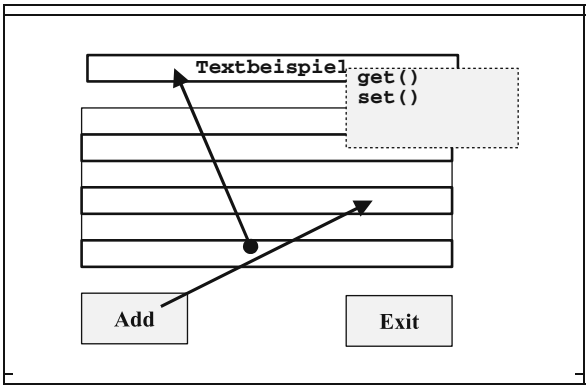


Das Menuefenster enthalte eine Liste mit Texteinträgen, ein Texteingabefeld und zwei Bedienungsknöpfe. Sobald der „ADD“–Knopf bedient wird, soll der Inhalt des Textfeldes an den Inhalt der Liste als weiteres Feld angefügt werden.

Der „ADD“–Knopf wirkt also auf die Liste, was im Entwicklungssystem durch Ziehen eines Pfeiles von Knopf zur Liste bewerkstelligt wird. Da das Entwicklungssystem weiß, um was für ein Objekt es sich bei dem Zielobjekt handelt, kann es ein

Fenster mit den möglichen Aktionen öffnen und dem Entwickler anbieten. In diesem Fall wird die Methode `push_back()` ausgewählt, da die Liste erweitert werden soll.

Allerdings ist die Anwendung damit noch nicht fertig, da der „ADD“–Knopf nicht in der Lage ist, etwas zu liefern, was an die Liste angehängt werden kann. Würde nichts weiter passieren, so würden nur Leerzeilen an die Liste angehängt. Um die Quelle zu spezifizieren, wird vom ersten Ereignispfeil ein zweiter Ereignispfeil zum Testfeld gezogen.



Auch hier wird wieder eine Liste der möglichen Methoden ausgegeben, aus der jetzt `get()` ausgewählt wird. Symbolisch haben wir damit einen Umweg des Ereignisses erzeugt. Statt direkt vom „ADD“–Knopf zur Liste zu springen, macht das Ereignis einen Umweg über das Textfeld und holt dort gewissermaßen den Inhalt ab.

Um dazu automatisch passenden Code zu erzeugen, benötigt das System zwei Nachrichtenklassen, die in der Lage sind, die Aktionskette abzuwickeln.[6] Die erste Klasse transportiert eine Nachricht an eine bestimmte Stelle und enthält eine Fortsetzungsnachricht, so dass die Nachrichtenkette individuell fortgeführt werden kann

```
struct TransferAction: public EventBase {
    SObjID          destin;
    MethodID        mid;
    Ptr<DataObj*>   data;
    Ptr<EventBase*> tact;
};//end struct
```

Die zweite Klasse ist eine verkürzte Version der ersten und beendet die Nachrichtenkette am Zielort.

[6] Da im Grunde kaum ungewöhnliches bei diesen Aktionsketten geschehen kann, enthält das Ereignismodell entsprechende Klassendefinitionen für solche Fälle. Natürlich ist auch denkbar, dass bei Bedarf spezielle Ereignisklassen nur für eine ganz bestimmte Aktion kreiert werden.

```
struct EndAction: public EventBase {
    SObjID          destin;
    MethotID        mid;
    Ptr<DataObj*>   data;
};//end struct
```

Die Funktionsweise dürfte aus der Klassendefinition unmittelbar hervorgehen. Mit dem Attribut `destin` wird ein ganz bestimmtes Objekt angesprochen, wobei hier eine statische Kennung (`SobjID`) und keine dynamische Kennung (`ObjID`) wie in Kap. 10.1.1 gemeint ist. Das Menuefenster wird ja in der Entwicklung als statisches Bild kreiert, das heißt das Entwicklungssystem kann für die feststehenden Einträge anstelle einer Liste mit variablem Inhalt direkt Attribute der entsprechenden Klassen generieren und diese statisch durchnummerieren, so dass die Kennungen kein Problem bereiten.

```
class MyFrame {
   TextWindow t1;
   TextListe  t2;
   Button     Add;
   Button     Exit;
   ...
};//end class
```

Das Objekt entnimmt `methodID`, welche Methode aufgerufen werden soll. Das Entwicklungssystem weiß für jede Verknüpfung, welche Methoden zur Verfügung stehen, so dass für die Aufstellung der Methodenliste bei hinzu kommenden Klassen keine Regeln notwendig sind.

Generalisiert werden muss allerdings das Datenfeld, da ja neben Strings auch Zahlen, Listen, Bilder usw. transportiert werden müssen und nicht für alles eigene Meldungsklassen aufgestellt werden können. Was mit Eingangs- und Ausgangsdaten geschieht, hängt von den aufgerufenen Methoden ab.

Der weiter zu reichende Nachrichtentyp kann beliebig sein, ist aber typischerweise von einem der beiden Spezialtypen, je nachdem, wie lang die Umwegkette ist. Erzeugt wird alles am Ort des Objektes „ADD“; weder für das Textfeld noch für die Liste sind irgendwelche Ergänzungen notwendig. Die Felder können so in mehrere Meldungsketten eingebunden werden, ohne dass es zu irgendwelchen Verwicklungen kommen kann. Ergänzungen sind nur dann notwendig, wenn weitere für den Bildschirmaufbau nicht benötigte Attribute oder Objekte bedient werden müssen oder spezielle Funktionen oder Fallunterscheidungen notwendig werden.

Auch das lässt sich auf der grafischen Oberfläche erledigen. Um den Text nur in Großbuchstaben in die Liste einzufügen, ist eine Modifikation der Klasse notwendig. Bei Auswahl der Methode `get()` wird dazu die Option „modifizieren“ aktiviert. Hierbei wird ein Textfenster zum Erfassen der `upcase()`–Anweisung geöffnet. Den Einbau in die Methode `get()`, wahlweise in der Original- oder in

einer von `TextFenster` erbenden Klasse,[7] einschließlich eines Schalters, dass die Anweisung nur in dieser Meldekette ausgeführt wird, führt das Entwicklungssystem selbständig durch, ohne das der Entwickler sich um Klasse und Ort kümmern muss.

Fassen wir zusammen: unser einfaches Übungsmodell lässt sich mit ein wenig Überlegen für selbstkonfigurierende Bildschirmanwendungen nutzen (*man muss nur die Objekte in der gewünschten Reihenfolge in die Liste fallen lassen und dem Trägerobjekt die Möglichkeit geben, seine Größe entsprechend zu modifizieren*). Grafische Entwicklungssysteme können mit sehr einfachen Grundprinzipien bereits recht komplexe Anwendungen erstellen, und die Entwicklung eines FrameWorks erscheint Ihnen nun hoffentlich weniger geheimnisvoll als vielmehr als langwierige Fleißaufgabe. Natürlich wird in Entwicklungssystemen teilweise mit anderen Mitteln gearbeitet (*beispielsweise mit „ Ressource–Dateien“ mit der Dateierweiterung .rc, die das spezielle Innenleben von Objekten beinhalten und mit einem eigenen Compiler bearbeitet werden, im Grunde aber nur eine abgespeckte Schnittstelle zu den Klassen darstellt*). Auch die dargestellten Kommunikationswege zwischen den Objekten unterscheiden sich in den verschiedenen Systemen. Hier weiter auf bestimmte Interna einzugehen würde aber nur den Lesern unter Ihnen nützen, die dieses System einsetzen. Als Übungsaufgabe sei Ihnen deshalb diesmal anheim gestellt, eine „Hallo Welt“–Anwendung Ihres Entwicklungssystems zu erstellen und in Bezug auf die hier diskutierten Konstruktionsmerkmal zu analysieren.

11.2 Funktoren – Aktoren

Das Aktionsbild, das wir gerade gezeichnet haben, geht davon aus, dass der Initiator einer Aktion nur eine begrenzte Menge von Informationen zur Verfügung stellt und der Empfänger genau weiß, was er damit anzufangen hat. In komplexeren Anwendungen kann aber auch der Fall auftreten, dass nur der Initiator weiß, welche Funktion durchzuführen ist, und gegebenenfalls auch über einen Teil der notwendigen Daten verfügt, während der Empfänger hauptsächlich darüber entscheidet, wann die Aktion auszuführen ist und bei Bedarf weitere Daten beisteuert. Prinzipiell wäre das ebenfalls mit hin und her laufenden Ereignissen realisierbar, aber das Ganze wird schnell unübersichtlich, unflexibel und fehleranfällig.

Es sind also weitere Mechanismen notwendig, die zunächst nur dem Initiator bekannte Methoden oder Objekte zum Aufruf durch den Empfänger bereit stellen. Solche Techniken werden in grafischen Entwicklungsumgebungen auch in größerem Stil eingesetzt; da sie aber auch andernorts sinnvoll nutzbar sind, setze ich das Thema kapitelmäßig etwas von der „Bildschirmarbeit“ ab, obwohl es formal eine logische Weiterentwicklung des begonnenen Modells ist.

---

[7] Wenn nichts weiter dazu kommt, implementiert die Klasse nur die Methode get() neu.

11.2.1 Verschieben von Funktionsaufrufen

Um neben Daten auch den Aktionskode übertragen zu können, sieht C das Konzept von Funktionspointern vor. Diese werden funktionstypspezifisch, das heißt mit genau definierten Übergabeparametern und Rückgabewerten deklariert.

```
int foo(double* array, int len);
double a[100]; int len, result;
int (*pfoo)(double*,int);
pfoo=&foo;
result=pfoo(a,len);
```

Die Variable `pfoo` kann auf beliebige Funktionen mit dieser Schnittstelle zeigen und weist auch sonst sämtliche Eigenschaften normaler Pointer auf. Beispielsweise lassen sich Felder von Funktionszeigern erzeugen, in denen man sich mittels Index oder im Iteratorstil bewegen kann:

```
int (*func[10])(int,int);       // statisches Feld

typedef int (*Func)(int,int);  // dynamisches Feld
Func* pf = new Func[10];

pf[0]=&foo;
res1=pf[0](10,10);
res2=(* ++pf)(5,5);
```

Liegen nun Zuweisungs- und Aufrufort des Funktionspointers in verschiedenen Programmteilen und sind mehrere Sorten von Funktionen auf diese Weise zu transportieren, so ist ein Kontrollmechanismus sinnvoll, der dafür sorgt, dass Funktionen so aufgerufen werden, wie ihre Schnittstelle definiert ist. Dies muss ein Mix aus Compilezeit- und Laufzeitkontrolle sein: Compilezeitkontrolle, da Fehlzuweisungen unterbunden werden, bevor eine Anwendung merkwürdige Ergebnisse liefert, Laufzeitkontrolle, da bei unterschiedlichen Quellen und Zielen auch nicht passende Pointer in einem Programmteil ankommen können und bei einer zu scharfen Compilezeitkontrolle gar nichts mehr zum Laufen kommt.

Zur Implementation der Kontrollen nutzen wir wieder das Template–Konzept, wobei für die Compilerzeitkontrollen wie üblich Typparameter, für die Laufzeitkontrolle Wertparameter eingesetzt werden. Wertparameter in Templates sind ein weniger bekanntes Konzept in C++, das es erlaubt, klassenspezifische Konstanten als Templateparameter zu definieren:

```
template <int i>
struct T {  enum { typeValue=i };};
```

Für den typunabhängigen Transport, der uns von der strengen Compilezeitkontrolle entbindet und eine Laufzeitkontrolle ermöglicht, definieren wir zunächst eine neutrale Basisklasse, die über eine Template–Funktion einen Zeiger auf sich selbst

liefern kann, wenn der eigene Typ – verschlüsselt in einem Attribut – mit dem Wertparameter des Template–Parameters übereinstimmt, bei Abweichungen aber einen Nullzeiger liefert:

```
class FunctionTransporter {
public:
    template <class T>
    const T* Get() const{
        if(T::transport = =transportedType)
            return this;
        else
            return 0;
};//end function
protected:
    int transportedType;
    FunctionTransporter (int i):
       transportedType (i){};
private:
    FunctionTransporter();
};//end struct
```

Die Konstruktoren deklarieren wir im geschützten beziehungsweise privaten Bereich, so dass diese Klasse direkt gar nicht und von den erbenden Klassen auch nur unter Einhaltung bestimmter Regeln instanziiert werden kann.

Die eigentlichen Transporter für Funktionen sind Template–Klassen, die von der Basisklasse erben. Dabei stellt sich aber ein Problem: Template–Klassen mit einer bestimmten Anzahl von Parametern können nicht ein zweites Mal mit einer anderen Parameteranzahl unter dem gleichen Namen definiert werden, das heißt wir müssten mit verschiedenen Klassenbezeichnungen je nach Anzahl der Parameter arbeiten. Wir umgehen das Problem durch einen Default–Typ, der uns auch die Möglichkeit einer zentralen Implementation gibt. Die verschiedenen Funktionstypen beschreiben wir durch das folgende Template, das nun auch den Wertparameter enthält:

```
struct NullType {}

template <int i,
          typename R,
          typename P1=NullType,
          typename P2=NullType, ... >
class Transporter: public FunctionTransporter {
    typedef R  ResultType;
    typedef P1 ParameterType_1;
    typedef P2 ParameterType_2;
    ...
    enum {transport=i};
    ...
```

Wie viele Parameter Sie insgesamt vorsehen, hängt von den Anwendungen ab; in einer realen Implementation dürfen Sie sicher bis etwa 10 gehen, um für die meisten Fälle gewappnet zu sein. Die Vergabe der Wertparameter organisieren wir in einer zentralen Typtabelle, die allerdings vom Entwickler gepflegt werden muss, beispielsweise:

```
typedef Transporter<1,int>     Func1;
typedef Transporter<2,double>  Func2;
typedef Transporter<3,int,int> Func3;
...
```

Die Pflege einer solchen Tabelle – eindeutige Wertparameter und keine versehentliche Mehrfachdefinition gleicher Funktionstypen – lässt sich zwar automatisieren, ist aber mit einigem theoretischen und praktischen Aufwand verbunden und benötigt ebenfalls einige zentrale Objekte. Wir bleiben deshalb hier bei der technisch einfacheren Variante einer manuellen Pflege. Für die Praxis empfiehlt sich die Aufstellung eines Regelschemas für die Vergabe der Wertparameter (*zum Beispiel Anzahl der Stellen = Anzahl der Parameter*); die Vergabe verschiedener Typnummern für gleiche Funktionstypen erlaubt eine interne Differenzierung.

Bei der Instanziierung von Objekten ist im Konstruktor der Funktionszeiger zu übergeben. Wir sehen entsprechend viele Konstruktoren vor, je nach Anzahl der Funktionsparameter. Dabei ist allerdings zu kontrollieren, dass nicht zu wenige Parameter übergeben werden (*zu viele Parameter würden durch die Abweichung des angegebenen Typs vom Nulltyp vom Compiler bemerkt*). Wir prüfen daher, ob der nächste freie Parameter den Nulltyp besitzt:

```
template <class T > struct TypeAssert;
struct TypeAssert <NullType> {};
..class Transporter ... {
public:
    Transporter(ResultType (*_func)()) {
        TypeAssert <P1> ();
        func0=_func;
        func1=0; ...
};//end constructor
    Transporter (
          ResultType (*_func)(ParameterType_1)){
        TypeAssert <P2>();
        func0=0; func1=_func; ...
};//end constructor
    ...
private:
    Transporter ();
    ResultType (*func0) ();
    ResultType (*func1) (ParameterType_1;
    ...
```

Sehen wir uns die Wirkung an einem Beispiel an:

```
// Zu transportierende Funktion:
int foo(int a) {  ... }

// Transporter:
/*1*/     Transporter<1,int,int>       t1(&foo);
/*2*/     Transporter<2,int>           t2(&foo);
/*3*/     Transporter<3,int,int,int>   t3(&foo);
/*4*/     Transporter<4,int,double>    t4(&foo);
```

Fall 1 ist korrekt implementiert und wird vom Compiler akzeptiert. Fall 2 und Fall 4 führen zu einem Compilerfehler, da in beiden Fällen der Typ des Funktionsparameters nicht mit dem Vorlagenparameter übereinstimmt (`int` *versus* `NullType` *beziehungsweise* `double`). Fall 3 scheitert ebenfalls im Compiler, weil dieser für `TypeAssert<int>` keine Implementation finden kann. Ein Transporter kann also nur mit einem Zeiger auf eine Funktion belegt werden, die über die gleichen Parametertypen wie der Transporter selbst verfügt.

Der Aufruf einer Funktion erfolgt mit dem ()–Operator, wobei wir hier ebenfalls so viele Versionen vorsehen, wie Funktionstypen möglich sind, und ebenfalls wieder kontrollieren, ob zu viel oder zu wenig Parameter übergeben werden:

```
..class Transporter ... {
public:
    ResultType operator()(){
        TypeAssert<ParameterType_1>();
        return func0 ();
};//end function

    ResultType operator()(ParameterType_1 p1){
        TypeAssert<ParameterType_2>();
        return func1 (p1);
};//end function
    ...
```

Der Aufruf des falschen Operators scheitert wiederum bereits am Compiler, der entweder die korrekten Typübereinstimmungen nicht findet oder über `TypeAssert` feststellt, dass zu wenige Parameter im Aufruf stehen.

```
k=t1(15);     // OK
k=t1();       // Parameter fehlt, TypeAssert
k=t1(15,15);  // int != NullType beim 2. Parameter
k=t1(15.0);   // double/int-Inkompatibilität
```

Transportiert wird der Funktionszeiger durch ein neutrales Objekt des Typs `FunctionTransporter` . Ob der an einem Auswertungsort verwendete Funktionstyp mit dem transportierten übereinstimmt, lässt sich nun leicht feststellen:

```
void Eval(FunctionTransporter& ft){
    Func3* fp;
    ...
    if((fp=ft.Get<Func3>())!=0){
        k=(*fp)(15);
    ...
```

Ein Verstoß gegen die Regeln bei der Get–Auswertung wird durch einen Laufzeitfehler geahndet. Die weitere Ausschmückung – Übertragung mehrerer Funktionen in einem Container und Auswahl der auszuführenden Funktion aus einer Menge gleichartiger – überlasse ich Ihnen.

11.2.2 Aufruf von (virtuellen) Klassenmethoden

Die Transportklasse für Funktionen kann auch zum Transport statischer Klassenmethoden verwendet werden.

```
struct T{ static int foo(); };
Transporter<10,int> st(&T::foo);
```

Das ist jedoch oft nicht ausreichend. Zusätzlich zu den Methoden sind die Objekte selbst zu transportieren, um die notwendigen Daten bereit zu stellen, und bei vielen Objekten ist die objektspezifische Version der Methode aufzurufen. Die Syntax solcher unterscheidet sich von der normalen Syntax und verwendet spezielle Operatoren:

```
struct T{ int foo(); };
int (T::*pf)();
pf=&T::foo;
T t;
(t.*pf)();
```

In der Deklaration des Funktionszeigers ist die Klasse anzugeben, mit der der Zeiger verwendet werden soll, für den Aufruf werden die speziellen Operatoren. * und ->* mit starker Bindung durch Klammern benötigt. Während die Klassenangabe in der Deklaration für die Durchführung der Compilerkontrollen gut verständlich ist, wirkt die Aufrufsyntax etwas gekünstelt und gewöhnungsbedürftig. Dafür ist C++ aber auch in der Lage, virtuelle Methoden zu transportieren. Wir betrachten das Beispiel

```
class Base {
public:
    virtual void doo(){
        cout << "Basisklassenmethode" << endl;
}};
class Next: public Base {
```

```
public:
    void doo(){
        cout << "Überschriebene Methode" << endl;
}};
```

und schreiben die Transporterklasse zum Transport von Klassenmethoden um:

```
template <class T> Transporter{
public:
    Transporter(T& o, void (T::*_func)()){
        obj=&o; func=_func;};
    void operator()(){
        (obj->*func)();
    };//end function
pivate:
    T* obj;
    void (T::*func)();
    ...
```

Bei der Implementation

```
Next h;
Transporter<Base>t(h,&Base::doo);
...
t();
```

wird der Text der Methode `Next::doo()` ausgedruckt, das heißt trotz des Konstruktoraufrufs mit `&Base:doo` wird die virtuelle Referenz über das Objekt aufgelöst. Der Compiler übernimmt also nicht einfach die Adresse der Methode, sondern im Fall von virtuellen Methoden die Adresse der Methodentabelle zur Laufzeitauflösung des Methodenaufrufs.

Problematisch bei diesem Transporter gegenüber dem reinen Funktionstransporter ist, dass das Objekt zum Zeitpunkt des Aufrufs noch existieren muss, um nicht zu Laufzeitfehlern oder sonstigen unvorhersehbaren Ergebnissen zu führen. Gleich ob wir hier mit einem Zeiger oder einer Referenz in der Transportklasse arbeiten: wenn das Objekt lokal ist und der Transporter über die Lokalitätsgrenze hinaus transportiert wird, ist das Bezugsobjekt nach Verlassen der Methode nicht mehr existent und das Ergebnis des Methodenaufrufs nicht definiert. Dieses Problem muss je nach Anwendungspolitik durch eine der beiden folgenden Methoden behoben werden:

- **Rückwirkungsfreier Transport**. Im Transporter wird anstelle des Zeigers eine lokale Kopie des Objektes angelegt.

  Diese Strategie ist jedoch nur in Fällen geeignet, in denen durch die Ausführung keine Änderungen am Initiatorort erfolgen.
- **Transport mit Zustandssynchronisation**. Der Transporter (*und alle anderen Anwendungsteile*) arbeitet mit Zeigern mit Mehrfachreferenzen. Der Zeiger wird

mit `NewReference()` eingetragen und im Destruktor des Transporters mittels `Delete()` wieder freigegeben.

Erweiternd kann die Ausführung der Objektmethode am Zielort auch davon abhängig gemacht werden, ob das Objekt am Quellort oder anderswo noch existiert.

Die Ausgestaltung dieser und anderer Möglichkeiten überlasse ich wiederum Ihnen.

11.3 Filterschlangen

11.3.1 Einfache Schlangen (Einführung)

Filter- oder Arbeitsschlangen sind überall dort notwendig, wo Daten mit unterschiedlichen und von Fall zu Fall wechselnden Algorithmen in mehreren Stufen bearbeitet werden müssen. Beispiele sind Bild- und Videobearbeitung, bei denen nacheinander Farbkorrekturen, Helligkeitsanpassungen, Störungsbeseitigung usw. durchgeführt werden, oder Datenübertragungsprotokolle, in denen die Daten komprimiert, verschlüsselt usw. werden. „Filter“ sind somit Algorithmen, die auf bestimmte Daten angewandt werden, und in den meisten Fällen sind sie für sich alleine bereits arbeitsfähig und sinnvoll einsetzbar. Bei Hintereinanderschalten mehrerer Filter ist das Endergebnis abhängig von der Reihenfolge der Filter.

Hinsichtlich der Arbeitsweise lassen sich Stromfilter und Blockfilter unterscheiden. Stromfilter verarbeiten Daten in dem Maße, wie sie dem Filter angeboten werden. Während am Eingang weiterhin Daten in den Filter gelangen, werden am Ausgang bereits Ergebnisse ausgegeben; der Filterprozess ist gewissermaßen kontinuierlich. Blockfilter fangen mit der Arbeit erst an, wenn alle Daten für den Arbeitsschritt vorliegen. Welcher Filtertyp zum Einsatz kommen kann, hängt vom Charakter der Daten ab. Beispielsweise können Binärdaten Byte für Byte in ASCII–Hexadezimalformat umgewandelt werden, während eine Mustersuche in einem Bild erst dann durchgeführt werden kann, wenn das Bild vollständig im Filter vorliegt. Zwischen zwei kontinuierlich bearbeiteten Datenblöcken kann außerdem ein Datenabschluss notwendig sein, wie es im später behandelten Beispiel der Base64–Kodierung der Fall ist. Außer dem Eingeben und der Entnahme von Daten müssen Filter je nach Typ mit Arbeitsparametern versorgt und gegebenenfalls auch Statusinformationen abgefragt werden. Beispiele sind Verschlüsselungsfilter, denen natürlich irgendwann auch der zu verwendende Schlüssel mitgeteilt werden muss.

Fassen wir diese Erkenntnisse konstruktiv zu einem Basiskonzept zusammen: Die Strukturen von Daten und Parametern sind anwendungsabhängig. Wir transportieren beides deshalb auf unstrukturierten binären Puffern, wie wir ihn in Kap. 9.6 konstruiert haben. In den Filtern muss allerdings eine Kontrolle der eingehenden Daten stattfinden. Für eine interne anwendungsspezifische Strukturierung von Daten und Parametern stehen eine Reihe von Methoden zur Verfügung:

- Mit speziellen Allokatoren kann den Daten eine direkte Objektstruktur aufgeprägt werden (*Kap. 4.4*), die in den Filtern überprüft werden kann. Transportiert werden in diesem Fall Objekte.
- Parameterstrings können zur Strukturierung von ASCII–Daten eingesetzt werden (*Kap. 5.4*). Die Daten sind in diesem Fall weniger kompakt, die Struktur ist aber sehr leicht kontrollierbar.
- Die noch zu diskutierende Strukturspache ASN.1 kann zur systemunabhängigen Strukturierung von binären Daten eingesetzt werden. Diese Möglichkeit verknüpft eine effektive Platz sparende Art der Speicherung mit einer leichten Kontrollierbarkeit.

Die Filterobjekte enthalten Methoden zum Eingeben und zum Auslesen von Daten. Für die Bildung von Filterketten weisen sie Eigenschaften verketteter Listen auf. Sofern Ketten vorliegen, werden die fertigen Daten eines Filterobjektes automatisch an das nächste Objekt der Kette übergeben, so dass die Daten an einem Ende der Kette eingegeben und am anderen Ende ausgelesen werden können. Das hört sich einfach an, es fehlen aber noch einige wesentliche Vereinbarungen, auf die man schnell stößt, wenn man sich bereits hier an eine Implementation wagt. Wir denken daher zunächst ein wenig weiter:

(a) Viele Datenverarbeitungen sind bidirektional, zum Beispiel Verschlüsselung mit symmetrischen Verschlüsselungsverfahren wie DES oder AES oder verlustfreie Datenkompression. Grundsätzlich könnte ein Filter so aufgebaut werden, dass er beides kann, was formal dazu führt, dass eine Filterkette von beiden Seiten mit Daten gefüttert werden kann. Eine derartige Realisierung führt aber schnell zu einigen Ungereimtheiten, so dass wir vereinbaren:
    Filterketten sind unidirektional. Bidirektionale Anwendungen sind durch zwei unidirektionale Ketten zu realisieren.
(b) Wenn Daten in eine Datei geschrieben oder aus einer Datei gelesen werden (*an die Stelle der Datei kann auch ein Netzwerk, eine Tastatur, usw. treten*), steht dem Anwender nur ein Ende der Kette zur Verfügung, das er dann als Datensenke oder als Datenquelle nutzen kann. Je nach Charakter des Ankerobjektes ist eine Kette daher eine Senke oder eine Quelle, und alle Objekte in der Kette müssen die gleiche Charakteristik aufweisen.

Wir fassen dies nun in einer Klassendefinition der Basisklasse `MFilter` zusammen, wobei wird die Klasse von der Basisklasse der Objektfabrik erben lassen. Wir erreichen damit

- eine einfache Identifizierbarkeit jedes Objektes in einer Kette über seine Objekt–ID,
- Mehrfachreferenzenverwaltung der Objekte,
- Verwaltung der Klassenbibliothek, da häufig von sehr großen Filterbibliotheken auszugehen ist,
- integrierte Konzepte für individuelle Bedienung von Spezialfiltern.

Mit den bei der Objektfabrik definierten Makros lautet die Klassenvereinbarung schlicht

```
NewFactoryClass(MFilter,FactoryObjectsBase)
    ...
protected:
    DBuffer m_in;
    list<DBuffer> m_out;
    bool m_end,put_chain,get_chain;
    MFilter * chain;
    ...
```

Im Implementationsteil wird eine Reihe von Methoden der Objektfabrik ebenfalls bereits durch Makros implementiert, so dass nichts zu tun ist. Sehen wir uns die angelegten Attribute an:

- Der Eingangspuffer `m_in` speichert eingehende Daten als Zwischenspeicher, falls der Filter im Blockmodus arbeitet oder die Daten nicht in dem Umfang angeboten werden, wie sie vom Filteralgorithmus benötigt werden.
- Der Ausgangspuffer ist eine FIFO–Kette von Datenpuffern, da weder garantiert werden kann, dass die Daten in dem Umfang abholt werden, in dem sie erzeugt werden, noch dass bei Verarbeitung eines Eingabedatenblockes nur ein Ausgabedatenblock entsteht.

  Zur Kennzeichnung, was genau auf der Ausgabekette vorliegt, dient das Attribut `m_end`. Wir vereinbaren:

  - Alle Datenblöcke auf der Ausgabekette bist auf den letzten (*jüngsten*) sind als abgeschlossen zu betrachten, das heißt sie sind vollständige Eingabeblöcke für den nächsten Filter (*oder den Abnehmer*) im Sinne eines im Blockmodus arbeitenden Filters
  - der letzte Datenblock ist abgeschlossen, wenn `m_end==true` ist, andernfalls handelt es sich um einen kontinuierlichen noch nicht abgeschlossenen Datenstrom.

Wieso diese Unterscheidung wichtig ist, werden wir bei der Untersuchung der Arbeit von Ketten begründen.

- `put_chain` beziehungsweise `get_chain` kennzeichnen Datenquellen oder Datensenken. Es darf jeweils nur ein Attribut den Wert `true` aufweisen; besitzen beide Attribute den Wert `false`, so werden die Daten nach Verarbeitung nicht an das nächste Objekt der Kette weitergegeben, sondern müssen manuell abgeholt werden. Die Objekte einer Kette sind so zwischenzeitlich auch als alleinverarbeitende Objekte nutzbar.
- Schließlich existiert noch der Zeiger `chain`auf das nächste Objekt der Kette, für das bei Ungültigwerden

```
if(chain!=0) chain->Delete();
```

aufgerufen wird. Für die korrekte Referenzzählung bei der Nutzung von Objekten an verschiedenen Stellen hat der Anwender selbst zu sorgen (*siehe aber auch* `Merger` *weiter unten*).

Die Verwaltung der Ketten erfolgt mit den öffentlichen Methoden

```
void SetPutChain();
void SetGetChain();
void ResetChain();
bool IsPutChain();
bool IsGetChain();

MFilter *Attached(int n);
int Add(MFilter* outQ);
int Insert(MFilter* outQ, int n);
bool Erase(MFilter* outQ);
bool Detach(MFilter* outQ);
```

`Attached(..)` gibt einen Zeiger auf das an das n–te Objekt gekettete Objekt zurück, wobei die Zählung C–konform bei Null beginnt. `Attached(0)` liefert damit einen Zeiger auf das an den Anker gekettete Filterobjekt. `Add(..)` und `Insert(..)` fügen weitere Kettenglieder an, wobei die Zählung wieder mit Null beginnt und bei zu großem Index an das Ende angefügt wird. `Add(..)` fügt weitere Elemente am Ende der bestehenden Kette an, `Insert(..)` kann auch eine Filterkette zwischen zwei Objekte einfügen.
`Erase(..)` und `Detach(..)` entfernen einzelnen Objekte aus der Kette, wobei `Detach(..)` die `Delete()`–Methode nicht aufruft, das Objekt also für anderweitige Verwendung freigibt.

Aufgabe. Es wird Zeit, dass Sie auch einmal wieder zum Arbeiten kommen und ich nicht nur alles vorbete. Implementieren Sie die Methoden!

Diese Methoden werden für alle Filter benötigt. Sie sind weder virtuell noch dürfen sie überschrieben werden. Bei weiteren Methoden gelangen wir in den individuellen Bereich. Die folgende Methodengruppe besitzt eine ähnliche Trennung in allgemeinen und individuellen Teil, wie wir das schon bei der Objektfabrik vorgenommen haben, und erlaubt das Bearbeitung von Parametern für Filterobjekte:

```
public:
    bool SetParameter(int ObjID, Dbuffer& db);
    bool GetStatus(int ObjID, Dbuffer& db);

protected:
    virtual bool Parameter(DBuffer& db);
    virtual bool Status(DBuffer& db);
```

```
// Implementation
bool MFilter::GetStatus(int oid, Dbuffer& db){
   if(oid==ObjectID())
       return Status(db);
    else if(chain!=0)
        return chain->GetStatus(iod,db);
    else{
        db.clear();
        return false;
    }//endif
}//end function

bool MFilter::Status(DBuffer& db){
       db.clear();
       return true;
}//end function
```

Die Trennung in öffentliche verbindliche und geschützte virtuelle Teile entlastet den Filterprogrammierer von der Berücksichtigung der Filterverkettung. Die Parametervariablen beider Methoden sind veränderbar, das heißt auch beim Setzen von Parametern können die Objekte bereits Nachrichten zurückgeben.
Nach einem ähnlichen Muster sind Schreibmethoden aufgebaut:

```
public:
    void Put(char inByte);
    void Put(const DBuffer& inString);
    void MessageEnd();
protected:
    virtual void Transform();
    virtual void Finish();
    void PutChain();
    void GetChain();
```

Die Methoden `Put(..)` dienen zum Eingeben von Daten in das Objekt. Eingehende Daten werden zunächst auf dem Eingabepuffer gespeichert und anschließend eine Bearbeitung eingeleitet. Danach bereits fertige Daten werden an den nächsten Filter weitergegeben. Hierfür sorgen die Methoden `PutChain()` und `GetChain()`, auf deren Innenleben wir weiter unten eingehen. Die Methode `MessageEnd()` wird zum Abschluss eines Eingabezyklus aufgerufen. Sie leitet die Verarbeitung aller noch vorhandenen Daten im Eingabepuffer und den Abschluss des obersten Ausgabepuffers ein und reinitialisiert die Eingabe, so dass neue Datenblöcke verarbeitet werden können. Intern werden die Methoden `Transform()` und `Finish()` aufgerufen, die die filterspezifischen Algorithmen enthalten.

Aufgabe. Implementieren Sie die Methoden anhand der Funktionsbeschreibung. `PutChain()` und `GetChain()` werden erst später implementiert. Prüfen Sie Ihre Implementation vor dem Weiterlesen anhand der Musterlösung, da sich die weiteren Bemerkungen darauf beziehen.

Wenn Sie Ihren Code wie in der Musterlösung entwickelt haben, ist bereits das Basisobjekt ein nutzbares Filterobjekt: Es sammelt alle übergebenen Daten im Eingabepuffer und gibt sie als kompletten Block bei der Anweisung `MessageEnd()` auf den Ausgabepuffer. Der Ausgabepuffer ist somit immer abgeschlossen und ein Objekt kann einen Übergang von einem kontinuierlichen zu einem Blockfilter vermitteln. Das hat gewisse Konsequenzen für die Programmierung eigener Filter! Das Attribut `m_end` muss bei einer Stromverarbeitung in der Methode `Transform()` auf den Wert `false` gesetzt werden, sobald Teildaten auf die Ausgabeliste gegeben werden. Die Ausgabe erfolgt dabei auf einen neuen Datenblock in der Liste, der mit `push_back(..)` anzulegen und im weiteren durch Aufruf der Methode `top()` zu bearbeiten ist.

Bevor wir auf die Kettenverarbeitung eingehen, sehen wir uns die Gegenstücke zur Dateneingabe an:

```
int  OutLength();
int  OutMessages();
bool OutputEnd();

int Get(char &outByte);
DBuffer Get();
```

Die erste Methode gibt die Anzahl der Bytes im obersten (*ältesten*) Ausgabepuffer an. Ist kein Ausgabepuffer, also auch kein leerer, vorhanden, wird (-1) ausgegeben.[8] Die zweite Methode gibt die Anzahl der abgeschlossenen (!) Ausgabeblöcke an. Ist der letzte Datenblock bei Datenstromfiltern noch nicht abgeschlossen, so sind trotz des Rückgabewerts Null weitere Daten abrufbar. Die dritte Funktion gibt an, ob der oberste Block abgeschlossen ist oder nicht. Die beiden `Get(..)`–Methoden lesen Daten aus, wobei ausgelesene Daten vom Puffer gelöscht werden und leere abgeschlossene Puffer aus der Ausgabeliste entfernt werden. Zusätzlich muss in allen Methoden die Filterkette bearbeitet werden. Exemplarisch betrachten wir eine der Methoden.

```
int MFilter::Get(char &outByte){
    if(get_chain) GetChain();
    if(m_out.size()==0)
        return 0;
    outByte=m_out.front().front();
    m_out.front().erase_front();
```

[8] Denken Sie darüber nach, warum ein Unterschied zwischen einem leeren Puffer und dem Fehlen eines Puffers gemacht wird!

```
    if(m_out.front().size()>0)
        return 1;
    if(m_out.size()>1 || m_end){
        m_out.pop_front();
        return 2;
    }else{
        return 1;
    }//endif
}//end function
```

Die Rückgabewerte sind Null, wenn kein Byte zum Auslesen zur Verfügung steht, Eins bei Rückgabe eines gültigen Bytes und Zwei bei gleichzeitigem Ende eines gültigen Blockes. Ale Ausgabemethoden sind fest implementiert und müssen nicht überschrieben werden.

> **Aufgabe.** Implementieren Sie die anderen Methoden gemäß Funktionsbeschreibung.

Die Methoden beinhalten jeweils die Kettenverarbeitung, die wir bisher nur durch die Funktionsköpfe der Methoden `PutChain()` und `GetChain()` berücksichtigt haben. Wird die Kette in Schreibrichtung verarbeitet, so sind nach `Put(..)` / `MessageEnd()` die Inhalte der Ausgabepuffer an das nächste Objekt weiterzureichen, das auf gleiche Weise verfährt. Dies ist der einfache Teil der Übung.

```
void MFilter::PutChain(){
    if(chain!=0 && m_out.size()!=0){
        while(OutMessages()>0){
            chain->Put(Get());
            chain->MessageEnd();
        }//endwhile
        if(OutLength()>0)
            chain->Put(Get());
    }//endif
}//end function
```

Ist ein Ausgabepuffer abgeschlossen, so ist auch `MessageEnd()` für das Folgeobjekt aufzurufen. Beachten Sie unter diesem Gesichtspunkt die vielleicht etwas merkwürdige Definition der Methode `OutMessages()` .

> **Aufgabe.** In der Gegenrichtung ist die Arbeit formal ähnlich. Bevor die Daten ausgegeben werden können, müssen erst die anderen Objekte nach Daten befragt und diese übernommen werden.

An dieser Stelle sollten Sie sich zwei Sachen klar machen:

(a) Bei längeren Ketten und Datenstau in einem oder mehreren Objekten kommt es infolge der Verschachtelungen zu einem längeren Hin und Her. Beispielsweise löst jedes `chain->Get()` in der letzten Methode die komplette Aufrufkette

im tiefer liegenden Kettenteil aus. Sofern dabei die Daten am Quellenende aber nicht schneller angeliefert werden, als sie verarbeitet werden können, entstehen aber keine Probleme (*die Put–Kette weist dieses potentielle Problem ohnehin nicht auf*). Solche Zustände sind aber als Ausnahmen zu betrachten.

(b) Hätten wir die Ketten nicht unidirektional gemacht und nach Quellen und Senken getrennt, hätten wir ein ernstes Problem erzeugt, da sich dann die Aufrufe in einer Get–Kette im Kreis drehen (*nehmen Sie einmal die Abfragen von* `get_chain` *und* `put_chain` *heraus und simulieren Sie das Szenarium*).

11.3.2 Filterobjekt aus der Datenübertragung (Beispiel)

Wir sind zwar mit den Filterobjekten noch nicht am Ende, betrachten aber zwischendurch als Anwendungsbeispiel für eine Arbeitsklasse eine Base64–Kodierung. Die Anfänge der Internetprotokolle rühren noch aus Zeiten her, in denen viele den Begriff „Computer“ tatsächlich für einen Lockruf für größere Hühnervögel hielten. Zu übertragen waren damals überwiegend Textdaten, und um auch die Tests zu vereinfachen, wurden die meisten Protokolle so aufgebaut, dass sie (*für den Menschen*) lesbare Zeichen übertragen. Da die Datenübertragungsraten zunächst gering waren, verwenden sie zu einem großen Teil ein 7–Bit–Alphabet, um das damals nicht notwendige achte Bit und damit Zeit einzusparen.[9] Moderne Anwendungen besitzen aber in der Regel 8 Bit–Dateneinheiten. Diese müssen auf das 7–Bit–Alphabet abgebildet und nach dem Empfang wieder in das 8–Bit–Format zurückgewandelt werden. Beschränkt man sich auf Buchstaben, Ziffern und wenige Sonderzeichen für die Übertragung (*nicht alle 7 Bit–Zeichen sind tatsächlich lesbar; einige sind zur Ansteuerung der Geräte notwendi g*), so kommt man auf ein 6–Bit–Alphabet, auf das die 8–Bit–Zeichen abgebildet werden müssen. Der Trick der Abbildung besteht nun darin, drei 8–Bit–Worte zu einem 24–Bit–Wort zusammen zu fassen und dieses durch vier 6–Bit–Worte zu kodieren. Man erhält dann folgende Bitzuordnungen der 8–Bit– und 6–Bit–Worte untereinander (*eine zweite Konvention war die Darstellung der Reihenfolge wie bei einer Zahl, das heißt die höchstwertigste Ziffer kommt zuerst. Deshalb die Bitreihenfolge 8->1. Heute würde man eher zu der Reihenfolge 0->7 tendieren*).

8	7	6	5	4	3	2	1	8	7	6	5	4	3	2	1	8	7	6	5	4	3	2	1
6	5	4	3	2	1	6	5	4	3	2	1	6	5	4	3	2	1	6	5	4	3	2	1

[9] Mit 30–50 Bit/sec war die Übertragung zunächst nur unwesentlich schneller als Morsen, zumal auf diesem Gebiet bereits während des 2. Weltkrieges Burst-Morsegeräte von der deutschen Kriegsmarine entwickelt worden sind, die das Einpeilen der Position sendender Uboot verhindern sollte.

Da die ersten Zeichen im ASCII–Alphabet für Steuerzeichen reserviert sind und die lesbaren Zeichen erst später beginnen, erfolgt eine „Umwertung“ der 6–Bit–Worte in darstellbare Zeichen mit Hilfe einer Tabelle.

```
static const char vec[] =
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ"+
          "abcdefghijklmnopqrstuvwxyz0123456789+/";
```

Außerdem ist nicht jede Nachrichtenlänge durch drei teilbar. Fehlende Nachrichtenbytes werden deshalb gegebenenfalls mit Null aufgefüllt, um eine eindeutige Kodierung des letzten Zeichens zu erhalten. Das Ergebnis wäre zwar formal eindeutig – pro fehlenden Nachrichtenbyte fehlt dem kodierten Datensatz ebenfalls ein Zeichen – aber nicht fehlerresistent. Für den kodierten Datensatz wurde deshalb folgende weitere Vereinbarungen getroffen:

- Nach einer festgelegten Zeilenlänge (*60 oder 72 Zeichen, je nach Breite der Terminals*) wird ein Zeilenvorschub eingefügt. Alle Zeilen außer der letzten haben die gleich Länge.
- Die Anzahl der Zeichen ist immer durch Vier teilbar. Fehlende Zeichen in den Urdaten werden durch das Sonderzeichen $'='$ ergänzt.

Sie können nun leicht nachvollziehen, dass Übertragungsfehler mit Zeichenverlusten oder Zeichenvermehrung immer erkannt werden, Übertragungsfehler mit Zeichenverfälschung mit einer Wahrscheinlichkeit von `w=0,75`auffallen und die Länge der ursprünglichen Nachricht immer korrekt rekonstruiert werden kann.

Zu implementieren sind die Methoden `Transform ()` und `Finish ()`. Im Prinzip handelt es sich um einen kontinuierlichen Filter (*lediglich der Abschluss eines Datenblockes muss signalisiert werden*), aber `Transform()` darf erst dann eine Umwandlung durchführen, wenn mindestens drei Zeichen vorhanden sind. Außerdem ist bei der Ausgabe jeweils zu kontrollieren, ob ein Zeilenvorschub zusätzlich auszugeben ist.

```
NewFactoryClass( Base64Encoder,MFilter)
...
void Transform(){
    while(messageIn.size()<3){
        if(m_end){
            messagesOut.add_back(DBuffer());
            m_end=false;
        }//endif
        messagesOut.back().add_back(
                  vec[(messageIn[0] & 0xFC) » 2]);
        messagesOut.back().add_back(
                 vec[((messageIn[0] & 0x03) << 4) |
                       (messageIn[1] » 4)]);
        messagesOut.back().add_back(
```

```
                vec[((messageIn[1] & 0x0F) << 2) |
                      (messageIn[2] » 6)]);
        messagesOut.back().add_back(
                 vec[messageIn[2] & 0x3F]);
        messageIn.erase_front();
        messageIn.erase_front();
        messageIn.erase_front();
        zl_size++:
        if(zl_size%20==0)
           messagesOut.back().add_back('\n');
    }//endwhile
}//end function
```

`Finish()` muss die Anzahl der vorhandenen Zeichen auf drei ergänzen und nach der Kodierung eine entsprechende Anzahl von Zeichen am Ende des Ausgabepuffers durch das Sonderzeichen ersetzen.

```
void Finish(){
    int i,l;
    if(messageIn.length()>0){
        l=3-messageIn.length();
        for(i=0;i<l;++i) Put(0);
        for(i=0;i<l;++i)
            messagesOut.back().erase_back();
        for(i=0;i<l;++i)
            messagesOut.back().add_back('=');
    }//endif
    messagesOut.push_back(DBuffer());
    m_end=true;
    zl_size=0;
}//end function
```

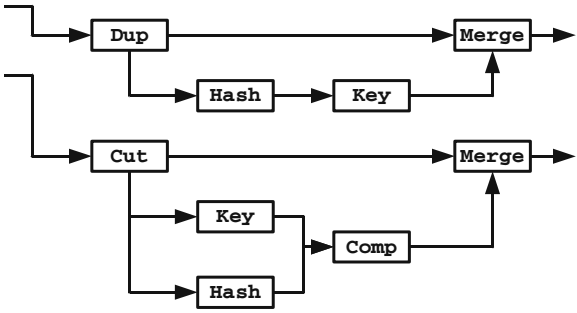
Aufgabe. Implementieren Sie die Klasse `Base64Encoder` sowie die Klasse `Base64Decoder` für die Umkehrung der Verschlüsselung und Teste sie. Als Praxistest untersuchen Sie Base64–kodierte Emails Ihres Email–Programms.

Aufgabe. In gleicher Weise implementieren Sie die Klassen `HexEncoder` und `Hexdecoder` für die Kodierung von Binärdaten im Hexadezimalformat. Im Unterschied zum ersten Kodierungsverfahren wird `Finish()` nicht benötigt.

11.3.3 Verzweigungen

In einer Reihe von Anwendungen sind lineare Filterkette nicht ausreichend. Als Beispiel betrachten Sie das Signieren einer Nachricht. Durch eine Signatur kann festgestellt werden, ob die unverschlüsselte Nachricht von einer bestimmten Person stammt und nicht verfälscht worden ist. Dazu wird von der Nachricht ein

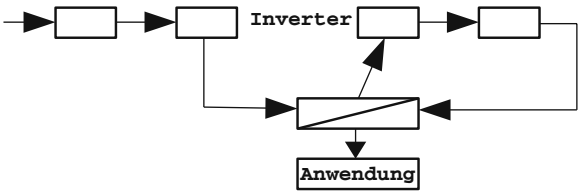
Hashwert erzeugt und dieser mit dem privaten Schlüssel eines asymmetrischen Verschlüsselungsverfahrens verschlüsselt. Dieser kann nun vom Empfänger mit dem öffentlichen Schlüssel entschlüsselt und mit dem nochmals berechneten Hashwert verglichen werden. Den Datenfluss für die Signatur zeigt das folgende Diagramm in der oberen Hälfte, die Prüfung ist in der unteren Hälfte dargestellt.



Die Verarbeitungsschritte in den linearen Teilen lasse sich durch Standardalgorithmen durchführen. Es liegt also nahe, die Verzweigungspunkte ebenfalls durch spezielle Filterklassen zu modellieren anstatt die Gesamtanwendung als Monolith zu programmieren.

11.3.3.1 Inverter

Der erste notwendige Baustein folgt direkt aus einer genaueren Datenflussanalyse. Die obere Kette im Diagramm ist normalerweise eine Put–Kette, die untere eine Get–Kette, und bei der letzten stößt man auf ein Problem: Die Get–Methode des Objekts „Cutter “ wird von den Objekten „Key“, „Hash“ und „Merger “ aufgerufen, wobei „Cutter“ weder weiß, wer gerade fragt noch sicher sein kann, dass die Abfrage zyklisch erfolgt. Um solche Unklarheiten zu vermeiden, muss die Datenflussrichtung mit der Put–Richtung übereinstimmen (*dass es dann keine Probleme gibt, muss aber auch noch belegt werden*). Erreichbar wird dies durch ein Inverterobjekt:



Die Daten werden zunächst in einer normalen Kette über die `Get()`–Funktion gezogen. Die Problemkette wird ausgekoppelt und in `Put()`–Richtung betrieben, indem das Inverterobjekt die aus dem ersten Kettenteil mit der `Get()`–Funktion ausgelesenen Daten in den zweiten Kettenteil hineingibt und vom Endobjekt mit

`Get()` das verarbeitete Ergebnis ausliest, das nun ebenfalls über die `Get()`–Funktion an die Anwendung weitergeleitet werden kann.[10] Ankerobjekt für die Anwendung ist das Inverterobjekt.

Die Funktionalität der Inverterklasse, die von der Filterklasse erbt, ist schnell zusammengestellt:

- Zwei zusätzliche Attribute werden für die zweite Kette und dessen Endobjekt eingeführt.
- Die erste Kette wird grundsätzlich als `get_chain` deklariert, die zweite Kette als `put_chain`. Die Methoden zum Einstellen der Richtung sind funktionslos zu machen, die `Put()`–Methoden des Inverters weisen keine Wirkung auf.
- Die Methoden `GetChain()` und `Detach(..)` sind zu überarbeiten, um beide Ketten bedienen zu können. `Transform()` und `Finish()` sind zu deaktivieren.

Die Schnittstelle der Klasse bekommt damit das Aussehen

```
NewFactoryClass(M Inverter,MFilter)
    MFilter *InvAttached(int n);
    int  InvAdd(MFilter* outQ);
    int  InvInsert(MFilter* outQ, int n);
    bool Detach(MFilter* outQ);
protected:
    MFilter* ichain, *chainback;
    void SetPutChain();
    void SetGetChain();
    void ResetChain();
    void Transform();
    void Finish();
    void GetChain();
    MInverter& operator=(const MFilter&);
};//end class
```

Bei jeder Einfüge- oder Löschoperation von Objekten ist das aktuelle Kettenende der inversen Kette mit Hilfe der Methode `Attached(..)` zu ermitteln. `GetChain()` führt das Umleiten auf die zweite Kette durch. Dazu wird die erste Kette mit `Get()` ausgelesen und das Ergebnis mit `Put()` auf die zweite Kette geschrieben anstatt in den eigenen Pufferbereich. Anschließend wird das Ende der zweiten Kette mit `Get()` ausgelesen und die nun fertigen Daten in den eigenen Pufferbereich geschrieben, wo sie mit `Get()` von der Anwendung abgeholt werden können. Die Methode besteht also aus zwei Versionen von sich selbst.

---

[10] Das Diagramm stellt diesen Sachverhalt nur unzureichend dar, da die Pfeile zwar die Datenflussrichtung signalisieren, aber nicht zum Ausdruck bringen, dass im linken Teil die Daten durch Get() „abgesaugt“, im rechten Teil die Daten mit Put() „hineingepunpt“ werden.

Aufgabe. Implementieren Sie die Methoden und testen Sie mit zwei Ketten, jeweils bestehend aus den Hexadezimal– und den Base64–Kodierern.

Aufgabe. Der Inverter liest die Daten am Ende der umgekehrten Kette aktiv mit `Get()` aus, das heißt das letzte Objekt der Kette besitzt einen Nullzeiger und zeigt nicht auf das Inverterobjekt. Warum wird die Kette nicht geschlossen?

11.3.3.2 Cutter

Wesentlich für die weitere Funktion ist das Cutter–Objekt, das eine Nachricht nach bestimmten Kriterien für die verschiedenen Ausgabefilter aufbereitet. Diese Kriterien sind natürlich anwendungsspezifisch, so dass die einfache Lösung die Konstruktion individueller Cutter–Klassen ist. Man kann sich aber leicht vorstellen, dass dies ins uferlose ausarten kann, da für jede Kombination von Aufbereitungsarten eine eigene Klasse implementiert werden muss. Wir treiben die Abstraktion daher weiter: Eigentlich sind für jede Unterkette nur spezielle Transform–Methoden notwendig, die wir als abstrakte Klassen einführen. Anstatt für jede Kombination von Transform–Methoden eine eigene Cutter–Klasse zu definieren, muss für eine Ausgabekette nur die Transform–Klasse geladen werden.

```
NewFactoryClass(MTransform,FactoryObjectsBase)
public:
    virtual void SetCondition(DBuffer& db);
    virtual void GetCondition(DBuffer& db);
    virtual bool Transform(DBuffer& data);
protected:
    DBuffer parm;
};//end class
```

Die Cutter –Klasse selbst erbt von der Filterklasse, erhält aber eine eigene Attributliste, die die verschiedenen Ausgabeketten und zu jeder Ausgabekette das zugehörende Filterobjekt und eine Ausgabeschlange enthält. Die Attribute der Mutterklasse außer dem Eingabepuffer werden nicht genutzt; die Objekte der Klasse arbeiten grundsätzlich in der Richtung `put_chain` und sind nicht anders konfigurierbar.

```
NewFactoryClass(MCutter,MFilter)
...
protected:
    vector<MFilter*>        vchain;
    vector<MTransform*>     vcond;
    vector<List<DBuffer> >  vout;
    int actual;
    ...
};//end class
```

Das Attribut `actual` wird durch die Methode `SetParameter(..)` bedient und dient bei Aufruf der Methoden `Attach(..), Insert(..)` und `Add(..)` zur Festlegung, welche Unterkette gemeint ist (*für die anderen Befehle ist das unnötig, Warum?*). In Bezug auf die Filter– und Transform–Objekte legen wir fest, dass sie immer parallel bedient werden. Bei Einfügen eines neuen Elementes wird ein Objekt der Basistransformklasse erzeugt, bei Löschen eines Elementes wieder vernichtet

```
int MCutter::Add(MCutter* outQ){
    if(vchain[act]==0){
        vchain[act]=outQ;
        vcond[act]=new MTransform();
    }else
        vchain[act]->Add(outQ);
}//end function
```

Aufgabe. Implementieren Sie die Methoden `Insert(..)` und `Detach(..)` und den Destruktor.

Aus dieser Vereinbarung resultiert, dass zuerst der Beginn einer neuen Unterkette angelegt werden muss, bevor die Transformation der Daten festgelegt wird. Die Methode `SetParameter(..)` verdient in diesem Zusammenhang eine genauere Untersuchung. Wir vereinbaren folgende Parameterstrings:

```
ActChain( __nr__ )
TransformObject( __name__ )
TransformParameter( __ASCII_HEX__ )
```

Die Auswertung erfolgt nacheinander, das heißt die drei Parametersätze können gleichzeitig übertragen werden, müssen aber in dieser Reihenfolge auftreten, wenn sie sich auf eine Unterkette beziehen. Der zweite Parametersatz erlaubt den Austausch der Basistransformobjekte gegen spezielle Objekte, wobei der Mechanismus der Objektfabrik eingesetzt wird:

```
Parameter p;
p.SetSequence(db.c_str());
if(p.Find("CObject")>=0 && vchain[act]!=0){
    MDecision * h = CreateObject(p.SubPar
                                 ("CObject",0));
    if(h!=0){
        delete vcond[act];
        vcond[act]=h;
    }//endif
}//endif
```

Die Parameter für die Transformobjekte müssen unabhängig von der benötigten Form im ASCII–Format sein, um mit Parameterstrings verarbeitet werden zu

können. Sofern es sich um Textparameter handelt ist dies kein Problem, andernfalls können die Klassen `HexEncoder` und `HexDecoder` für eine geeignete Umformatierung eingesetzt werden.[11]

Bei der Arbeit des Objektes müssen wir sicherstellen, dass die Unterketten synchron betrieben werden. Dazu legen wir fest:

- Die Transformation wird erst dann durchgeführt, wenn die Eingabe durch `MessageEnd()` abgeschlossen wird. Teilstromverfahren sind nicht zulässig. Die Objekte arbeiten somit grundsätzlich im Blockmodus.
- Unterketten werden in der Reihenfolge aktiviert, in der sie angelegt sind. Falls die Daten am anderen Ende wieder zusammengeführt werden, treten sie dort damit auch in einer festgelegten Reihenfolge auf, obwohl alle Ketten auf die gleiche `Put()`–Methode des zusammenfassenden Objektes zugreifen.

Es laufen somit nur zusammengehörende Nachrichten durch die Unterketten. Zu überarbeiten sind die Methoden `Finish()` und `PutChain()`.

```
void MCutter::Finish(){
    DBuffer db;
    int i;
    for(i=0;i<vcond.size();++i){
        db=m_in;
        if(vcchain[i]!=0 && vcond[i]->Transform(db))
            vout[i]->push_back(db);
    }//endif
}//end function
```

Aufgabe. `PutChain()` behandelt nacheinander alle Unterketten. Der Leser implementiere die Klasse nun vollständig und Teste sie mit einer Transformklasse, die die Daten nicht verändert und immer den Rückgabewert `true` liefert (*Datenvervielfachung*).

11.3.3.3 Merger

Als letzte Objektgruppe sind Merger –Objekte zu untersuchen, also solche, die Daten aus verschiedenen Ketten wieder zusammenführen. Es lässt sich leicht überlegen, dass hier keine neuen abstrakten Klassen notwendig werden, sondern normale Filterklassen mit einigen speziellen Eigenschaften einzusetzen sind. Beim Einfügen in eine verzweigte Kette ist nur auf die Referenzzählung zu achten:

```
MCutter * mc=MCutter::New();
...
```

[11] Das ganze Verfahren scheint auf den ersten Blick nicht sonderlich elegant zu sein, gliedert sich jedoch problemlos in die Vorarbeiten ein. „Sondermethoden“ oder aufgeblähte Schnittstellen in den Basisklassen wären sicher die schlechtere Lösung.

```
MFilter * mf=MFilter::New();
...
mc->SetParameter(DBuffer("ActChain(0)"));
mc->Add(mf);
mc->SetParameter(DBuffer("ActChain(1)"));
mc->Add(mf->NewReference());
...
```

Aufgabe. Entwickeln Sie folgende Filterklassen:

(a) Ein kompletter Datenblock soll mit einem Kopfteil und einem Fussteil versehen werden. Kopf und Fuß sind zu parametrieren.
(b) Eine parameterierbare Anzahl vollständiger Datenblöcke soll zu einem neuen vollständigen Datenblock zusammengefügt und dann als kompletter Datenblock ausgegeben werden.

Konstruieren Sie damit folgende Transformation:

```
Eingabe:     ___irgendein_Textstring___
Ausgabe:     Stringstart
             ___irgendein_Textstring___
             Stringende
             Base64-Kodierung
             ___kodierter_Textstring___
             EndeKodierung
```