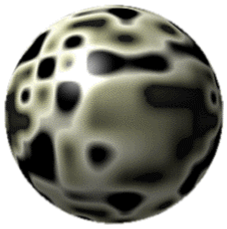
1 Design Patterns im Einsatz

Wenn das Klassendiagramm so zentral ist, sollten doch viele Projekte vor uns mit ähnlichen Problemen konfrontiert gewesen sein, bis ihre Lösungen dokumentiert wurden. Und genau das war die Motivation, bestimmte Klassen mit ihren Beziehungen als Lösungsmuster zu sammeln und **klassifiziert** zu veröffentlichen.[DP95][i]



Muster oder Patterns gehören zweifellos zu den nützlichsten und erfolgreichsten Konzepten des Software Engineering.
Dass sich Design Patterns mit UML visualisieren lassen, ist ja selbstverständlich [MK01][ii].

2 Motivation

„A Pattern is a proven Solution for a general Problem“

- Jedes Pattern beschreibt eigentlich ein Problem, das immer wieder vorkommt und sich dadurch wiederholt auflösen lässt.
- Pattern ist das Ergebnis, als auch die Regel selbst.
- Man muss **zuerst** das Problem erkennen, bevor man die Lösung einsetzen kann.

2.1.1 Wiederverwendung

- Wiederverwendung indem Softwareelemente oder Komponenten wie Bausteine für die Wiederverwendung entworfen werden.
- Komponenten selbst sind zu konkret, um z.B. zu einer vernünftigen Architektur zu gelangen.
- Wenn eine Lösung sich bewährt hat, ist eine Katalogisierung zum Auffinden der Patterns nötig.[iii]

2.1.2 Nutzen

Patterns werden von erfahrenen Entwicklern schon lange verwendet. Ihre Verwendung zur Beschreibung von Problemlösungs-Beziehungen bringen laut [DP95] folgende drei Vorteile:

Ein gemeinsames Design Vokabular

Patterns bieten durch ihren festgelegten Namen eine einfache Möglichkeit zur Kommunikation und Dokumentation. Ihr Name reicht aus, um über diverse Alternativen zu sprechen.

Dokumentations- und Lernhilfe

Beschreibt man ein System mit Design Patterns und den entsprechenden Diagrammen, lassen sich im Nachhinein die vorhandenen Funktionen verständlicher erklären.

> *Da Design Patterns für allgemeine Probleme, die immer wieder vorkommen, Lösungen anbieten, liegt es nahe sie für den gesamten Lehrbereich im Fach Softwarebau oder Engineering einzusetzen.*

Erweiterung zu bestehenden Techniken

- Design Patterns beinhalten Erfahrung von Experten.
- Design Patterns beschreiben, warum eine bestimmte Alternative, d.h. ein anderes Pattern, verwendet wird.
- Dadurch lässt sich der Übergang vom Use Case zum Klassendiagramm vereinfachen.

2.2 Katalog

Damit Patterns in einem definierten Sinne zur Verfügung stehen, braucht es einen Katalog oder ein Archiv.

2.2.1 Einteilung

Muster besitzen fünf fundamentale Bestandteile:

Musternname

Der Musternname benennt das Problem, die Lösung und die Konsequenzen in ein bis zwei Worten.

Problem

Das Problem beschreibt das Umfeld des Musters. Es wird zuerst der Kern des Problems allgemein beschrieben.

Kontext

Beschreibung der Situation, in der das Pattern sich einsetzen lässt.

Lösung

Die Lösung spezifiziert die Elemente, ihre Beziehungen und die Interaktion bei der Lösung des konkreten Problems.

Konsequenzen

Konsequenzen ergeben sich aus der Anwendung des Patterns. Die Konsequenzen benennen Vor- und Nachteile der Lösung.

Weitere Bestandteile

- Es sind dies die **Struktur** des Patterns
- Das **Laufzeitverhalten** mit den Komponenten
- Die **Implementation** mit Hinweisen auf mögliche Probleme

2.2.2 Pattern Kategorisierung

Creational Patterns beschäftigen sich mit der dynamischen Erzeugung und Wiederbelebung von Objekten unabhängig der Typen:

- Singleton
- Abstract Factory
- Builder

Structural Patterns beschreiben den statischen Zusammenhang von Objekten und Klassen, andere Klassen binden oder zu Strukturen führen:

- Decorator
- Wrapper
- Adapter
- Proxy

Behavioral Patterns charakterisieren das dynamische Verhalten von Objekten und Klassen:

- Observer
- Mediator
- Lock
- Strategy
- Template
- Visitor

Alternative Kategorisierung

- Einteilung nach ihrem Abstraktionsgrad ein. Man beginnt mit der höchsten Abstraktionsstufe der **Architektur** Patterns.
- Die geringste Abstraktion weisen sogenannte **Idioms** auf. Idioms gehören zu einer konkreten Programmiersprache.

2.3 Patterns in Delphi

Design Patterns sind auf objektorientierte Entwicklung zugeschnitten haben bewährte Implementierungen hinter sich [MK03][iv]:

- Structural Patterns – Design Time – Structure
- Creational Patterns – Load Time – Cycle
- Behavioral Patterns – Run Time – Function

2.3.1 Singleton

Stellen Sie sicher, dass eine Klasse mit genau einer Instanz existiert und ermöglichen Sie genau einen Zugriffspunkt auf dieses Kreieren der einzigen Instanz.

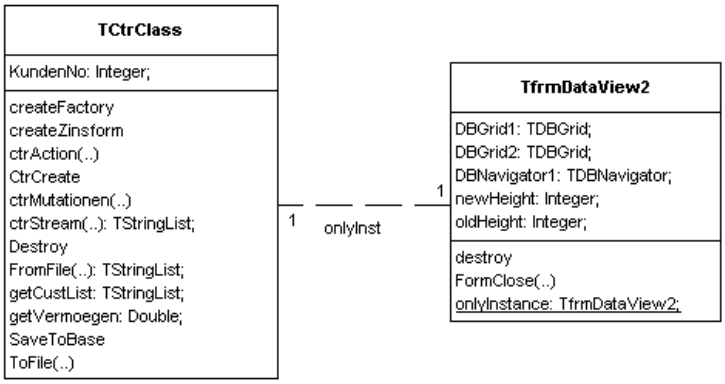


Abb. 6.1: Das Singleton mit einer Instanz

Die einzige Instanz wird über eine Klassenmethode erzeugt. Eine Klassenmethode ist eine Methode, die mit Klassen arbeitet, weil ja die Instanz zum Zeitpunkt des Aufrufes noch gar nicht existieren kann.

```
class function TfrmDataView2.onlyInstance: TfrmDataView2;
  begin
    if not assigned(FInstance) then
    FInstance:= TfrmdataView2.Create(application);
   result:=FInstance ;
  end;
```

Das Wechseln der Views über die Case Struktur im Controller garantiert auch das jeweilige Checken mit assigned, so dass wirklich nur eine Instanz existiert:

```
destructor TfrmDataView2.destroy;
  begin
    FInstance:= NIL;
    inherited destroy;
  end;
```

Sicherstellen das eine Applikation als Gesamtes nur einmal gestartet wird, genügt ein Mutex als Synchronisationsvariable:

```
var hMutex: THandle;
begin
  hMutex:= createMutex(nil, true, 'broker');
  if getLastError = ERROR_ALREADY_EXISTS then begin
     showmessage('broker already running');
     HALT;
  end;
```

2.3.2 Wrapper

Umhülle eine Gruppe von Funktionen mit einer objektorientierten Schnittstelle und erweitere somit eine andere Klasse mit der erzeugten Umhüllung.

Z.B. einen instanzierbaren OO-Zugriff auf den File Import/Export.

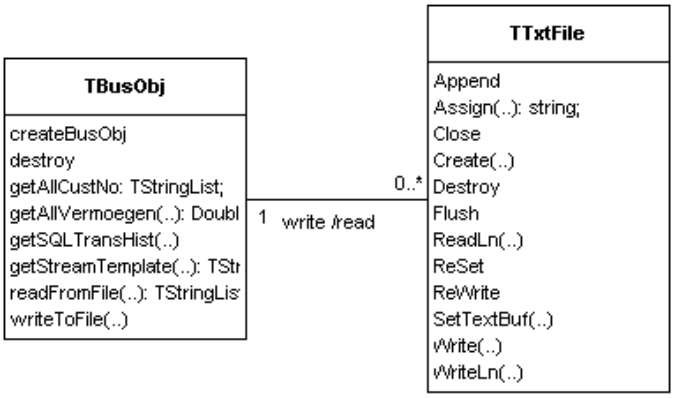


Abb. 6.2: Der Wrapper umhüllt den Export

Wrapper bieten dort einen Vorteil, wo lose Prozeduren und Funktionen durch eine Kapselung in Klassen übersichtlicher und einfacher im Aufruf werden:

```
begin
    q:= TTxtFile.Create(FILENAME);  //constructor
    q.ReWrite;
    if kList.count > 1 then begin
      for i:= 0 to kList.count -1 do
          q.Writeln(kList.strings[i]);
      q.Close; q.Free
    end;
 end;
```

Ein ähnliches Muster ist beim Erzeugen eines ActiveX vorhanden. ActiveX-Steuerelement-Experte erzeugt eine Implementierungs-Unit, die vom Objekt TActiveXControl und vom VCL-Objekt des Steuerelements, das eingekapselt werden soll, abstammt.

> *Der Begriff Wrapper wird in der angelsächsischen Literatur dem Adapter Pattern gleichgesetzt, was eigentlich nicht zutrifft. Der Adapter hat eine klare Aufgabe, ein Wrapper ist vom Kontext her vielseitiger.*

```
TTxtFile = class
  private
     FTextFile : TextFile;
     function GetEof : Boolean;
     procedure SetActive(state : Boolean);
     procedure SetMode(const NewMode : TFileMode);
  public
     constructor Create(Name : TFileName);
     destructor Destroy; override;
     procedure Append;
     procedure Assign(FName : string);
     procedure Close; virtual;
.........
```

2.3.3 Adapter

Passe die Schnittstelle einer Klasse an ein andere erwartete Schnittstelle an. Ein Adapter läßt Klassen zusammenarbeiten, die wegen inkompatibler Schnittstellen sonst nicht fähig wären.

Der Adapter dann zum Einsatz, wenn z.B. ein Form nicht von TObserver abstammen kann, aber kompatibel sein muss, weil es auf die Methoden der Schnittstelle angewiesen ist.

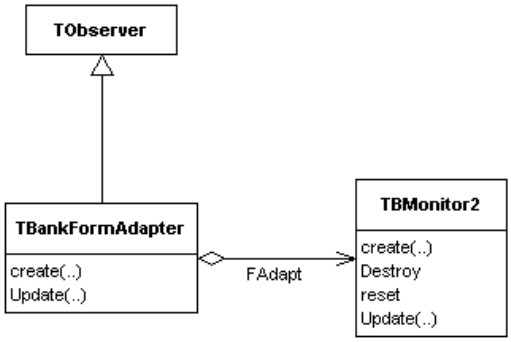


Abb. 6.3: Der Adapter vermeidet inkompatible Schnittstellen

- In den meisten Fällen sind Typen nicht kompatibel zueinander.
- Das Form klinkt sich in einen Adapter ein, d.h. es macht seinen Typ dem Adapter über den Konstruktor bekannt.
- Da der Adapter schon von TObserver abstammt, stammt nun indirekt auch das neue Form von TObserver ab:

```
TBankFormAdapter = class(TObserver)
  private
    FAdapt: TBMonitor1;
  public
    constructor create(aForm: TBMonitor1);
  end;
```

Die eigentliche Kopplung des Objektes an den Adapter geschieht im Konstruktor:

```
constructor TBankFormAdapter.create(aForm: TBMonitor1);
begin
  inherited Create;
  FAdapt:= aForm;
end;
```

2.3.4 Observer

> *Definiere eine Abhängigkeit von 1 : n zwischen Objekten, so dass bei einer Zustandsänderung alle registrierten Objekte benachrichtigt werden und sich dann automatisch aktualisieren können.*

Mit den Behaviorals gilt der Observer als Star und Liebling. Mechanismen sind raffiniert und besitzen hohen Stellenwert. Im Zusammenhang mit dem MVC Prinzip, ist das Observer bekannt.

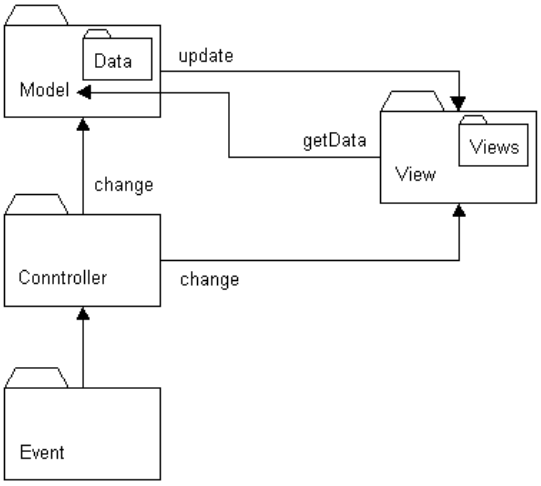


Abb. 6.4: Der Observer in vollem Einsatz

- Eine Klasse, die als Model die einzelnen Objekte die benachrichtigt werden wollen, registrieren kann.
- Jedes Objekt (View) lässt sich also von einer Basisklasse aus gesehen im Model einzeln registrieren:

```
TDispCenter = class(TObservable)
  private
    FObservers: TList;
    FAmount: double;
  public
    constructor Create;
    destructor Destroy; override;
    procedure Add(Observer: TObserver); override;
    procedure Remove(Observer: TObserver); override;
    procedure Notify; override;
    function getAmount: double;
```

```
    procedure changeAmount(value: double);
  end;
```

Subjekte werden am Anfang vom Form konstruiert und registriert, d.h. in eine Liste aufgenommen, die das Model (DispCenter) verwaltet.:

```
  obBars:= TProgBars.createBars(grpBox);
  obMemo:= TBMonitor.create(NIL);
  with ctr.dispCtr do begin   //registrieren
    add(obBars);
    add(obMemo);
  end;
```

Bei einer Zustandsänderung möchten die Objekte benachrichtigt werden um sich zu aktualisieren. Geschieht über die Methode changeAmount die wiederum mit Notify die Objekte einzeln benachrichtigt:

```
procedure TDispCenter.changeAmount(value: double);
begin
  if value <> 0  then begin
    FAmount := value;
    Notify;           // state change notification
  end;
```

Die Methode Notify ist nun angewiesen, dass alle Objekte die Methode Update besitzen, damit die Nachricht überhaupt ankommt:

```
procedure TDispCenter.Notify;
 begin
  for i := 0 to pred(FObservers.Count) do
```

```
    TObserver(FObservers.Items[i]).Update(Self);;
end;
```

- Jedes Objekt ist nun nach der Benachrichtigung selbst verantwortlich, die aktuellen Daten aus dem Model zu holen und je nach Kontext und Zeitpunkt sich zu aktualisieren.
- Jedem Objekt wird der self Parameter übergeben, der wie beim Strategie Muster nun die aktuelle Adresse des Model kennt um die Daten mit der Methode getAmount zu holen:

```
procedure TBMonitor.Update(dispCenter: TObservable);
begin
  with memMon.lines do begin
    add(intToStr(ctr.KundenNo));
    add(floaTtoStr((dispCenter as TDispCenter).getAmount));
    add(timeToStr(time)+' / '+ dateToStr(date));
  end;
end;
```

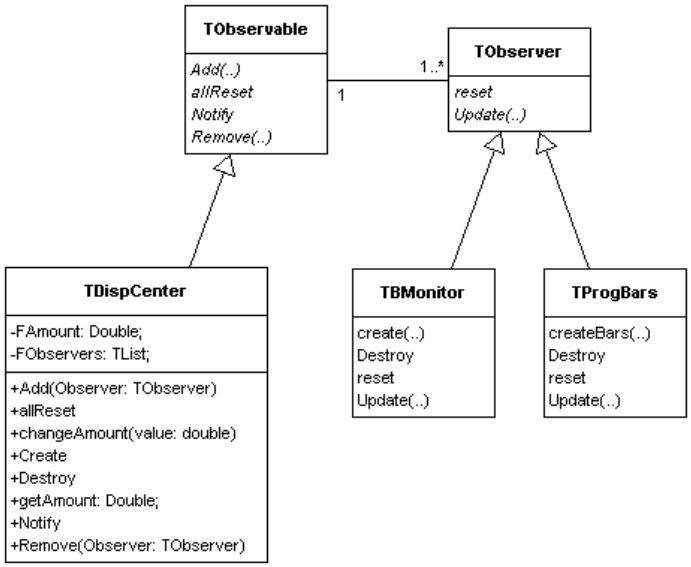


Abb. 6.5: Der Observer mit seinen Objekten

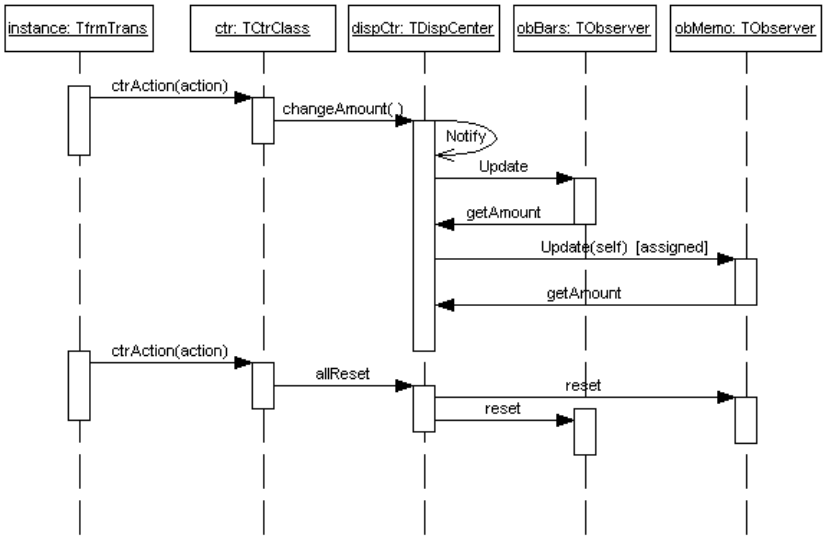


Abb. 6.6: Der Observer im Nachrichtendienst

2.3.5 Strategy

Bestimme eine Gruppe von Algorithmen, kapsle jeden einzelnen und ermögliche deren Austauschbarkeit während der Laufzeit. Das Strategiemuster ermöglicht den Algorithmus unabhängig vom Aufrufer zu variieren.

Das Strategiemuster legt den Fokus auf die Algorithmen und deren Aufgabe etwas zu berechnen. Ein Simulator, zyklische Berechnungen oder eine Schachanwendung leben von Strategien.

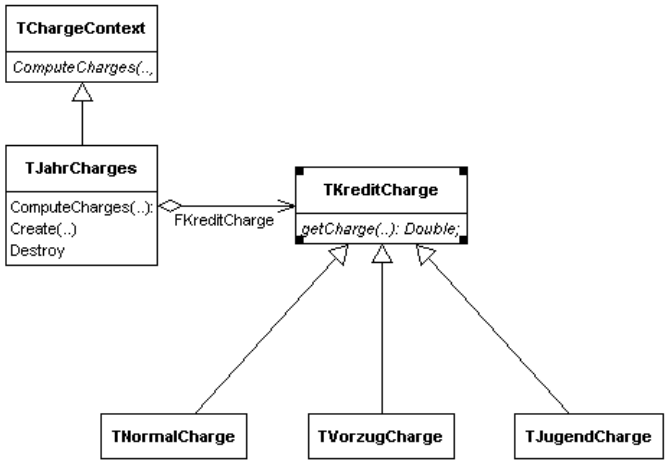


Abb. 6.7: Die Strategie hat 3 Algorithmen parat

Wenn man die gewünschte Kontogebühr wählt, lässt sich zur Laufzeit von der abstrakten Methode ComputeCharges die konkrete Methode mit der entsprechenden Strategie aufrufen:

```
51: begin
      TKreditkonto(konto).Jahresgebuehr:=
```

```
          FNormalChargs.ComputeCharges(StrToFloat(amount));
      end;
52:   begin
       TKreditkonto(konto).Jahresgebuehr:=
          FVorzugChargs.ComputeCharges(StrToFloat(amount));
      end;
53:   begin
       TKreditkonto(konto).Jahresgebuehr:=
          FJugendChargs.ComputeCharges(StrToFloat(amount));
      end;
```

Der Aufruf lässt sich dann dem Kontext übergeben, der durch den bekannten abstrakten Typ die konkrete Instanz in FKreditCharge weitergeben kann. Das Strategiemuster verdeutlicht mit der Übergabe der **Objektreferenz**, die maximale Flexibilität mit dem self Parameter.

Interfaces, Virtuelle Methoden und self-Pointer sind Techniken, die zum Standardrepertoire von Patterns gehören.

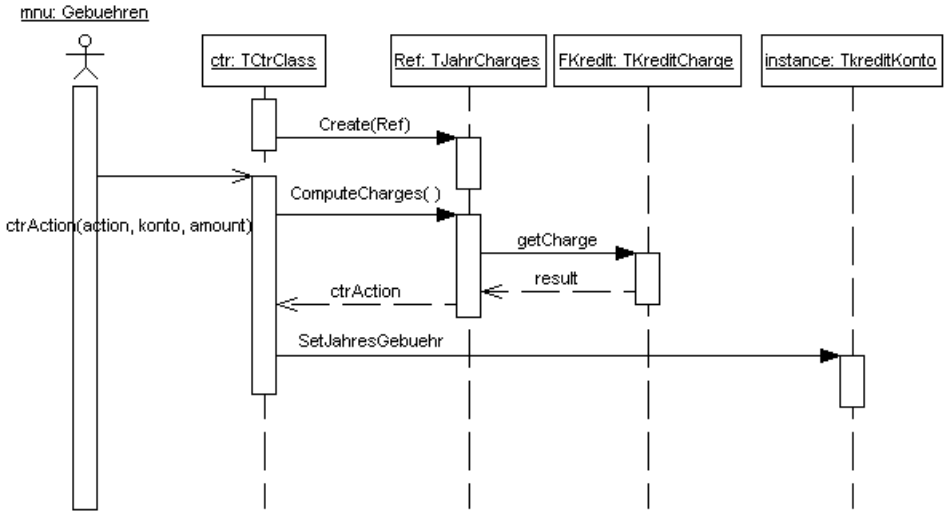


Abb. 6.8: Aus einer Strategie kann man wählen

2.3.6 Lock

> *Ermögliche einen Mechanismus, der kurzfristig einige Aspekte des Verhaltens eines Objektes deaktiviert. Blende kontrolliert eine Methode ein oder aus.*

Der Lock Mechanismus ist in seiner Implementierung einfach, hat aber weitreichende Konsequenzen auf das Zeitverhalten und birgt dann bei wirklich Echtzeit nahen Systemen so seine Geheimnisse.

Auch unter dem Namen ReferenceCounter im Einsatz.

Z.B.: Temporäres Deaktivieren von globalen Transaktionsdaten benötigt, damit die lokalen Transaktionsdaten in der Anzeige des Transaktionsmonitors nicht verloren gehen.

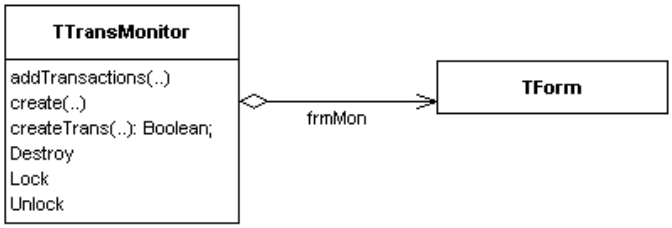


Abb. 6.9: Lock und Unlock in einer Klasse

Beim Eintreffen einer Buchung wird der globale Datenstrom unterbunden, da setLocking die Ereignisbehandlung auf NIL setzt:

```
procedure TTransMonitor.setLocking(signal: boolean);
begin
  if signal then application.onMessage:= NIL else
    application.onMessage:= showTransMessage;
end;
```

Lock selber kennt einen Zähler, der wie eine Semaphore den Zustand kontrolliert. Ein Lock-Manager implementiert auch Sperren und Semaphoren, um einen gleichzeitigen Zugriff zu verhindern.

Ein gleichzeitiger Zugriff auf gemeinsam genutzte Ressourcen kann u.a. zu Aktualisierungsverlusten oder Dateninkonsistenten führen.

```
procedure TTransMonitor.Lock;  //locked > 0 , unlock =0
begin
  inc(STLock);
  if STLock = 1 then setLocking(true);
end;
```

In der Zwischenzeit kann die Buchung im Transaktionsmonitor zur Anzeige erfolgen:

```
procedure TTransMonitor.addTransactions(amount: string);
begin
 STLock:=0; //unlock zum anfang
 lock;
 try
   with memMon.lines do begin
     add('Buchung: ' + amount);
     add(timeTostr(time)+' / '+ dateToStr(date));
   end;
 finally  // garantierte freigabe (deadlock verhindern)
   unlock
 end;
end;
```

Lock Patterns auch bei Ressourcenspendern. Ein Ressourcenspender verwaltet den nicht dauerhaften gemeinsam benutzten Status bezüglich der Komponente innerhalb eines Prozesses. (z.B. der MTS). Dies kann eine geöffnete Datei oder einen synchronen Port betreffen.

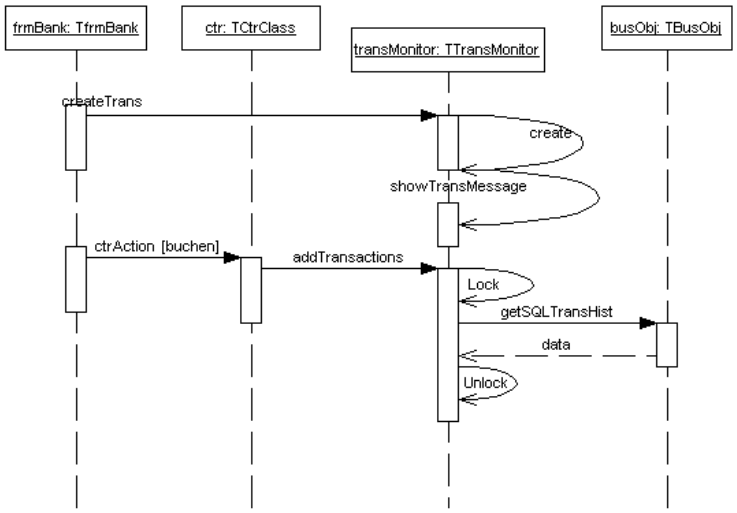


Abb. 6.10: Kontrollierte Synchronisation mit Lock

---

[i] [DP95] E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns, Elements of Reusable OO-Software, Addison-Wesley, 1995

[ii] Max Kleiner, UML mit Delphi, Frankfurt, 2000

[iii] R. Helm, R. Johnson, J. Vlissides, Auffinden von wiederverwendbaren Objekten, Addison-Wesley, Forth printing 1995

[iv] Kleiner, Delphi Design Patterns, Frankfurt 2003