kleiner kommunikation
max@kleiner.com

1 Das Decorator Pattern als Generator

(Gut Dekoriert ist halb gebaut)

Gelegentlich wollen wir Methoden einzelner Objekte ändern, ohne die bestehenden Klassen zu verändern. Vielleicht kennen Sie die „Tales of Crypt“ (Geschichten aus der Gruft), die mich zu folgendem Artikel verführt haben. Anhand einer kleinen Verschlüsselungsroutine möchte ich das Decorator Pattern vorstellen, um Ihnen einen weiteren Baustein der OO-Welt näherzubringen. Der klassische Dekorierer hat zudem Eigenschaften des Composite-Pattern, welches ziemlich zentral in der Patternwelt dasteht.

1.1 Dynamisch Erweitern

Das Design Pattern Decorator gehört zu den objektbasierten Strukturmustern. Es lässt sich folgendermassen definieren: Erweitere ein Objekt dynamisch um Zuständigkeiten ohne viele neue Unterklassen bilden zu müssen. Solche Dekorierer bieten eine flexible Alternative zur Unterklassenbildung, die meist mit klassenbasierten Mustern wie Vererbung gelöst werden, um die Basisfunktionalität der Klasse dynamisch zu erweitern. Aber durch die Vererbung alleine ist nur eine statische Erweiterung möglich, die auch in die bestehende Vererbungshierarchie eingreifen muss.

Das Problem stellt sich so dar: Die Funktionalität eines Objekts soll erweitert werden ohne die bestehende Klasse zu ändern. Die Vererbung ist inflexibel und nicht zur Laufzeit möglich, zumal Klassendefinitionen versteckt sein können oder aus anderen Gründen zum Ableiten nicht ersichtlich sind. Mit dem Decorator lässt sich nun die zusätzliche Funktionalität um das Objekt umschliessen (auf gut deutsch einhüllend), sozusagen als funktionelle Dekoration. Dies hat auch den Vorteil, dass die Vererbungshierarchie übersichtlich bleibt und die erweiterte Funktionalität auch wieder entfernt werden kann. Weitere Vorteile sind:

- Bessere Flexibilität im Vergleich zu statischer Vererbung
- erlaubt schlanke, leichte Komponentenklassen
- Dekorierer lassen sich rekursiv schachteln
- Stabile Basisklasse bleibt unangetastet (ohne nachträglich virtual einzubauen)
- Kombinierbar mit dem Composite Pattern

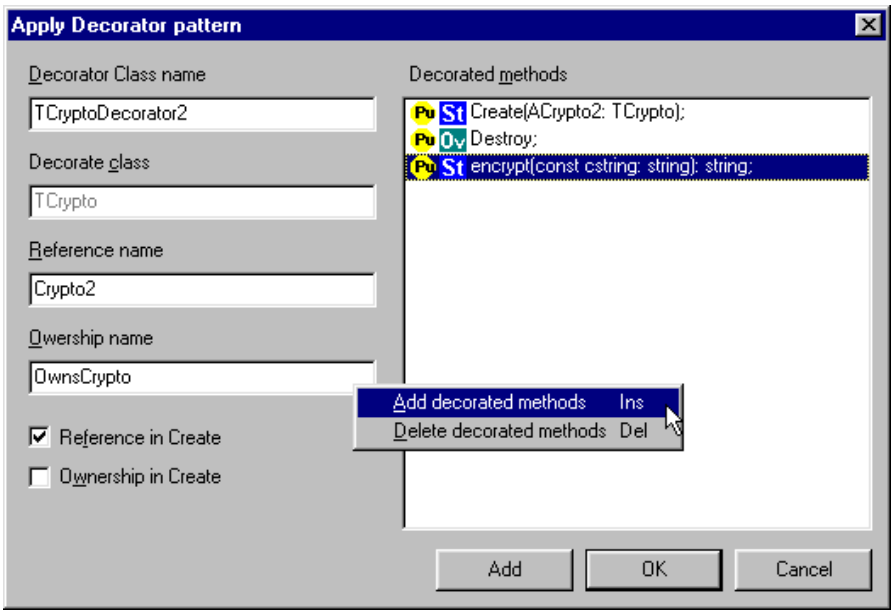
Der Decorator ist ein spezieller Fall des Composite, da nur genau eine Rückführung (1..1) vorhanden ist. Denn der Decorator leitet seine Aufrufe an sein Komponentenobjekt weiter und verwaltet keine Listen. Somit lassen sich auch vor oder nach dem Weiterleiten des Aufrufes zusätzlich Operationen ausführen, siehe unser Beispiel der Kryptoklasse. Als mögliche Anwendung sind auch Kompressoren, Filter für Soundboxen, Formatierer oder Streamer anzutreffen, die meistens eine Basiskomponente aufweisen und zusätzliche Klassen als Dekorierer.

Als erstes benötigen wir eine kleine Basiskomponente, die exemplarisch einen String verschlüsselt. Unser Ziel ist es, eine einfache XOR-Verschlüsselung zu implementieren, wobei das Gebiet der Kryptologie am Schluss nur kurz gestreift wird, da der Schwerpunkt ja auf dem Design liegt. Bevor wir mit dem Ableiten des Pattern beginnen, bauen wir mal die Klasse, die dann dekoriert, d.h. zur Laufzeit erweitert wird:

```
TCrypto = class (TObject)
  private
    FcryptKey: LongInt;
    FReferenceCnt: Integer;
  protected
    procedure SetcryptKey(const Value: LongInt);
    procedure SetReferenced(IsReferenced: Boolean);
  public
```

```
    procedure AddReference;
    function encrypt(const cstring: string): string;
    property cryptKey: LongInt read FcryptKey write SetcryptKey;
  end;
```

Viele Tools, wie auch ModelMaker, bieten einen Patterngenerator an, den wir nun exemplarisch aktivieren. Die Möglichkeit bestehende Klassen mit Patterns zu erweitern, ist bei den Strukturmustern am meisten gegeben (wie auch Adapter oder Wrapper). Bei den Erzeuger- oder Verhaltensmustern muss teilweise auch die bestehende Klasse angepasst oder von Anfang an geplant werden. Der folgende Dialog zeigt die nötigen Angaben, die zur Generierung des Pattern vonnöten sind, vor allem die zu erweiterten (zu dekorierenden) Funktionen, in unserem Fall die Methode `encrypt`, muss man in der bestehenden Klasse markieren, und lassen sich dann in einem Rutsch hinzufügen. Der Generator erzeugt also eine neue Klasse `TCryptoDecorator2`.



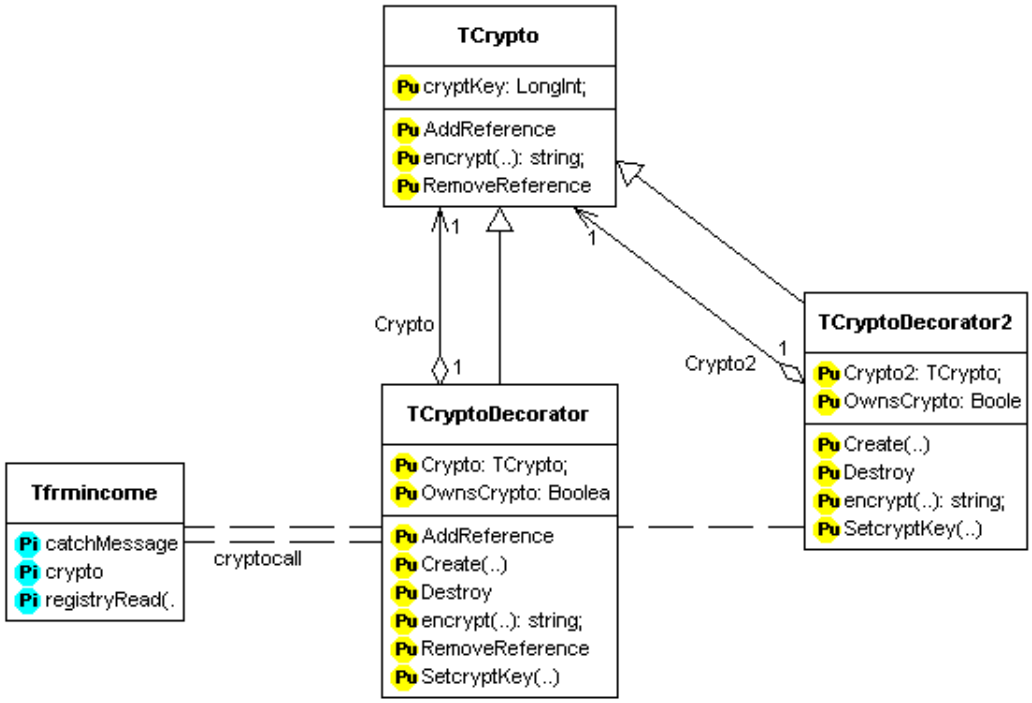
Mit ein paar Handgriffen zum Decorator

Die Unit *cryptobox.pas* wird schlussendlich aus drei Klassen bestehen, der Komponentenklasse und den zwei Dekorierern, die ich (resp. ModelMaker) nacheinander erzeugt hat. Die abgeleitete Klasse als Dekorierer benötigt noch einen Referenznamen (Crypto2), der dann als Zeiger zur Basisklasse dient. Die generierte Klasse hat nach einigen Sekunden folgende Struktur:

```
TcryptoDecorator2 = class (TCrypto)
  private
    FCrypto: TCrypto;
    function GetCrypto: TCrypto;
    procedure SetCrypto(Value: TCrypto);
  public
    constructor Create(ACrypto: TCrypto);
    destructor Destroy; override;
    procedure AddReference;
    function encrypt(const cstring: string): string;
    procedure SetcryptKey(const Value: LongInt);
    property Crypto2: TCrypto read GetCrypto write SetCrypto;
end;
```

Auffallend ist der Konstruktor, der als Parameter die Instanz der Basisklasse erhält. Welche Idee dahinter steht, sehen wir gleich später. Zusätzlich erhält die neue Klasse den erwähnten member `Crypto2` als property, der dann zur Laufzeit die Instanz der Basisklasse `TCrypto` erhält.

Das nun Typische am Decorator ist die doppelte Relation von Vererbung und Aggregation zwischen den Klassen, welche maximale Flexibilität erlaubt und sich durch eine Aggregation gut steuern lässt. Schlussendlich bestimmt ja der Klient zur Laufzeit, wann welcher Dekorierer zum Zuge kommt.



Das Klassendiagramm des Decorator

Der eigentliche Konstruktor übernimmt die Referenz der Basiskomponente und weist sie dem member `Crypto` oder `Crypto2` zu. Gewissermassen besteht eine Ähnlichkeit zum Proxy-Pattern, welches „nur“ eine reine Weiterleitung bewirkt und keine Erweiterung an sich ist. Nicht zu vergessen sei das Freigeben des members im Destruktor:

```
constructor TCryptoDecorator.Create(ACrypto: TCrypto);
begin
  inherited Create;
  Crypto := ACrypto;
end;

destructor TCryptoDecorator.Destroy;
begin
  Crypto.free;
  Crypto := NIL;
  inherited Destroy;
end;
```

1.2 Client kann wählen

Nachdem die Unit soweit gebaut ist, werfen wir einen Blick auf den Aufruf des Client sowie auf das erwähnte Dekorieren um die Methode `encryt` herum. Beim Aufruf werden gleich zwei Konstruktoren angeworfen, der eigentliche Dekorierer und die Referenz der Basisklasse zugleich, welche sich direkt als Parameter erzeugen lässt. Man spricht hier auch von Indirektion, eine Technik, die auch das Strategie-Pattern effektvoll einsetzt. Jeder weitere Aufruf wird dann an die Basisklasse mit oder ohne „Dekoration“ weitergeleitet.

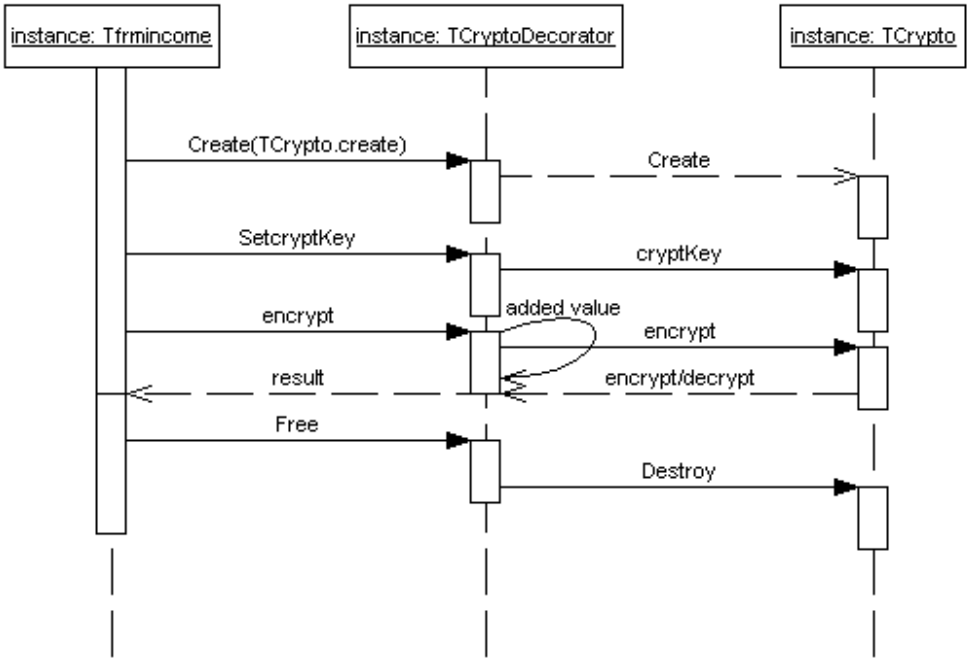
```
result:= Crypto.encrypt(cstring); //dispatch
```

Die Methode `encrypt` lässt sich nun erweiteren, in unserem Beispiel mit einer Validierung **vor** und dem Schreiben in ein File **nach** der Methode. Hier wird nun die mögliche Verschachtelung deutlich, die auch rekursiv möglich ist. Man kann hier einwenden, dies sei auch mit virtual/override möglich, aber das Nachteilige ist eben: einmal virtual, immer virtual.

Zum besseren Verständnis sie noch das Sequenzdiagramm beigefügt, welches das Weiterleiten mit Zusatznutzen des Decorator darstellt.

```
with TCryptoDecorator.create(TCrypto.create) do begin
  setCryptKey(ccrypt);
  edtGeheim.text:=encrypt(crstring);
  free;
end;

function TCryptoDecorator.encrypt(const cstring: string): string;
var cryptoF: TextFile;
begin
  if (length(cstring) > 255) then raise
    ECryptError.create('only shortstring possible');
  result:= Crypto.encrypt(cstring); //dispatch
  AssignFile(cryptoF, 'mycrypt.txt');
  Rewrite(cryptoF);
  writeln(cryptoF,result);
  CloseFile(cryptoF);
end;
```



Das Weiterleiten von der Unter- zur Oberklasse

Beim Kryptoalgorithmus handelt es sich um einen simplen symmetrischen XOR-Schlüssel, der wie eine normale binäre Addition pro Char funktioniert. In diesem Bereich gibt es andere Kaliber, zu finden unter: `www.torry.ru\security` oder `Delphi Super Page`. XOR hat die nette Eigenschaft, umkehrbar zu sein, d.h. encrypt (Verschlüsseln) und decrypt (Entschlüsseln) bedeuten dieselbe Funktion.

Beispiele für symmetrische Verfahren sind DES und IDEA, während RSA das bekannteste (und erste) asymmetrische Verfahren ist. Interessant seien abschliessend noch ein paar Gedanken zur ewig umstrittenen Schlüssellänge erwähnt. Der Aufwand für eine Brute-Force-Attacke ist einfach zu berechnen. Bei einer Schlüssellänge von 8 Bit ergeben sich 2 hoch 8 (256) Möglichkeiten. Den Schlüssel zu finden benötigt demnach 256 Versuche, wobei eine Chance von 50% besteht, den Schlüssel nach der Hälfte der Versuche gefunden zu haben. Bei einem 56 Bit Schlüssel rechnet ein Mainframe, welcher in der Sekunde 1 Million mal probiert, 2286 Jahre für das Finden des exakten Schlüssels. Bei 64 Bit, dauert die Suche 58500 Jahre und bei 128 Bit dauert das Hacken des Schlüssels mittels Brute-Force 10 hoch 25 Jahre.

```
function TCrypto.encrypt(const cstring: string): string;
var
  s: string[255];
  c: array[0..255] of Byte absolute s;
  i: Integer;
 begin
  s:=cstring;
    for i:= 1 to length(s) do c[i]:= c[i] XOR FcryptKey;
  result:= s;
end;
```

Max Kleiner