

BE32

Beispielentwurf zu PÜ32 Projektübungsblatt: Codierung von Dateien

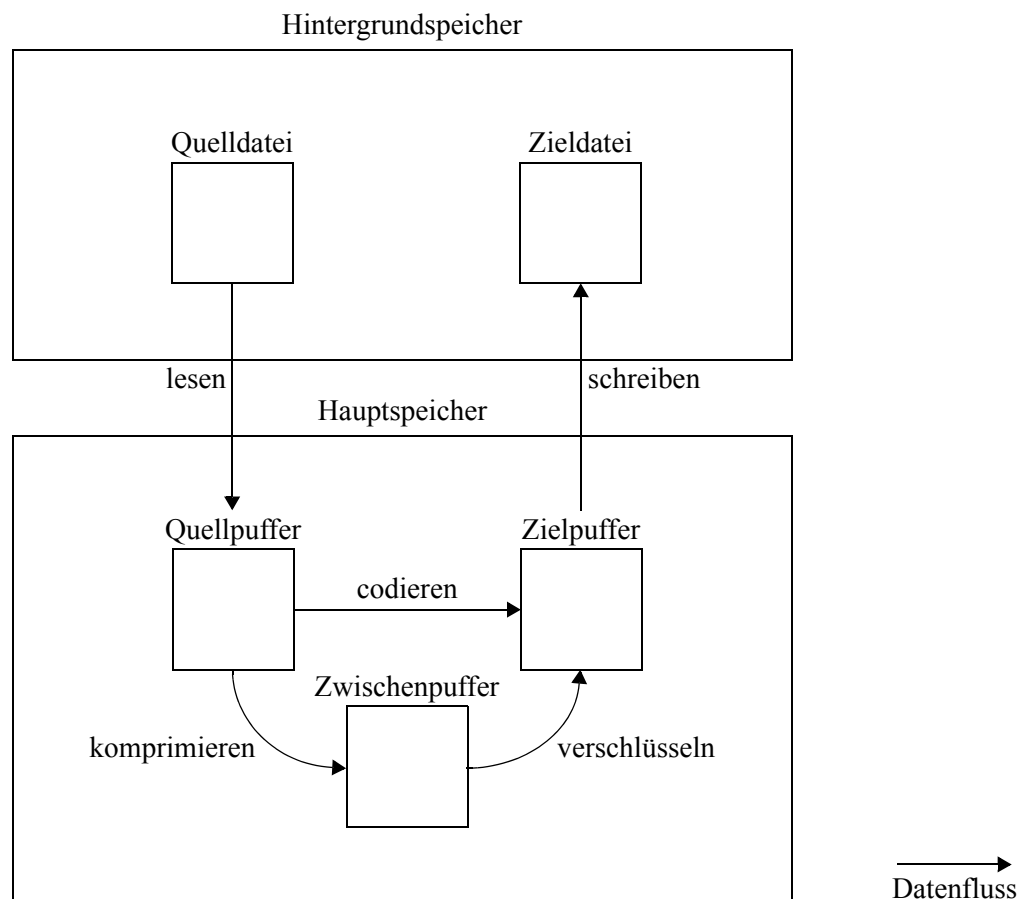
Zu *PÜ32 Projektübungsblatt: Codierung von Dateien* entwickeln wir einen Beispielentwurf mit zwei Varianten, der die *LS Leitlinien zum Softwareentwurf* beachtet und die in *KE32 Kritik eines Entwurfs zu PÜ32 Projektübungsblatt: Codierung von Dateien* erkannten Schwächen vermeidet. Die Implementationssprache ist C++.

BE32.1

Physisches Modell des Problembereichs

Bild BE32.1 setzt *Leitlinie LS.2, PÜ32.4(2)(a)* um. Der Inhalt einer Quelldatei (Eingabedatei) wird in den Hauptspeicher gelesen, dort codiert und in eine Zieldatei (Ausgabedatei) geschrieben. Da Codierung am Ort nicht immer möglich ist, ist je ein Quell- und Zielpuffer vorgesehen. Verschlüsselung ist byte- oder blockweise möglich, aber Kompression und Fehlerprüfungscodierung nur blockweise. Da wir alle Codierungsarten mit einer Abstraktion erfassen wollen und der Blockansatz allgemeiner ist, sehen wir im Hauptspeicher große Puffer vor, die idealerweise den ganzen Dateiinhalt fassen.

Bild BE32.1
Physisches Modell der
Codierung von Dateien



BE32.2

Allgemeine Entwurfsentscheidungen

- (1) „Mach es so einfach wie möglich, aber nicht einfacher!“ Dieses Motto von Albert Einstein und Niklaus Wirth gilt auch hier.
- (2) Die **Entwurfsmethode** ist primär objektorientiert, ggf. mit modularen oder prozeduralen Teilen. Im Folgenden bedeutet Einheit = Entwurfseinheit = Modul oder Klasse. Der Entwurf ist weitgehend programmiersprachenunabhängig, lässt sich also mit verschiedenen objektorientierten Sprachen implementieren.
- (3) Freilich beachten wir alle *LS Leitlinien zum Softwareentwurf* auch bei diesem Projekt. Betont seien hier:

- (a) Die **einheitliche Dokumentationssprache** (*Leitlinie LS.8*) ist Englisch (*MB.7(9)*, *PÜ32.5(9)*).
- (b) **Dokumentation** (*Leitlinie LS.9*): Jede Quellcodedatei enthält einen Kopfkomentar (*MB.7(10)*, *PÜ32.5(10)*). Alle Schnittstellen sind so dokumentiert, dass Programmierer sie benutzen und erweitern können, ohne Implementationen studieren zu müssen (*MB.7(12)*, *PÜ32.5(12)*).

Beim Gestalten der Schnittstellen von Einheiten achten wir besonders darauf:

- (c) **Trennung von Abfragen und Aktionen** (*Leitlinie LS.31*): Dienste verändern entweder einen Zustand oder liefern eine Information.
- (d) **Nebeneffektfreiheit** (*Leitlinie LS.32*): Direkt aus (c) folgt, dass Dienste keine beobachtbaren Nebeneffekte bewirken (*MB.7(7)*, *PÜ32.5(7)*). Dies schließt nicht aus, in gut begründeten Ausnahmefällen nebeneffektbehaftete Funktionen einzusetzen (Beispiel: Fabrikfunktion).
- (e) **Ausnahmefreiheit**: Schnittstellen sollen einfach sein. Ausnahmen (*exception*) sind schwer zu dokumentieren. Deshalb sollen Schnittstellen ausnahmefrei sein: Dienste sollen keine Ausnahmen werfen, Kunden sollen keine Ausnahmebehandlungen vorsehen müssen (*MB.7(8)*, *PÜ32.5(8)*). Dies schließt das Auslösen und Behandeln von Ausnahmen innerhalb einer Einheit nicht aus (Beispiel: Zerteiler in Projekt 1).
- (4) **Trennung von Ein-/Ausgabe und Funktion** (*Leitlinie LS.14*, *Leitlinie MB.1*, *Leitlinie MB.2*, *Leitlinie PÜ3.2*, *PÜ32.5(5)*): Eine wesentliche Teilung der Lösung ist die zwischen dem Ein-/Ausgabeteil (E/A-Teil) und dem Funktions- teil. Der E/A-Teil besteht nur aus der Implementationsdatei *main.cpp*, der Funktions- teil aus allen anderen Dateien.
- (5) **E/A-Teil**: *main.cpp* enthält die in C++ unvermeidliche *main*-Funktion. Sie übernimmt Argumente von der Kommandozeile und analysiert sie, ruft funktionale Teile auf, und gibt deren Ergebnisse in die Konsole aus (*Leitlinie MB.3*, *Leitlinie PÜ3.1*, *PÜ32.5(5)*).
- (6) **Funktionsteil**: Wir zerlegen ihn in mehrere Einheiten, wobei wir diese Anforderungen beachten:
 - (a) **Schwache Kopplung** (*Leitlinie LS.13*): Die Schnittstelle zwischen E/A-Teil und Funktionsteil ist möglichst schmal. Der E/A-Teil kennt möglichst wenige funktionale Einheiten, also bestenfalls nur eine.
 - (b) **Erweiterbarkeit**: Erweiterungen des Funktionsteils dürfen seine Schnitt- stelle zum E/A-Teil nicht wesentlich ändern und im E/A-Teil nur entspre- chende Erweiterungen der Kommandozeilenargumente und ihrer Analyse erfordern.

Entwurfsmuster
Fassade

Die funktionale Einheit, die die Schnittstelle zum E/A-Teil bildet, ist die Klasse *FileCoder*. Sie folgt dem Entwurfsmuster Fassade, indem sie alle anderen funk- tionalen Einheiten hinter ihrer Schnittstelle verbirgt (*Leitlinie LS.27*).

- (7) **Programmierschema für schlechte Fälle** (*Krimskrams – Vorbedingung, Fehl-anzeige oder Ausnahme? (VFA)*): Nicht alle Teilaufgaben sind stets durchführbar: Das Öffnen einer Datei unbekannten Namens scheitert, das Schreiben und das Dekodieren einer Datei können scheitern. Welches Programmierschema verwenden wir für Ergebnismeldungen? Wegen (3)(e) scheiden Ausnahmen für schlechte Fälle an Schnittstellen aus. Wir nutzen einheitlich das Aktion-Abfrage-Schema:

Aktion-Abfrage-Schema, lieferantenseitig

```
public:
    void try_action (); // and set 'a_result'.
    int result () const {return a_result;}

    static const int done          = 0;
    static const int failed_action = 1;
    // Values for 'result()'.

protected:
    int a_result;
```

Auch die *stream*-Klassen der C++-Standardbibliothek verwenden das Aktion-Abfrage-Schema für Ergebnismeldungen. Sein Vorteil ist Einfachheit: Es kommt ohne spezielle Sprachkonstrukte aus und ist leicht zu dokumentieren. Nachteile sind: Kundenprogrammierer können Ergebnisabfragen vergessen oder ignorieren. Falls Ergebnismeldungen über mehrere Aufrufe zurückzugeben sind, führen Ergebnisbehandlungen in Kunden nach dem Muster

Aktion-Abfrage-Schema, kundenseitig

```
supplier.try_action();
a_result = supplier.result();
if (a_result != done) return;
```

zu unelegantem und unstrukturiertem oder tief geschachteltem Quellcode. Der Quellcode mit Ausnahmebehandlung ist aber höchstens dann besser, wenn Ergebnismeldungen über mehrere Aufrufe zurückzugeben sind.

- (8) **Laufzeiteffizienz**: Daten werden nicht unnötig zwischen Hintergrund- und Hauptspeicher übertragen und nicht unnötig im Hauptspeicher (Laufzeitkeller, Halde) kopiert. Bild BE32.1 zeigt die unvermeidlichen Kopiervorgänge:
- (a) vom Hintergrundspeicher in den Hauptspeicher,
 - (b) im Hauptspeicher für jede Codierung von einem Puffer in einen anderen,
 - (c) vom Hauptspeicher in den Hintergrundspeicher.

Hintergrundspeicherzugriffe kosten viel Laufzeit. Deshalb ist unbedingt zu vermeiden, dass bei mehrfacher Codierung Zwischenergebnisse in Dateien geschrieben und wieder gelesen werden. Zu vermeiden sind auch unnötige Kopiervorgänge im Hauptspeicher, sei es durch ungeschickt entworfene Dienst-schnittstellen mit Puffern als Wertparameter oder Funktionsergebnisse, sei es durch ungeschickte Implementationen.

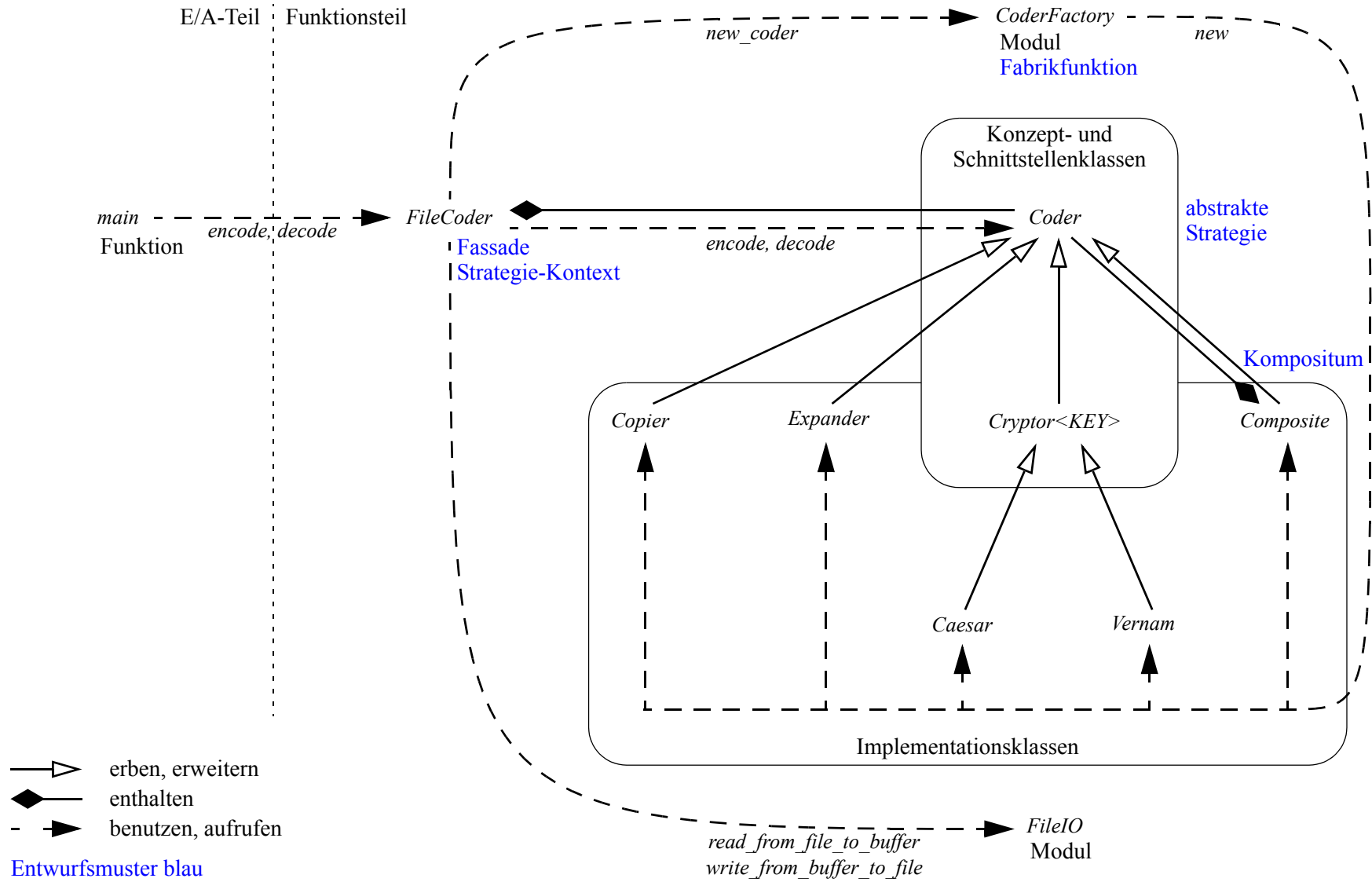
- (9) **Programmierstil**: Da C++ eine hybride Sprache ist, setzen wir den objektorientierten Programmierstil beim Umsetzen des Entwurfs nicht zwanghaft überall ein, sondern nur dort, wo er gut begründet ist. Wir gestalten einzelne Lösungsteile modular oder prozedural, wenn es sinnvoll ist.

BE32.3

Zerlegung in Module und Klassen

Bild BE32.2 zeigt die Entwurfseinheiten, ihre Rollen in angewandten Entwurfsmustern, und ihre Beziehungen zueinander.

Bild BE32.2 Modul- und Klassendiagramm des Entwurfs



BE32.4**Schnittstellen der Module und Klassen**

Die vollständigen Schnittstellen mit Verträgen siehe in den Quellcode-Header-Dateien unter

http://informatik.karlheinz-hug.de/Lehrmaterial/Informatik_3/C++/Coding/Include

Hier folgen Begründungen für wichtige Schnittstellenelemente. Die Reihenfolge ist von außen nach innen, von oben nach unten: vom Kunden zum Lieferanten, vom Abstrakten zum Konkreten.

Erweiterbarkeit: Bei Klassen vereinbaren wir alle Methoden und Destruktoren als *virtual*, damit bei späteren Erweiterungen Polymorphie und dynamisches Binden nicht von vornherein ausgeschlossen sind. Freilich sind davon ausgenommen Getter, die wir stets als Inline-Funktionen vereinbaren, und *static* Methoden.

Lesbarkeit: Getter benennen wir ohne blähendes „get_“-Präfix; stattdessen erhalten die geschützten Attribute das Präfix „a_“, um Namenskonflikte zu lösen. Dieses Entwurfsdokument wirft syntaktischen Ballast ab: *virtual* vor Methodenvereinbarungen, *void* als Ergebnistyp, „= 0“ bei abstrakten Methoden, „;“ am Ende von Methodenköpfen.

BE32.4.1**Main**

Hier ist nicht die Schnittstelle, sondern die Benutzungsoberfläche von *main*, also die Syntax und Semantik der Kommandozeilenargumente, zu beschreiben. Die Syntax legen wir mit diesen EBNF-Regeln fest:

```

command      = command_name source_filename target_filename direction function .
command_name  = 'CodingI'.
source_filename = Pfadname .
target_filename = Pfadname .
direction     = 'encode' | 'decode'.
function      = 'copy' | 'compress' | ( 'crypt' | 'compress+crypt' ) method .
method        = 'caesar' caesarkey | 'vernam' vernamkey .
caesarkey     = integer .
vernamkey     = string .

```

Ihre Semantik ist hoffentlich ohne Erläuterung verständlich. Als Beispiel zur Kombination von Codierungen ist Kompression gefolgt von einer Verschlüsselung ('compress+crypt') gewählt. Hier seien die Fragen in [PÜ32.4\(11\)](#) beantwortet: Sowohl Kompression als auch Verschlüsselung wollen in der Quelle enthaltene Redundanz beseitigen; Kompression will die Größe möglichst stark reduzieren, Verschlüsselung das Entschlüsseln möglichst schwer machen. Wird erst verschlüsselt, bleibt u.U. keine Redundanz zur Kompression übrig. Deshalb ist nur die Reihenfolge „erst komprimieren, dann verschlüsseln“ sinnvoll. Fehlerprüfungs-codierungen können folgen.

Die Abweichung der Syntax vom Beispielsatz in [PÜ32.4\(2\)\(b\)](#) ist begründet: Es erleichtert die Analyse der Argumente, wenn die Richtung (*encode*, *decode*) von der Funktion (*copy*, *compress*, *crypt*) getrennt ist. (Dafür nehmen wir die Doppelung *encode copy* und *decode copy* in Kauf.) Auf Konformität mit MS-DOS-Shell-Regeln („/“, „=“) verzichten wir der Einfachheit halber. Auf kryptische Abkürzungen verzichten wir zugunsten Verständlichkeit.

BE32.4.2**FileCoder**

FileCoder ist eine konkrete Klasse, die Dateien codiert und decodiert. Die Parameter ihrer Methoden *encode* und *decode* sind die Namen einer Quell- und einer Zieldatei. Die Dateinamen sind Eingabeparameter beider Methoden, realisiert als Referenzen (&) auf konstante (*const*) *string*-Objekte:

```
encode (const string & source_filename, const string & target_filename)
decode (const string & source_filename, const string & target_filename)
```

Da *encode* und *decode* invers zueinander sind, hängen sie funktional stark zusammen. Deshalb gehören sie in dieselbe Einheit (*Leitlinie LS.11*, *Leitlinie LS.12*). Die Zerlegung ist scheinbar rein prozedural: Zwei Aktionen, keine Abfragen, kein Zustand. Allerdings kommt nach BE32.2(7) eine Abfrage für die Ergebnismeldung hinzu (Aktion-Abfrage-Schema).

Warum ist *FileCoder* nicht als (abstrakte) Schnittstellenklasse entworfen, zu der beliebig viele Implementationsklassen verschiedene Codierungsverfahren implementieren? Weil *FileCoder* die Dateiarbeit enthält und an ihre Ableitungsklassen vererben würde. So wäre die Dateiarbeit stark mit den Codierungsverfahren gekoppelt. Codierungsverfahren wären u.U. nur extrem ineffizient kombinierbar.

Entwurfsmuster
Strategie

Ziel ist, Dateiarbeit und Codierungsverfahren so zu entkoppeln, dass die Dateiarbeit austauschbar und wiederverwendbar wird und die Codierungsverfahren austauschbar und kombinierbar werden. Zur Entkopplung von Dateiarbeit und Codierungsverfahren dient das Entwurfsmuster Strategie (*Leitlinie LS.27*): *FileCoder* entspricht dem Kontext, die abstrakte Klasse *Coder* der abstrakten Strategie, die Konkretisierungen von *Coder* den konkreten Strategien.

Warum ist *FileCoder* eine Klasse, kein Modul? *FileCoder* hat als unveränderlichen Zustand ein *Coder*-Objekt, das sie zur Codierung benutzt. Verschiedene *FileCoder*-Objekte mit verschiedenen *Coder*-Objekten können gleichzeitig existieren. Die Modulvariante würde dies ausschließen.

FileCoder verbirgt sein *Coder*-Objekt und alle *Coder*-Typen hinter seiner Schnittstelle (*Leitlinie LS.28*). Es bietet einen Konstruktor, der von *Coder*-Typen abstrahiert, indem er sie durch das Zeichen *coder_type* codiert, aber trotzdem beliebige Konfigurationen erlaubt. Der Destruktor ist erforderlich, um das *Coder*-Objekt zu löschen:

public:

```
FileCoder (char coder_type, const string & key)
// Values for 'coder_type':
// 'e' : Expander
// 'c' : Caesar
// 'C' : Composite (Expander, Caesar)
// 'v' : Vernam
// 'V' : Composite (Expander, Vernam)
// default : Copier

~FileCoder ()
```

protected:

```
Coder * const coder // implementation invariant: coder != 0
```

Der Testbarkeit des Programms (*PÜ32.5(3)*) dient es, wenn der Name der Klasse des ausführenden *Coder*-Objekts auf der Konsole erscheint. (Für Benutzer ist diese Information belanglos.) Dazu muss *FileCoder* dem E/A-Teil keinen Zugriff auf sein *Coder*-Objekt erlauben, sondern nur diese Abfrage bieten:

```
string coder_name () const
```

BE32.4.3

FileIO

FileIO ist ein Modul, dessen Methoden von Dateien in Puffer lesen und von Puffern in Dateien schreiben:

```
static read_from_file_to_buffer (const string & filename, vector<char> & buffer)
static write_from_buffer_to_file (const vector<char> & buffer, const string & filename)
```

Analog zu *FileCoder* gilt: Da *read_from_file_to_buffer* und *write_from_buffer_to_file* invers zueinander sind, hängen sie funktional stark zusammen. Deshalb gehören sie in dieselbe Einheit ([Leitlinie LS.11](#), [Leitlinie LS.12](#)). Auch hier ist die Zerlegung scheinbar rein prozedural, aber nach BE32.2(7) kommt eine Abfrage für die Ergebnismeldung hinzu (Aktion-Abfrage-Schema).

Warum sind *FileCoder* und *FileIO* zwei Einheiten, warum sind sie nicht in eine Einheit integriert? Weil *FileIO* unabhängig von *FileCoder* eine Existenzberechtigung besitzt: Als eigene Einheit ist *FileIO* in anderen Programmen wiederverwendbar.

Warum ist *FileIO* ein Modul (realisiert als Klasse mit ausschließlich *static* Methoden), keine Klasse? Weil es keinen Zustand hat; die Ergebnismeldung zählt nicht als Zustand. Objekte ohne Zustand erzeugen zu müssen ist sinnlos, solange die Objekte nicht austauschbar sein müssen.

Warum ist *FileIO* als Modul nicht mit einem Namensraum realisiert? Weil Klassen mit *static* Methoden zwar etwas mehr Schreibaufwand als Namensräume erfordern, aber hinsichtlich Änderbarkeit, Erweiterbarkeit und Schutz flexibler als Namensräume sind.

Was ist nachteilig daran, dass *FileIO* ein Modul ist? Es ist als Modul nicht austauschbar. Ist Austauschbarkeit gefordert, so lässt sich *FileIO* leicht vom Modul in eine Schnittstellenklasse mit verschiedenen Implementationsklassen umwandeln.

Warum ist der Puffertyp *vector<char>*, nicht *char** oder *char[]*? Weil letztere antiquierte, unsichere Datentypen aus der C-Erblast von C++ sind, die gute C++-Programmierer nur dort verwenden, wo es gar nicht anders geht ([PÜ3.3\(1\)](#)). Warum ist der Puffertyp nicht *string*? Das ist doch ein neuer, sicherer Typ aus der C++-Standardbibliothek? *string* ist für Zeichenketten entworfen. Hier sind aber rohe Daten als Bytestrom zu bearbeiten. Bytes sind *chars*. Dazu passt der Typ *vector<char>* am besten.

BE32.4.4

Entwurfsmuster
Fabrikfunktion

CoderFactory

CoderFactory ist ein Modul, das eine Fabrikfunktion zum Erzeugen von *Coder*-Objekten enthält ([Leitlinie LS.27](#)):

```
static Coder * new_coder (char coder_type, const string & key)
    // Values for 'coder_type':
    // 'e' : Expander
    // 'c' : Caesar
    // 'C' : Composite (Expander, Caesar)
    // 'v' : Vernam
    // 'V' : Composite (Expander, Vernam)
    // default : Copier
```

Die Parameterliste, insbesondere der Parameter *coder_type* und seine Werte, gleicht der Einfachheit halber der des Konstruktors von *FileCoder*. Die beiden Schnittstellen sind jedoch unabhängig voneinander änderbar. Es ist durchaus machbar, die Codertypen an beiden Schnittstellen unterschiedlich zu codieren.

Warum ist *CoderFactory* eine eigene Einheit, nicht integriert in eine andere Einheit, etwa *FileCoder*? *CoderFactory* wird zwar von *FileCoder* benutzt, wir wollen es aber nicht damit vermengen, weil es unabhängig davon wiederverwendbar sein soll.

Warum ist *CoderFactory* ein Modul, keine Klasse? Warum ist es als Klasse mit einer *static* Methode realisiert, nicht mit einem Namensraum? Aus denselben Gründen, die bei *FileIO* genannt sind.

Erweiterbarkeit: Wird der *Coder*-Baum um eine Implementationsklasse erweitert, so muss für *new_coder* ein neues Zeichen als Wert für *coder_type* definiert werden und im Anweisungsteil von *new_coder* ein Fallzweig für dieses Zeichen mit einer Erzeugung eines Objekts der neuen Klasse eingefügt werden.

BE32.4.5**Coder**

Coder ist eine Konzept- und Schnittstellenklasse (*Leitlinie LS.25*), die das Konzept Codierung mit zwei abstrakten Methoden darstellt, die mit Puffern arbeiten:

```
encode (const vector<char> & source_buffer, vector<char> & target_buffer)
decode (const vector<char> & source_buffer, vector<char> & target_buffer)
```

Analog zu *FileCoder* gilt: Da *encode* und *decode* invers zueinander sind, hängen sie funktional stark zusammen. Deshalb gehören sie in dieselbe Einheit (*Leitlinie LS.11*, *Leitlinie LS.12*). Auch hier ist die Zerlegung scheinbar rein prozedural, aber nach BE32.2(7) kommt eine Abfrage für die Ergebnismeldung hinzu (Aktion-Abfrage-Schema).

Entwurfsmuster
Strategie

Coder erlaubt sowohl abstrakte Spezialisierungen als auch Konkretisierungen mit verschiedenen Codierungsverfahren. Konkrete Ableitungsklassen von *Coder* sind konkrete Strategien, die der Kontext *FileCoder* benutzt. Mittels Polymorphie und dynamischem Binden sind Codierungsverfahren (Strategien) zur Laufzeit wählbar (*PÜ32.3(10)*). Da *Coder* mit Puffern arbeitet, sind beliebige Kombinationen effizient möglich (*PÜ32.3(11)*).

BE32.4.6**Copier**

Copier ist ein Beispiel eines konkreten *Coders*, bei dem *encode* und *decode* einfach kopieren. Schnell codiert für den ersten Test!

BE32.4.7**Expander**

Expander ist ein Beispiel eines konkreten *Coders*, bei dem *encode* jedes Byte verdoppelt und *decode* die doppelten Bytes wieder reduziert. Auch schnell codiert für den zweiten Test als Beispiel für ein „Kompressionsverfahren“. Echte Kompressionsverfahren sind aufwändig und nicht Hauptzweck dieses Projekts. Das Blähverfahren ist ein bequemer Platzhalter dafür.

BE32.4.8**Cryptor<KEY>**

Cryptor<KEY> ist eine generische Konzept- und Schnittstellenklasse (*Leitlinie LS.25*), die das Konzept Verschlüsselung als Spezialisierung von Codierung darstellt. Verschlüsselungsverfahren haben immer einen Schlüssel, mit dem sie ver- und entschlüsseln. Der Schlüsseltyp hängt vom Verschlüsselungsverfahren ab.

Warum erscheint der Schlüssel nicht als zusätzlicher Parameter von *encode* und *decode*? Damit könnte Verschlüsselung (*Cryptor*) keine Spezialisierung von Codierung (*Coder*) sein. Ziel ist, Kompression, Verschlüsselung und Fehlerprüfungs-codierung unter einer Abstraktion zu fassen.

Warum erscheint der Schlüssel nicht erst bei konkreten Verschlüsselungsklassen wie *Caesar* und *Vernam*, wo sein Typ feststeht? Damit müssten Codeteile unnötig repliziert werden. Besser ist es, gleiche Codeteile heraus zu faktorisieren. Hier gelingt dies, indem wir den Typ des Schlüssels generisch machen, also von verschiedenen konkreten Schlüsseltypen abstrahieren. Die abstrakte generische Verschlüsselung *Cryptor* hat ein Schlüsselattribut, das gesetzt und abgefragt werden kann.

public:

```
set_key (const KEY & new_key)
KEY key () const {return a_key;}
```

protected:

```
KEY a_key
```

Warum hat *Cryptor* eine Setter-Prozedur, aber keinen Konstruktor, der den Schlüssel setzt? Konstruktoren haben zwei Nachteile: Sie sind klassenspezifisch und nicht ver-

erbbar, und sie dürfen bei jedem Objekt nur einmal bei seiner Erzeugung aufgerufen werden; sie sind also doppelt unflexibel. Hätte *Cryptor* einen parametrisierten Konstruktor, so müsste jede Ableitungsklasse auch einen Konstruktor haben, der den von *Cryptor* aufruft – unnützer Schreibaufwand. Hätte *Cryptor* einen Konstruktor, aber keinen Setter, so könnte jedes Verschlüsselungsobjekt nur mit dem anfangs eingestellten Schlüssel arbeiten. Das wäre unflexibel und Ressourcen verschwendend. Die Setter-Variante ist leichter zu programmieren und flexibler: Sie ist in der abstrakten generischen Klasse fertig codiert und wird dorthin vererbt, wo sie gebraucht wird. In einer Konkretisierung ist nur darauf zu achten, dass sie mit dem Standardwert ihres Schlüssels fehlerfrei funktioniert.

Warum hat *Cryptor* eine Getter-Funktion, die den Schlüssel liefert? Aus Prinzip: Gibt es einen Setter, soll es auch einen Getter dazu geben, damit vergessliche Kunden ein Objekt fragen können, was sie hineingesetzt haben. Ein direkt änderbarer Zustand soll abfragbar sein. Die Beispielimplementation benutzt diesen Getter allerdings nicht, da sie *Cryptor*-Objekte nur als *Coder*-Objekte benutzt.

BE32.4.9

Caesar

Caesar ist der erste exemplarische konkrete *Cryptor* mit dem Schlüsseltyp *char*. *encode* verschiebt jedes Quellbyte per Addition um den Schlüssel.

BE32.4.10

Vernam

Vernam als zweites konkretes *Cryptor*-Beispiel hat den Schlüsseltyp *vector<char>*. *encode* verknüpft ein Quellbyte und ein Schlüsselbyte bitweise mit exklusivem Oder.

BE32.4.11

Entwurfsmuster
Kompositum

Composite

Composite ist ein konkreter *Coder*, der zwei *Coder*-Objekte kombiniert (PÜ32.3(11)). Es ist eine Anwendung des Entwurfsmusters Kompositum (Leitlinie LS.27). Die Komposition und ihre Teile sind gleichartig. *Composite* erhält seine *Coder*-Objekte durch Parameter eines Konstruktors, verbirgt sie dann aber, indem es keine Getter dazu bietet:

public:

```
Composite (Coder * const new_first, Coder * const new_second)
```

```
// precondition: new_first != 0
```

```
// precondition: new_second != 0
```

```
~Composite ()
```

protected:

```
Coder * const first, // implementation invariant: first != 0
```

```
* const second // implementation invariant: second != 0
```

Die *Coder*-Teile müssen außerhalb des *Composite*-Objekts erzeugt werden (Vorbedingungen), gehen mit der Parameterübergabe in den alleinigen Besitz des *Composite*-Objekts über und werden bei dessen Löschung durch den Destruktor *~Composite* gelöscht. Das Löschen von Objekten in C++ ist heikel, weil der Programmierer dafür sorgen muss, dass einerseits keine hängenden Zeiger auf gelöschte Objekte weiter benutzt werden, andererseits keine nicht mehr erreichbaren Objekte als Müll (*garbage*) auf der Halde übrig bleiben.

BE32.5

Implementationen der Module und Klassen

Die vollständigen Implementationen siehe in den Quellcode-Implementationsdateien unter

http://informatik.karlheinz-hug.de/Lehrmaterial/Informatik_3/C++/Coding/Source

Hier folgen Erläuterungen zu Implementationsdetails.

BE32.5.1**Main**

Die *main*-Funktion zerlegen wir top-down prozedural, der Programmierstil ist hier also prozedural. Erster Kandidat für die Zerlegung ist die Ausgabe der Beschreibung der Argumente der Kommandozeile:

```
show_usage_exit (string bad_message)
```

Sie ist für die Benutzbarkeit und Testbarkeit der Lösung an vielen Stellen erforderlich ([PÜ32.5\(2\)](#), [\(3\)](#)). Parametrisierung mit einer Meldung macht sie flexibel einsetzbar.

Die *main*-Funktion hat den längsten Anweisungsblock des Projekts. Den größten Anteil daran hat der Eingabeteil mit der Analyse der bis zu sechs Kommandozeilenargumente. Wem der Block zu lang ist, der möge ihn prozedural zerlegen.

Der Übergang zum Funktionsteil besteht aus vier Anweisungen, nämlich der Initialisierung und dem Aufruf des *FileCoder*-Objekts, das mit Wertsemantik vereinbart ist, weil Zeigersemantik hier keinen Vorteil bietet:

```
FileCoder file_coder(coder_type, key);
if (direction.compare("encode") == 0)
    file_coder.encode(source_filename, target_filename);
else
    file_coder.decode(source_filename, target_filename);
```

Im Ausgabeteil werden vom *FileCoder*-Objekt nur noch die Abfragen *coder_name* und *result* aufgerufen, um deren Ergebnisse auszugeben.

BE32.5.2**FileCoder**

Der Konstruktor von *FileCoder* benutzt die Fabrikfunktion für *Coder*-Objekte. Da ihre Parameterlisten gleich sind, ist der Konstruktoraufruf nur an die Fabrikfunktion weiterzuleiten, sonst wären die Parameter anzupassen:

```
FileCoder::FileCoder (char coder_type, const string & key) :
    coder (CoderFactory::new_coder(coder_type, key))
{
    assert(coder); // implementation invariant
};
```

Der Konstruktorcode zeigt als syntaktische Spezialität von C++ eine Initialisierungsliste, die dem Anweisungsblock vorgelagert ist und die Standardinitialisierung ersetzt [C++98] 12.6.2 *Initializing bases and members* S. 197–200. Im Anweisungsblock ist nur die Einhaltung der Verträge zu prüfen.

Die Abfrage *coder_name* benutzt Sprachelemente der *Runtime Type Information (RTTI)* [C++98] 5.2.8 *Type identification* S. 71–72, 18.5 *Type identification* S. 341–343. Auf das Objekt **coder* wird der Operator *typeid* angewandt, auf das Ergebnis die Abfrage *name*:

```
#include <typeinfo>

string FileCoder::coder_name () const {
    return typeid(*coder).name();
}
```

Laufzeitinformation

encode und *decode* müssen die Quelldatei lesen, codieren, und in die Zieldatei schreiben (Bild BE32.1). Direkte Implementationen von *encode* und *decode* hätten fünf Vorbedingungen, zwei Vereinbarungen und neun Anweisungen gemeinsam und unterschieden sich nur in einer Anweisung, nämlich in

```
coder->encode(source_buffer, target_buffer);
coder->decode(source_buffer, target_buffer);
```

Keinen ernsthaften Programmierer lassen Codeverdopplungen dieses Umfangs ruhig schlafen. Herausfordernd ist hier, dass die unterschiedlichen Anweisungen mitten zwischen zwei Anweisungsblöcken liegen. Prozedurale Zerlegung ist mit den *FileIO*-Diensten schon ausgeschöpft; der übrige Code besteht aus den Vereinbarungen der Pufferobjekte und den Ergebnisbehandlungen. Da *Coder::encode* und *Coder::decode* dieselbe Signatur besitzen, bieten sich Methodenzeiger (*method pointer*, *member function pointer*, *pointer to member function*) an [C++98] 5.3.1 *Unary operators* S. 76–77, 5.5 *Pointer-to-member operators* S. 83, 8.3.3 *Pointers to members* S. 133. Aus *encode* und *decode* abstrahieren wir eine geschützte Methode *code* heraus, die als zusätzlichen Parameter den Methodenzeiger

Methodenzeiger

```
void (Coder::*method)(const vector<char> &, vector<char> &)
```

hat, der zu *encode* und *decode* passt. Die Vereinbarung von *code* muss in die Vereinbarung der *FileCoder*-Klasse in der Header-Datei, gehört aber freilich nicht zur Kundenschnittstelle der Klasse:

protected:

```
code (const string & source_filename, const string & target_filename,  
void (Coder::*method) (const vector<char> &, vector<char> &))
```

In der Definition von *code* wird der übergebene Methodenzeiger *Coder::**method** auf das Objekt angewandt, auf das *coder* zeigt, und dereferenziert:

```
(coder->*method)(source_buffer, target_buffer);
```

Aufgerufen wird *code* von *encode* und *decode*. Beispielsweise übergibt *encode* als Methodenzeigerwert *&Coder::encode*:

```
FileCoder::encode (const string & source_filename, const string & target_filename) {  
    code(source_filename, target_filename, &Coder::encode);  
};
```

Man beachte, dass „&“ in *&Coder::encode* kein Adressoperator ist! Beim Anwenden des Methodenzeigers wird (wie in objektorientiertem Kontext zu erwarten) dynamisch gebunden, d.h. beim Aufruf

```
(coder->*(&Coder::encode))(source_buffer, target_buffer);
```

wird das *encode* des dynamischen Typs von *coder*, also des Typs des Objekts, auf das *coder* zeigt, gebunden.

BE32.5.3

FileIO

Die *fstream*-Klassen der C++-Standardbibliothek bieten vielfältige Methoden für Dateizugriffe. Lesen und Schreiben können byte-, zeilen- oder blockweise erfolgen. Details entnehme man Beschreibungen der C++-Standardbibliothek.

Zum Öffnen der Dateien benutzen wir einfacherweise Konstruktoren von *ifstream* und *ofstream* anstelle der *open*-Methode. Der Öffnungsmodus ist *ios::binary* für rohe Byteströme, damit beliebige Dateien bearbeitet werden können (PÜ32.3(14)). Nicht fehlen darf die sofortige Abfrage, ob das Öffnen geklappt hat (Aktion-Abfrage-Schema):

```
ifstream stream(filename.c_str(), ifstream::binary);  
if (stream.fail())  
    a_result = failed_open;  
else {  
    // good case  
}
```

FileIO soll komplette Dateiinhalte am Stück lesen und schreiben. Vor dem Lesen muss die Größe des Puffers (des Ausgabeparameters *vector<char> & buffer*) auf die Größe der Datei eingestellt werden:

```
buffer.resize(size);
```

Also muss *size* vorher auf die Dateigröße eingestellt werden; dazu ist die Leseposition auf das Dateiende zu setzen und die Leseposition abzufragen:

```
stream.seekg(0, ios::end);
streamsize size = stream.tellg();
```

(Das kryptische Suffix „g“ steht für „get pointer“.) Freilich reduzieren wir die drei Anweisungen auf zwei. Nicht vergessen dürfen wir die Anweisung, die die Leseposition wieder auf den Dateianfang setzt:

```
stream.seekg(0, ios::beg);
```

Die Leseoperation hat die Signatur

```
istream & read (char *s, streamsize n)
```

Hier trifft der C++-Reihungstyp `vector<char>` auf den altertümlichen C-Zeigertyp `char*`. Kein Grund, auf `vector<char>` zu verzichten! Denn es gibt eine einfache Anpassung, wenn auch nicht ohne Benutzung eines speicherunsicheren Konstrukts, nämlich des Adressoperators „&“. Der `vector`-Typ garantiert (im Unterschied zum `string`-Typ!), dass alle Reihungselemente konsekutiv zusammenhängend im Speicher liegen. Auf das Anfangselement `buffer[0]` (vom Typ `char`) kann der Adressoperator angewandt werden, somit ist `&buffer[0]` vom Typ `char*`. Die Anweisung

```
stream.read(&buffer[0], buffer.size());
```

schreibt `buffer.size` **chars** von der Position `&buffer[0]` an konsekutiv zusammenhängend in den Speicher. Der `vector`-Typ und die zuvor korrekt eingestellte Puffergröße garantieren hier, dass keine nicht vorgesehenen Speicherbereiche beschrieben werden. Die Anweisung ist selbst dann korrekt, wenn `buffer.size == 0` ist.

Unsicheres
Sprachkonstrukt

Der Adressoperator ist in einer Hochsprache ein maschinennaher, nicht speichersicherer Fremdkörper. C/C++-Programmierer können ihn großzügig verwenden und damit beliebige Anweisungen unsicher machen. Dagegen müssen in C# unsichere Blöcke mit `unsafe` markiert sein. Deshalb sollte man auch in C/C++ unsichere Blöcke markieren:

```
{// unsafe!
  stream.read(&buffer[0], buffer.size());
}
```

Auch beim Schreiben wird der Adressoperator verwendet. Die Unsicherheit ist aber schwächer als beim Lesen: Es können keine Daten zerschrotet, sondern nur falsche Daten in die Zielfeile geschrieben werden.

BE32.5.4

CoderFactory

Die Fabrikfunktion `new_coder` enthält nur eine Mehrfachauswahl, die das `coder_type` entsprechende `Coder`-Objekt erzeugt. `Cryptor`-Objekten wird der Schlüssel `key` nach der Objekterzeugung per Setter-Aufruf übergeben. Beim `Caesar`-Objekt muss der Wert von `key` vom Typ `string` in den Typ `char` gewandelt werden. Das geschieht in drei Schritten: von `string` mit `c_str` nach `char*`, von `char*` mit `atoi` nach `int`, implizit von `int` nach `char`:

```
caesar->set_key(atoi(key.c_str()));
```

Beim `Vernam`-Objekt muss der Wert von `key` vom Typ `string` in den Typ `vector<char>` gewandelt werden. Das geschieht durch einen Konstruktoraufruf von `vector<char>`, der mit den `string`-Iteratoren `key.begin` und `key.end` parametrisiert ist:

```
vernarn->set_key(vector<char>(key.begin(), key.end()));
```

Beim `Composite`-Objekt ruft sich `new_coder` zweimal rekursiv auf:

```
case 'C':
  return new Composite(new_coder('e', ""), new_coder('c', key));
```

Das Beispiel zeigt, wie sich mit diesem Entwurf beliebig viele Codierungen einfach kombinieren lassen ([PÜ32.3\(11\)](#)).

BE32.5.5**Copier**

Man beachte, dass durch

```
target_buffer = source_buffer;
```

nicht der gesamte Dateiinhalt im Hauptspeicher kopiert wird, sondern nur eine kleine Datenstruktur, die einen Zeiger auf den Dateiinhalt enthält.

BE32.5.6**Expander**

In *encode* wird durch

```
target_buffer.resize(2 * source_buffer.size());
```

in *target_buffer* doppelt so viel Platz wie in *source_buffer* beschafft, damit jedes Byte verdoppelt werden kann. *decode* scheitert, wenn nicht jedes Byte in *source_buffer* doppelt vorkommt.

BE32.5.7**Caesar**

Byteweise verschieben – ein trivialer Algorithmus, der mit fünf Anweisungen inklusive einer Schleife auskommt. Dass mki-Studenten die Hinweise in [PÜ32.3\(6\)](#) ignorieren, stattdessen komplizierte, seitenlang wuchernde Caesar-Mutationen naiv aus minderwertigen Webseiten kopieren und als ihre „Lösungen“ dieses Projekts präsentieren, weckt Zweifel, ob sich das Projekt didaktisch gut als mki-I3P-Übung eignet.

Ein gute Implementation von *decode* repliziert den Algorithmus *nicht*, so einfach er auch sein mag, sondern ruft *encode* mit dem negativen Schlüssel auf. Leider ignorieren die meisten Studenten auch diesen Hinweis in [PÜ32.3\(6\)](#) und duplizieren ihren obskuren Bläh-Caesar.

BE32.5.8**Vernam**

Bitweise exklusiv verodern – ein genialer, kompakter Algorithmus, dessen Implementation einfach und im Hinweis in [PÜ32.3\(7\)](#) skizziert ist. Warum wagen sich nur wenige Studenten daran? Ein gute Implementation von *decode* repliziert den Algorithmus *nicht*, sondern ruft *encode* mit demselben Schlüssel auf.

BE32.5.9**Composite**

Wie der *FileCoder*-Konstruktorcode hat auch der *Composite*-Konstruktorcode eine Initialisierungsliste zum Initialisieren der Erbschaft und der eigenen Daten:

```
Composite::Composite (Coder * const new_first, Coder * const new_second) :
    Coder(),
    first (new_first),
    second (new_second)
{
    assert(new_first); // precondition
    assert(new_second); // precondition
    assert(first); // implementation invariant
    assert(second); // implementation invariant
}
```

Um zwei *Coder* zu einem dritten *Coder* zu kombinieren, ist in *encode* und *decode* nur ein Zwischenpuffer *interim_buffer* zwischen dem Quellpuffer des ersten und dem Zielpuffer des zweiten *Coders* anzulegen. Beim Decodieren ist die Reihenfolge umgekehrt wie beim Codieren.

BE32.6**Zwischenbilanz**

Messen

Der Entwurf in BE32.2 bis BE32.5 heißt ab hier „erste, einfache Entwurfsvariante“. Tabelle BE32.1 gibt mit verschiedenen Metriken einen Überblick zum Umfang der Einheiten. Die durchschnittlichen Anzahlen der Parameter und Anweisungen pro Prozedur sind meist klein und zeigen, dass das Ziel der Zerlegung, die Komplexität des Problems zu beherrschen, erreicht ist. Einzige Ausnahme und Kandidat für weitere prozedurale Zerlegung ist die *main*-Funktion mit 45 Anweisungen.

Analysieren

Kombinierbarkeit begrenzt

Welche Schwächen weist die erste Entwurfsvariante auf? Die Benutzungsoberfläche haben wir in BE32.4.1 mit der EBNF-Regel

```
function = 'copy' | 'compress' | ( 'crypt' | 'compress+crypt' ) method .
```

exemplarisch auf die Codierungskombination „Kompression gefolgt von einer Verschlüsselung“ beschränkt. Diese Beschränkung im E/A-Teil ist akzeptabel, so lange der Funktionsteil flexibel gestaltet ist. Die Schnittstelle des *Composite*-Konstruktors

```
Composite (Coder * const new_first, Coder * const new_second)
```

in BE32.4.11 ist in der Tat flexibel einsetzbar; sie bedarf keiner Änderung. In den Schnittstellen des *FileCoder*-Konstruktors in BE32.4.2 und der *CoderFactory*-Fabrikfunktion *new_coder* in BE32.4.4 haben wir mit

```
// Values for 'coder_type':
// 'C' : Composite (Expander, Caesar)
// 'V' : Composite (Expander, Vernam)
```

zwei Kombinationen festgelegt, wobei *Expander* exemplarisch für ein Kompressionsverfahren steht. In BE32.5.4 behaupten wir,

```
case 'C' :
    return new Composite(new_coder('e', ""), new_coder('c', key));
```

zeige, „wie sich mit diesem Entwurf beliebig viele Codierungen einfach kombinieren lassen“. Das ist zwar theoretisch richtig, stößt aber an praktische Grenzen.

Kombinationen statisch festgelegt

Erstens sind die Kombinationen in den Schnittstellen des *FileCoder*-Konstruktors und der Fabrikfunktion zur Übersetzungszeit festgelegt, d.h. zur Laufzeit lassen sich nur zuvor festgelegte Kombinationen erzeugen. Zweitens ist die kombinatorische Vielfalt statisch schwer zu beherrschen:

Kombinationsvielfalt schwer handhabbar

Angenommen, an die Stelle von *Expander* treten m Kompressionsklassen und zu *Cryptor* gibt es n Verschlüsselungsklassen, so ergeben sich $m * n$ Kombinationen von einer Kompression gefolgt von einer Verschlüsselung. Genau k Verschlüsselungen nacheinander ergeben n^k Kombinationen. Höchstens eine Kompression gefolgt von höchstens k Verschlüsselungen ergeben $(m + 1) * (n^{k+1} - 1) / (n - 1)$ Kombinationen. Für jede Kombination ist je ein codierendes Zeichen für die Schnittstellen des *FileCoder*-Konstruktors und der Fabrikfunktion zu finden. Da es nur 256 verschiedene und noch weniger druckbare Zeichen gibt, dürfen m , n , k nicht groß werden. Ein maximales Tripel ist z.B. $m = 4$, $n = k = 3$.

Als Ausweg wäre für *coder_type* ein Typ mit größerem Wertebereich wählbar. Näher liegt jedoch der Schluss, dass beliebiges Kombinieren von Codierungen zu einer kombinatorischen Explosion führt, die nicht handhabbar ist, indem jede Kombination statisch codiert und in der Mehrfachauswahlanweisung der Fabrikfunktion in einem eigenen Fallzweig erzeugt wird.

Coder-Codierung problematisch

Bedenklich ist auch die Codierung der *Coder*-Typen durch Zeichen. Vorteile der Lösung sind, dass sie einfach zu realisieren ist, die Schnittstelle zwischen Funktions- und E/A-Teil nicht belastet, und der Funktionsteil erweiterbar ist, ohne dass der E/A-Teil nachübersetzt werden muss (relevant wenn der Funktionsteil viele Kunden hat).

Tabelle BE32.1 Statistik der Einheiten der ersten, einfachen Entwurfsvariante

Einheit	Dateien	Modul, Klasse	Prozeduren ^a	Parameter	Anwei- sungen	Parameter / Prozedur	Anweisungen / Prozedur	Quelltext- seiten	Codegröße (KiloByte)
<i>main</i>	1	M	2	3	47	1,5	23,5	2	42
<i>FileCoder</i>	2	K	7	9	16	1,3	2,3	2	42
<i>FileIO</i>	2	M	3	4	18	1,3	6	2	78
<i>CoderFactory</i>	2	M	1	2	11	2	11	1	103
<i>Coder</i>	1	K	3	4	1	1,3	0,3	1	-
<i>Copier</i>	2	K	2	4	2	2	1	1	53
<i>Expander</i>	2	K	2	4	17	2	8,5	1	53
<i>Cryptor<KEY></i>	1	K	2	1	2	0,5	1	1	-
<i>Caesar</i>	2	K	2	4	8	2	4	1	54
<i>Vernam</i>	2	K	2	4	6	2	3	1	77
<i>Composite</i>	2	K	4	6	17	1,5	4,3	1	26
Gesamt	19	3 M, 8 K	30	45	145	1,5	4,8	14	1218 (exe)

^a Vereinbarungen von Prozeduren, Konstruktoren, Destruktoren, Gettern inklusive Redeklarationen.

Nachteile der Lösung sind: Sie ist nicht statisch typsicher, da falsche Zeichenwerte nicht zur Übersetzungszeit erkannt werden. Ihre Realisierung verstößt gegen die Programmierleitlinie, die literale Konstanten in Anweisungsteilen missbilligt. Eine softwaretechnisch sauberere und besser lesbare Implementation sieht so aus:

```
public:
    // Values for 'coder_type':
    static const char expander                = 'e';
    static const char caesar                  = 'c';
    static const char composite_expander_caesar = 'C';
    ...

case composite_expander_caesar :
    return new Composite(new_coder(expander, ""), new_coder(caesar, key));
```

Sie kommt einer statisch typsicheren Realisierung mit einem Aufzählungstyp nahe:

```
public:
    enum CoderType {expander, caesar, composite_expander_caesar, ...};
```

Kommandosyntax
seltsam

Ein Schönheitsfehler der ersten Entwurfsvariante ist, dass in der Kommandozeile zum Kopieren eine Richtung anzugeben ist (*encode copy* oder *decode copy*).

Codernamen seltsam
und ungenau

Als weiteres Manko lässt *coder_name* sein Ergebnis mit einem unerwünschten Zeichen beginnen und bei Kombinationen keine Information zu den Teilen liefern.

BE32.7

Zweite, flexible Entwurfsvariante

Die „zweite, flexible Entwurfsvariante“ vermeidet die in BE32.6 erkannten Schwächen der ersten Entwurfsvariante und erfüllt diese Anforderungen:

- (1) Der Funktionsteil ermöglicht es, beliebig viele Codierungen dynamisch zu kombinieren.
- (2) Einfache Codierungen sind statisch typsicher und softwaretechnisch sauber codiert.
- (3) Die Syntax der Kommandozeilenargumente erlaubt beliebig viele Verschlüsselungen und vermeidet die *encode/decode-copy*-Anomalie.
- (4) Zur Testbarkeit erscheinen die korrekten *Coder*-Namen und bei Kombinationen auch die Namen der Teile.

Dateien der beiden Varianten unterscheiden sich ggf. im Suffix „1“ oder „2“ oder durch ihre Lage in den Unterverzeichnissen *shared*, *1_simple*, *2_flexible*.

BE32.7.1

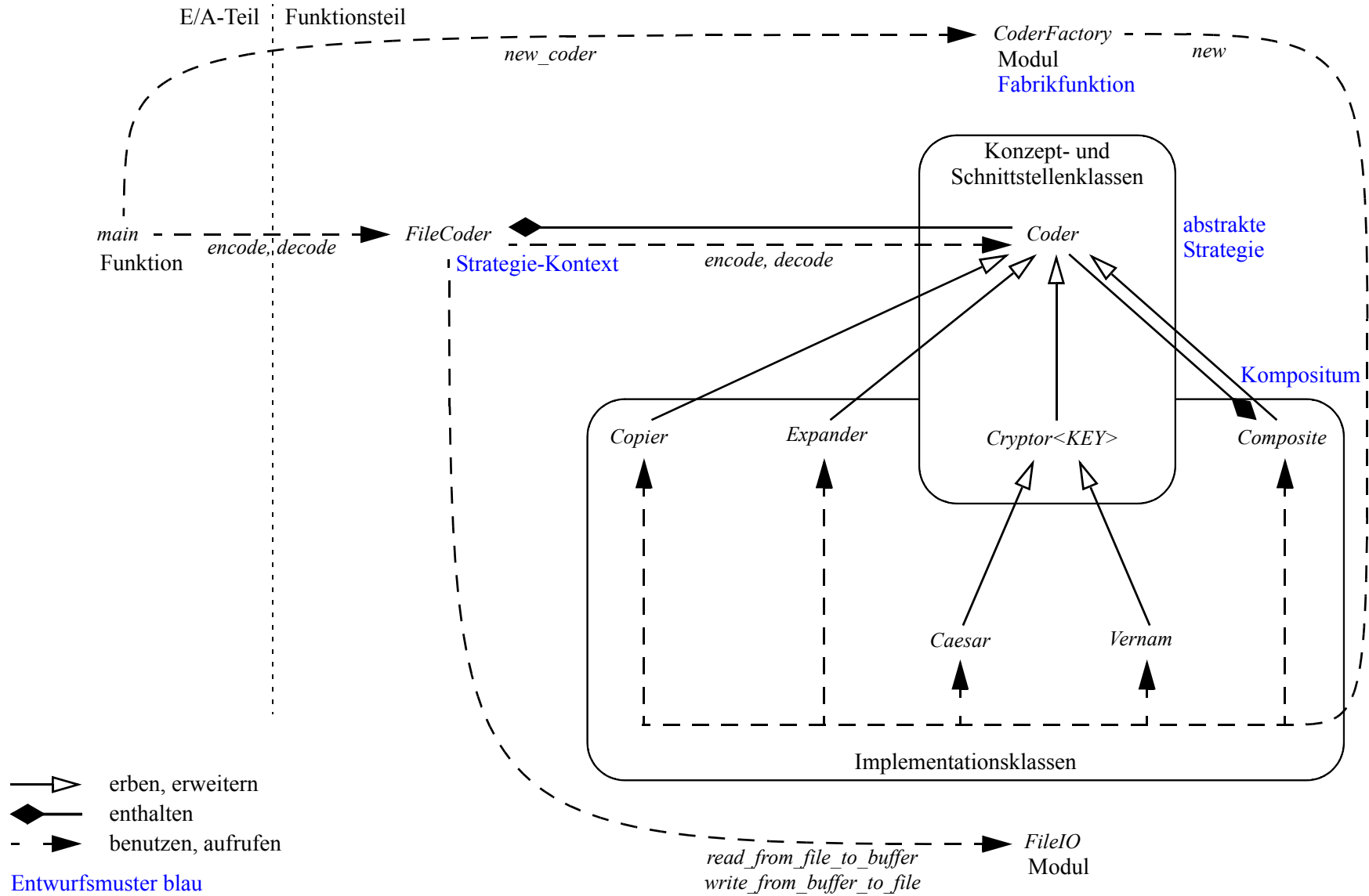
Entwurfsentscheidungen und Zerlegung

Um die Schwäche der statisch begrenzten Kombinierbarkeit von Codierungen zu beseitigen und Anforderung (1) zu erfüllen, müssen wir die kombinatorische Explosion an den Schnittstellen des *FileCoder*-Konstruktors und der *CoderFactory*-Fabrikfunktion verhindern. Dazu müssen wir das Erzeugen von Kombinationsobjekten teilweise aus der Fabrikfunktion auslagern. Nur zu *FileCoder* auslagern würde dessen Schnittstelle nicht vereinfachen. Ein Teil der Komplexität des Erzeugens von Kombinationsobjekten muss zum Kunden von *FileCoder*, also zum E/A-Teil, zur *main*-Funktion migrieren. Folglich muss der E/A-Teil die *CoderFactory*-Fabrikfunktion aufrufen können. Die allgemeine Entwurfsentscheidung BE32.2(6) zum **Funktionsteil** wird so geändert:

Die funktionalen Einheiten, die die Schnittstelle zum E/A-Teil bilden, sind die Klassen *CoderFactory*, *Coder* und *FileCoder*. *FileCoder* verliert seine Rolle als Fassade. Der E/A-Teil kennt nur den Typ *Coder*, benutzt dessen Schnittstelle aber nicht.

Bild BE32.3 stellt den geänderten Entwurf dar.

Bild BE32.3 Modul- und Klassendiagramm der zweiten, flexiblen Entwurfsvariante



Der Vorteil der erhöhten Flexibilität nach Anforderung (1) kostet diese Nachteile: Die Schnittstelle zwischen E/A-Teil und Funktionsteil ist breiter, ihre Kopplung stärker als bei der ersten Entwurfsvariante. Statt einer Fassadenklasse kennt der E/A-Teil jetzt das Modul *CoderFactory* und die Klassen *Coder* und *FileCoder*. Der E/A-Teil soll aber nur *CoderFactory* und *FileCoder* aufrufen, nicht *Coder*, von dem es nur den Namen zur Kommunikation zwischen *CoderFactory* und *FileCoder* braucht. Ob jeder Kundenprogrammierer damit diszipliniert umgeht?

Anforderung (1) betrifft *FileCoder* und *CoderFactory*; Anforderung (2) *CoderFactory*; Anforderung (3) *main*; Anforderung (4) *FileCoder*, *Coder* und *Composite*. Das Folgende beschreibt die geänderten Schnittstellen und Implementationen unter je einem Abschnitt.

BE32.7.2

Main

Damit *main* die Anforderung (3) erfüllt, restrukturieren wir seine Benutzungsoberfläche so, dass sie höchstens eine Kompression gefolgt von beliebig vielen Verschlüsselungen ermöglicht und die in BE32.6 erkannte Syntaxschwäche verliert. Die flexibilisierten EBNF-Regeln lauten:

```

command      = command_name source_filename target_filename codings .
command_name  = 'Coding2' .
source_filename = Pfadname .
target_filename = Pfadname .
codings       = 'copy' | ( 'encode' | 'decode' ) ( compress | crypt ) { crypt } .
compress      = 'compress1' | 'compress2' .
crypt         = 'caesar' caesarkey | 'vernam' vernamkey .
caesarkey     = integer .
vernamkey     = string .

```

Die definierenden Ausdrücke der Nichtterminale *compress* und *crypt* lassen sich leicht um beliebig viele Alternativen erweitern.

Die lange erste *main*-Variante zerlegen wir für die zweite Variante weiter prozedural. Aufrufe der kleinen Prozedur

```
check (bool good, string bad_message)
```

im Stil einer Zusicherung komprimieren in *main* viele Stellen, in denen eine Bedingung *good* zu prüfen und im *false*-Fall der Programmablauf mit der Meldung *bad_message* zu beenden ist.

Da in der EBNF-Regel für *codings* das Nichtterminal *crypt* zweimal vorkommt, lohnt sich die Funktion

```
CoderFactory::CoderType coder_type (string coding),
```

die *crypt*-Zeichenketten von der Kommandozeile auf Werte des Aufzählungstyps *CoderType* abbildet (BE32.7.4).

BE32.7.3

FileCoder

Um Anforderung (1) zu erfüllen, ist an der Schnittstelle von *FileCoder* nur die Signatur des Konstruktors zu ändern. Der flexible *FileCoder* verbirgt sein *Coder*-Objekt zwar wie der einfache *FileCoder*, erhält es aber durch einen Parameter des Konstruktors, statt es selbst zu erzeugen. Das *Coder*-Objekt muss außerhalb des *FileCoder*-Objekts erzeugt werden (Vorbedingung) und geht mit der Parameterübergabe in den alleinigen Besitz des *FileCoder*-Objekts über:

```
FileCoder (Coder * const new_coder)
// precondition: new_coder != 0
```

Die Implementation des Konstruktors speichert den übergebenen Zeiger *new_coder* im geschützten Attribut *coder*.

Um Anforderung (4) zu erfüllen, wird *coder_name* durch Weiterleitung an die neue Abfrage *name* von *Coder* reimplementiert:

```
string FileCoder::coder_name () const {
    return coder->name();
}
```

BE32.7.4

CoderFactory

Um Anforderung (1) zu erfüllen, erhält die flexible Fabrikfunktion *new_coder* als zusätzlichen Parameter den Zeiger *first* auf ein *Coder*-Objekt, der Wertebereich von *coder_type* wird zunächst auf einfache *Coder* reduziert:

```
static Coder * new_coder (Coder * const first, char coder_type, const string & key)
// Values for 'coder_type':
// 'e' : Expander
// 'c' : Caesar
// 'v' : Vernam
// default : Copier
// If 'first' != 0, then it is combined with a coder described by 'coder_type'.
```

Zeigt *first* auf nichts, so liefert *new_coder* ein einfaches *Coder*-Objekt gemäß *coder_type*, sonst eine Kombination aus dem *first*-Objekt und dem *coder_type*-Objekt. Das bedeutet, dass Kunden der Fabrikfunktion die Anfangsteile von Kombinationen bei sich verwalten müssen.

Um Anforderung (2) zu erfüllen, stellt *CoderFactory* den Aufzählungstyp

```
enum CoderType {copier, expander, caesar, vernam};
```

mit verständlichen Namen für die einfachen *Coder*-Typen bereit und setzt ihn in der Signatur der Fabrikfunktion ein:

```
static Coder * new_coder (Coder * const first, CoderType coder_type, const string & key);
// If 'first' != 0, then it is combined with a coder described by 'coder_type'.
```

Die Implementation der flexiblen Fabrikfunktion ist prinzipiell einfacher als die der einfachen Fabrikfunktion, da sie weniger Fälle als jene zu unterscheiden hat und keine rekursiven Aufrufe benötigt:

```
Coder * CoderFactory::new_coder
(Coder * const first, CoderType coder_type, const string & key) {
    Coder * result = 0;
    switch (coder_type) {
        case simple_coder :
            result = new SimpleCoder();
            ...
    }
    if (first) result = new Composite(first, result);
    assert(result); // postcondition
    return result;
}
```

BE32.7.5

Coder und Composite

Um Anforderung (4) zu erfüllen, stellt *Coder* die Abfrage

```
string name () const
```

bereit und implementiert sie mit einem kleinen Hack, der Zeigerarithmetik nutzt, um das unerwünschte erste Zeichen des Namens zu kappen:

```
string Coder::name () const {
    const char * name = typeid(*this).name();
    return ++name;
}
```

Composite überschreibt (redefiniert) *name* mit drei *name*-Aufrufen:

```
string Composite::name () const {
    return Coder::name() + "(" + first->name() + ", " + second->name() + ")";
}
```

So ist mit wenig Aufwand viel Effekt erzielt.

BE32.7.6

Bilanz

Was hat sich von der ersten zur zweiten Entwurfsvariante quantitativ geändert? Tabelle BE32.2 zeigt die geänderten Werte. Die Anzahlen der Module, Klassen und Quelltextseiten sind geblieben. Dazugekommen sind die Datei *Coder.cpp* wegen *name*, 4 Prozeduren, 3 Parameter, 25 Anweisungen. Die Anzahl der Parameter pro Prozedur ist auf 1,4 gesunken, die Anzahl der Anweisungen pro Prozedur ist auf 5 gestiegen, die Codegröße auf 1271 KiloByte. Also sind die quantitativen Änderungen nicht signifikant.

Tabelle BE32.2 Statistik der Einheiten der zweiten, flexiblen Entwurfsvariante

Einheit	Proze- duren ^a	Parameter	Anwei- sungen	Parameter / Prozedur	Anweisungen / Prozedur	Codegröße (KB)
<i>main</i>	4	6	63	1,5	15,8	47
<i>FileCoder</i>	7	8	16	1,1	2,3	41
<i>CoderFactory</i>	1	3	17	3	17	101
<i>Coder</i>	4	4	3	1	0,8	22
<i>Composite</i>	5	6	18	1,2	3,6	44
Gesamt	34	48	170	1,4	5	1271 (exe)

^a Vereinbarungen von Prozeduren, Konstruktoren, Destruktoren, Gettern inklusive Redeklarationen.

Die qualitativen Verbesserungen sind mehr Flexibilität und Gebrauchstauglichkeit. Mit geringem Aufwand und ohne große Strukturänderung haben wir die Qualität der zweiten Entwurfsvariante gegenüber der ersten verbessert. Was an diesem Beispielpunkt zu lernen ist, sei in eine Leitlinie abstrahiert:

Leitlinie BE32.1

Restrukturieren lohnt sich

Mach den ersten Programmentwurf möglichst gut und implementiere ihn! Gib dich damit nicht zufrieden, sondern analysiere und kritisiere Entwurf und Implementation! Verbessere dann beides!

BE32.A

Literatur

[C++98] ISO/IEC 14882: *International Standard, Programming Languages – C++, Langages de programmation – C++*. American National Standards Institute, New York (1998-09-01) First edition, 776 S., http://www-d0.fnal.gov/~dladams/cxx_standard.pdf (Zugriff 2012-08-03)