Stefan Roock
Eckernförder Str. 274
24119 Kronshagen
Tel. 0431 / 54 59 031
Fax. 0431 / 54 59 032
Email: Roock@softOrg.de

Henning Wolf
Eichenweg 67
21493 Schwarzenbek
Tel. 04151 / 8 29 22
Fax. 04151 / 89 40 20
Email: Wolf@softOrg.de

Grenzen visueller komponentenbasierter Softwareentwicklung

Header

Delphi gilt als eine der innovativsten und ausgereiftesten visuellen Entwicklungsumgebungen für PC. Im Gegensatz zu vielen anderen 4GL-Entwicklungsumgebungen liegt Delphi eine objektorientierten Programmiersprache - eine Version von Object-Pascal - zugrunde.
Wir sind in der Praxis mit Delphi jedoch schnell auf Phänomene gestoßen, welche die Ausnutzung dieser objektorientierten Fähigkeiten behindern. Allzuoft standen wir als Entwickler vor der Alternative, die objektorientierten Eigenschaften der Sprache zu nutzen *oder* die von der Entwicklungsumgebung angebotenen Möglichkeiten. Im folgenden sollen einige dieser und weiterer Probleme aufgezeigt werden. Auch wenn sich der Artikel auf Delphi bezieht, treten viele der Probleme in ähnlicher Form auch in anderen visuellen oder komponentenorientierten Entwicklungsumgebungen (wie z.B. auch Visual-Basic, SQL-Windows, Optima++ und PowerBuilder) auf. Ein besonderes Augenmerk richten wir dabei auf das Ereignissystem, welches in den meisten Entwicklungsumgebungen und vielen Frameworks stiefmütterlich behandelt wird. Wir werden zeigen, daß dies zu gravierenden Problemen führen kann.

Einleitung

Delphi gehört zur Gattung der visuellen komponentenorientierten Entwicklungsumgebungen, wie z.B. auch Visual-Basic. Neben den visuellen Eigenschaften bietet Delphi Möglichkeiten für den eleganten Umgang mit relationalen Datenquellen. Die verwendete Programmiersprache ist eine Version von Object-Pascal. In dieser wurde das frühere Borland-Pascal (davor Turbo-Pascal) um weitere objektorientierte und metasprachliche Eigenschaften erweitert. Delphi bietet unter anderem:

- Einfachvererbung
- Garbage-Collection für Komponenten
- Dynamisches Binden und Polymorphie
- Differenziertes Geheimnisprinzip (public, protected, private)
- Properties als kontrolliert öffentliche Attribute

Ein Großteil der Entwicklung wird in Delphi visuell durchgeführt, indem Komponenten auf Formularen positioniert und miteinander verbunden werden (vgl. Abb.1). Dabei lassen sich zwei Arten von Komponenten unterscheiden:

- Komponenten, die während des Programmablaufes sichtbar sind, wie z.B. Eingabefelder, Knöpfe, Listboxen etc.Komponenten, die während des Programmablaufes unsichtbar sind, wie z.B. Datenquellen, die den Zugriff auf relationale Daten ermöglichen.

Eine einfache Benutzerverwaltung (wie sie in Abb. 1 dargestellt ist) läßt sich mit diesen Mitteln innerhalb weniger Minuten zusammenstellen. Man benötigt dafür lediglich vier Kompontenten:
1. Die unsichtbare Komponente Table wird durch Angabe eines Dateinamens mit der entsprechenden Datenbank-Datei verbunden.
2. Die unsichtbare Komponente DataSource wird mit der Komponente Table verbunden. Dies ist notwendig, weil sich sichtbare Komponenten nicht direkt mit Komponenten vom Typ Table verbinden lassen.
3. Die sichtbare Komponente DBGrid wird mit der Komponente DataSource verbunden und zeigt schon während der Entwicklungszeit den Inhalt der Datenbank-Datei an.
4. Die sichtbare Komponente DBNavigator wird mit der Komponente DataSource verbunden und dient der Navigation durch die Daten aus der Datenbank-Datei. Außerdem können über diese Komponente existierende Datensätze gelöscht oder neue Datensätze eingefügt werden.
Damit steht schon die Kernfunktionalität einer einfachen Benutzerverwaltung zur Verfügung. Jetzt müssen lediglich noch einige Kleinigkeiten hinzugefügt werden, wie z.B. Knöpfe zum Verlassen des Fensters und zum Aufrufen der Hilfe.
Nähere Informationen zur Entwicklung mit Delphi finden sich z.B. in [Cal95]**.**

Probleme

Während der Entwicklung mit Delphi mußten wir feststellen, daß mit zunehmender Programmkomplexität und fortschreitender Entwicklungszeit die Produktivität und die Softwarequalität immer weiter absank. Nach eingehender Analyse machen wir dafür in erster Linie die folgenden drei Phänomene von Delphi verantwortlich:
1. Delphi verhindert durch seine Struktur die Bildung von objektorientierter Kapselung, wodurch eine sinnvolle Strukturierung größerer Systeme erschwert wird.
2. Das Delphi-Ereignissystem erweist sich in größeren Projekten schnell als kontraproduktiv. Es ist schlecht dokumentiert und bricht die prozedurale Kapselung innerhalb einer Klasse auf.
3. Die Komponenten bieten oft nicht die erforderliche Qualität und Anpaßbarkeit.
Wir werden im folgenden diese drei Punkte detailierter ausführen.

Verhinderung der Kapselung

Die spezielle Eigenschaften von visuellen Entwicklungsumgebungen ist das visuelle Zusammenstellen von Komponenten. Oben wurde bereits kurz skizziert, wie einfach eine Benutzerverwaltung zusammengestellt werden kann. Wesentlich verantwortlich für diesen einfachen Umgang sind die folgenden Eigenschaften:

- Visuelle und nicht-visuelle Black-Box-Komponenten können zur Entwicklungszeit in der Entwicklungsumgebung manipuliert, konfiguriert und miteinander verbunden werden. So kann schon vor dem Compilieren der Effekt von Programmänderungen gut abgeschätzt werden.
- Es sind Komponenten für die üblicherweise benötigten Bereiche vorhanden: Benutzungsoberfläche, Manipulation und Präsentation transienter sowie persistenter Daten.
- Der Code für die Einbettung der Komponenten in das Anwendungsprogramm wird generiert. Der generierte Code kann durch den Entwickler manipuliert werden. Delphi kann generierten und manuell erstellten Code voneinander unterscheiden.

Leider verhindern gerade diese Eigenschaften eine Kapselung spezieller Eigenschaften und damit eine sinnvolle Strukturierung größerer Softwaresysteme, wie sie z.B. in Abbildung 2a

dargestellt ist. In Delphi können Komponenten nur auf Formularen plaziert werden und es können nur Komponenten des selben Formulars visuell miteinander verbunden werden. Daher müssen sich alle miteinander verbundenen Komponenten auf dem selben Formular und somit auch in der selben Schicht befinden (vgl. Abb.2b). Wenn schon so einfache Strukturierungen schwierig herzustellen sind, läßt sich der Aufwand für die Realisierung ausgefeilterer Strukturen (wie z.B. in [KGZ93] dargestellt) nur erahnen.
Ein inzwischen schon berühmtes Beispiel mag hier als Demonstrationsobjekt dienen (siehe Abb. 3). Es soll mehrere Sichten auf Daten geben, in unserem Beispiel sollen die Daten einer Tabelle einmal als Tabelle und einmal als Balkendiagramm dargestellt werden. Dazu benutzen die beiden Sichten dasselbe Datenobjekt und ändern ihre Darstellung sofort, wenn sich die Daten ändern. Eine Strukturierung des Systems wie in Abb.3 dargestellt ist insbesondere dann wichtig, wenn man später weitere Sichten hinzufügen möchte. In Delphi ist eine solche Strukturierung nicht ohne weiteres möglich, da im Normalfall die Daten (Komponenten Table, DataSource) auf demselben Formular liegen, wie die Sicht (Komponenten DBGrid, DBNavigator). Das Hinzufügen einer zweiten Sicht ist nicht trivial. Der einzige Ausweg besteht darin, in diesem Fall auf die visuelle Konstruktion in Teilen zu verzichten. Die Komponten für die Darstellung würden erst zur Laufzeit mit den Komponenten zum Datenzugriff verbunden werden. Um so vorgehen zu können, muß man entweder hellseherisch begabt sein und vorher wissen, wo mehrere Sichten benötigt werden oder man verzichtet grundsätzlich auf das visuelle Verknüpfen der Komponenten.
In diesem Zusammenhang fällt auf, daß visuelle Entwicklungsumgebungen ein Formular als kleinste Strukturierungseinheit implizieren. Gleichzeitig stellen Formulare auch die Hauptstrukturierungseinheit dar, was bei obigem Beispiel noch kein Problem darstellt, für komplexerer Dialoge aber sehr wohl zum Problem wird. Die Strukturierung anhand der Benutzerdialoge erinnert an die Strukturierung der meisten uns bekannten COBOL-Programme an Bildschirmmasken. Leider läßt sich auch das Erben von Formularen nicht besonders elegant handhaben, so daß Entwickler schnell wieder wie in den guten alten COBOL-Zeiten „Copy-Paste“-Vererbung anwenden und das kopierte Formular den aktuellen Bedürfnissen anpassen. Vor diesem Hintergrund scheinen die meisten komponentenbasiert entwickelten Programme in ihrer Strukturierung den Programmen ähnlich zu sein, die vor 20 Jahren mit Cobol erstellt wurden. Der Grund hierfür lag damals wie heute an den nicht vorhandenen oder nur mangelhaft unterstützten Strukturierungsmöglichkeiten.

Das Delphi-Ereignissystem

Benutzereingaben werden über Ereignisse an das Softwaresystem gemeldet, wie in Abb.4 dargestellt. Zusätzlich führen bestimmte programmseitige Aktionen zum Auslösen von Ereignissen (siehe Abb.5). (Insbesondere trifft dies auf Veränderungen an Datendateien zu.) Ereignisse werden also von Ereignisquellen ausgesandt, um abhängigen Programmteilen die Möglichkeit zu geben, auf diese zu reagieren. Wir sprechen in diesem Zusammenhang daher anstelle von *Ereignismechanismen* auch von *Reaktionsmechanismen* (siehe [RW96]). Reaktionsmechanismen sind zentral für die Realisierung sogenannter reaktiver Systeme (siehe dazu auch [KGZ93]): ein reaktives System zeichnet sich durch die flexible Reaktion auf Benutereingaben aus, die im Zeitalter von grafischen Benutzungsoberflächen von jedem modernen Programm erwartet wird. Im Gegensatz dazu stehen ablaufsteuernde Softwaresysteme, die alle möglichen Benutzereingaben bereits vorgedacht und alle zugehörigen Programmabläufe einprogrammiert haben.
Aber zurück zu unserem Delphi-Beispiel: Hat man z.B. programmseitig einen Wert neu berechnet, der jetzt in dem Datensatz gespeichert werden soll, so erhält man das selbe Ereignis als wenn der Benutzer diesen Wert von Hand geändert hätte (nämlich

UpdateRecord). Dadurch ist oftmals zusätzlicher Aufwand notwendig, um die Herkunft der Ereignisse zu unterscheiden.
Delphi-Ereignisse können auf zwei Arten aktiviert werden:
1. Synchron: Das Ereignis wird sofort bei seiner Aktivierung an die angemeldete Methode einer Komponente gemeldet.
2. Asynchron: Das Ereignis wird bei seiner Aktivierung in eine Queue gestellt und erst dann an die angemeldete Methode einer Komponente gemeldet, wenn Delphi die Kontrolle zurückbekommt.
Dabei kann man nur durch Codeinspektion (sofern man den Quellcode einer Komponente zur Verfügung hat) feststellen, wann Ereignisse ausgelöst werden und ob diese synchron oder asynchron zugestellt werden. Eine Dokumentation dafür ist ebenfalls nicht vorhanden.
Jedes Ereignissystem zerbricht zumindest ansatzweise die Kapsel einer Klasse sowie potentiell die Kapsel einer Methode (hier als prozedurale Abstraktion bezeichnet).
Unter prozeduraler Abstraktion versteht man das Erstellen von Prozeduren, um bestimmte Programmroutinen mehrfach nutzen zu können. Sie abstrahieren von ihrer konkreten Verwendung und können durch Parameter an spezielle Kontexte angepaßt werden. Diese Art der Abstraktion ist keineswegs neu: Die Möglichkeiten zu prozeduraler Abstraktion fanden sich schon in frühen Programmiersprachen und finden sich natürlich auch noch in modernen objektorientierten Programmiersprachen. In diesem Zusammenhang werden die entsprechenden Codeteile in der Regel in private Methoden gekapselt.
Die Klassenkapsel kann durch die Verwendung von Ereignissen aufgebrochen werden, da eine Veränderung des Aktivierungszeitpunktes eines Ereignisses Auswirkungen auf andere Klassen haben kann. Daher kann die Implementation einer solchen Klasse nicht ohne weiteres unter Beibehaltung der Schnittstelle geändert werden. Es ist daher notwendig, die Ereignisse einer Methode als Teil ihrer Schnittstelle und nicht etwa als Implementationsdetail zu begreifen. Aus diesem Grund sollten Ereignisse, die von Methoden aktiviert werden, bei den Methoden dokumentiert sein. Weiterhin sollten die Klassen, die Ereignisse aktivieren und solche, die auf Ereignisse reagieren mit besonderer Umsicht behandelt werden. Dabei ist darauf zu achten, daß
1. möglichst wenig Klassen Ereignisse aktivieren oder auf Ereignisse reagieren,
2. den Klassen sofort angesehen werden kann, ob sie Ereignisse aktivieren bzw. auf Ereignisse reagieren und
3. es ein durchgängiges Konzept für die Aktivierung von Ereignissen gibt, aus dem klar wird, wann prinzipiell Ereignisse aktiviert werden.
Der erste Punkt läßt sich durch eine geeignete Strukturierung der Klassen erreichen. Dabei können Schichten gebildet werden, von denen nur einige wenige Ereignisse aktivieren bzw. empfangen können.
Der zweite Punkt läßt sich auf vielfältige Art und Weise realisieren. Im Prinzip reicht hierfür eine durchgängige Dokumentation der Klassen aus. Alternativ kann erzwungen werden, daß Klassen, die Ereignisse aktivieren oder empfangen von bestimmten Klassen erben (siehe hierzu z.B. das Observer-Muster in [GHJ+95] oder das Event-Observer-Muster in [RW96]).
Der letzte Punkt ist schwieriger zu lösen und bedarf einiger konzeptioneller Überlegungen. In jedem Fall ist hier eine Trennung zwischen benutzer- und programmseitig ausgelösten Ereignissen sinnvoll:
- Programmseitig ausgelöste Ereignisse sollten immer dann von einer Komponente ausgelöst werden, wenn sich der *abstrakte Zustand* dieser Komponente geändert hat. Die möglichen abstrakten Zustände einer Klasse und die Übergänge zwischen ihnen können wie in [RW96] gezeigt durch State-Charts modelliert und dokumentiert werden.
Vom Benutzer ausgelöste Ereignisse werden immer dann aktiviert, wenn der Benutzer eine *relevante Aktion* durchführt. Was eine relevante Aktion darstellt kann von Anwendung zu

Anwendung sehr stark variieren. So muß bei einer Textverarbeitung üblicherweise auf jede Tastatureingabe reagiert werden, während beim Ausfüllen eines Formulars in einem Fensters die Werte in den Feldern erst bei Betätigen eines Buttons relevant sind. Wie wir in [RW96] bereits diskutiert haben, sollten für die Reaktion auf Benutzeraktionen nicht Ereignisse, sondern Kommandos (Muster Command, siehe [GHJ+95]) verwendet werden.
Ein Kernproblem bei der Verwendung von Ereignissystemen sind reaktive Änderungen (siehe dazu auch [RW96]). Unter einer reaktiven Änderung verstehen wir die Änderung eines Objektes A durch ein Objekt B als Reaktion auf ein von Objekt A an Objekt B gesendetes Ereignis. Abb.6 veranschaulicht diesen Sachverhalt anhand eines erweiteren Interaktionsdiagrammes. (Verbreiterte Balken bedeuten hier Objektrekursion: Während eine Methode eines Objektes noch in der Ausführung steckt, wird dieselbe oder eine andere Methode desselben Objektes aufgerufen.)
Solche reaktiven Änderungen können leicht zu einer endlosen Ereignis-Reaktions-Schleife führen, wie in Abb.7 dargestellt. Solche Endlosschleifen führen immer zu einem Programmabsturz und sind extrem schwer aufzuspüren, da sie über mehrere Stufen das gesamte System durchziehen können.
Ein zweites Problem, welches zwar in der Regel kein Programmabsturz zur Folge hat, sind *Schwingungen* innerhalb eines Systems. Unter *Schwingungen* wollen wir hier die Existenz von endlichen Ereignis-Reaktions-Schleifen verstehen.
Aufgrund der Struktur des Delphi-Ereignissystems treten endlose Ereignis-Reaktions-Schleifen und Schwingungen relativ häufig und spontan auf. So kann das „naive" Einstreuen der Zeile

```
aCheckBox.state := cbChecked;
```

ein gesamtes System „umkippen". Bei der o.g. Anweisung handelt es sich nämlich um eine Zuweisung an ein *Property*. Immer wenn ein Property eines Objektes gelesen und geschrieben wird, wird automatisch eine zugehörige Get- bzw. Set-Methode aufgerufen. Diese Set-Methode führt bei der obigen Anweisung zum Auslösen eines OnClick-Ereignisses in der Komponente CheckBox. Reagiert unser Objekt auf dieses Ereignis, kann eine Schwingung oder endlose Ereignis-Reaktions-Schleife entstehen.

Komponenten

Um effektiv mit einem komponentenbasierten Entwicklungssystem wie Delphi arbeiten zu können, müssen die verwendeten Komponenten besonderen Ansprüchen genügen. Sie müssen

- höchsten qualitativen Ansprüchen gehorchen sowie
- extrem flexibel in ihrer Verwendung sein und möglichst viele Möglichkeiten zur Adaption bieten.

Diese Punkte sind so herausragend wichtig, weil ein Eingriff in die verwendeten Komponenten in der Regel gar nicht oder nur mit erhöhtem Aufwand möglich ist. So ist es in der Regel nicht möglich, von einer Komponente zu erben, um deren Verhalten zu spezialisieren. Ebensowenig ist es möglich, die Implementation der Komponente zu korrigieren, wenn sich diese als fehlerhaft erwiesen hat.
Aufgrund unserer Erfahrungen können wir zu diesem Thema auf jeden Fall feststellen, daß Delphi und ähnliche komponentenbasierte Entwicklungsumgebungen die geforderten Eigenschaften nicht ausreichend unterstützen:

1. Fast alle Komponenten enthalten Fehler oder zumindest Unzulänglichkeiten.
2. Wie mächtig und flexibel die gewählten Komponenten auch sein mögen, es wird immer Kundenwünsche geben, die sie nicht erfüllen.

Der letzte Punkt darf dabei auf keinen Fall den Kunden bzw. späteren Benutzern zur Last gelegt werden. Wenn Sie Individualsoftware kaufen, haben diese unserer Meinung nach einen berechtigten Anspruch auf einen „Maßanzug". Dieser darf natürlich nicht zwicken und kneifen. Der Punkt ist auch deshalb so ärgerlich, weil die Objektorientierung ja gerade die Mittel bereitstellt, die wir benötigen, um einen Anzug von der Stange maßzuschneidern. Leider werden diese Möglichkeiten von den heute verfügbaren komponentenbasierten Entwicklungsumgebungen in der Regel beschnitten oder ihre Anwendung behindert.

Fazit

Auch, wenn sich in diesem Artikel vorwiegend Kritik an Delphi findet, so sei hier nochmals darauf hingewiesen,
1. daß Delphi nur stellvertretend für viele andere visuelle bzw. komponentenorietierte Entwicklungsumgebung angeführt wurde und
2. daß Delphi sich trotz der vorhandenen Probleme hervorragend für den Bau von Prototypen sowie die Realisierung *kleinerer* typischer Datenbankanwendungen eignet.

Es geht uns also nicht um die Kritisierung eines bestimmten Softwareproduktes. Im Gegenteil halten wir Delphi für eine der besten komponentenbasierten Entwicklungsumgebungen, die heute erhältlich sind. Vielmehr wollten wir deutlich machen, daß der vielgepriesene komponentenbasierte Ansatz auch seine Tücken hat und unserer Meinung nach nicht die Objektorientierung ersetzen kann. Anzustreben ist eine Kombination aus Komponenten und Objektorientierung, bei der der maximale Nutzen aus beiden Welten gezogen werden kann. Damit liegen wir mit unseren Erfahrungen und Einschätzungen der Komponententechnologie in wesentlichen Punkten auf einer Linie mit Ulrich W. Eisenecker [Eis95].

Wir richten unser Augenmerk diesbezüglich zur Zeit auf Entwicklungen rund um JavaBeans (das Java-Komponentenmodell), sehen aber noch nicht die nötige Reife der Konzepte und vor allem der verfügbaren Produkte, um kommerzielle Projekte mit JavaBeans durchzuführen.

Bei größeren Softwareprojekten sollte heute der Einsatz anderer rein objektorientierter Entwicklungsumgebungen in Erwägung gezogen werden, sofern eine objektorientierte Architektur angestrebt wird. Durch die Berücksichtigung der obigen Hinweise läßt Delphi sich zwar auch für größere Projekte einsetzen, es muß dann allerdings auf einige seiner Vorteile verzichtet werden. In den meisten Anwendungsgebieten wird dann allerdings nicht viel mehr übrig bleiben, als „nur" ein GUI-Builder. Außerdem sind vorher mitunter größere „Umbauarbeiten" notwendig. Vor diesem Hintergrund scheint uns der Einsatz einer standardisierten und auf mehr Plattformen verfügbaren Programmiersprache wie C++, Eiffel oder Java für große Projekte sinnvoller.

Literatur

[Cal95] Charles Calvert. *Delphi Unleashed*. Sams Publishing. 1995.

[Eis95] Ulrich W. Eisenecker. „*Der Weg zur flinken Software*" in iX. September 1995. Seite 164 bis 169.

[GHJ+95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[KGZ93] Klaus Kilberth, Guido Gryczan, Heinz Züllighoven. *Objektorientierte Anwendungsentwicklung*. Braunschweig/Wiesbaden. Vieweg. 1993.

[RW96] Stefan Roock, Henning Wolf. *Konzeption und Implementierung eines „Reaktionsmusters" für objektorientierte Softwaresysteme*. Studienarbeit am Arbeitsbereich Softwaretechnik des Fachbereiches Informatik an der Universität Hamburg. 1996.

Verfügbar als Postscript-Datei (1 MB, ca. 130 Seiten) unter der WWW-Adresse: http://swt-www.informatik.uni-hamburg.de/~1roock. Alternativ kann eine gedruckte Version gegen einen Unkostenbeitrag bei den Autoren bezogen werden. Bitte an roock@softorg.de wenden.

Abbildungen

Abb.1: Formular Benutzerverwaltung aus Haushalt einmal unter Delphi (Datei Abb1a.bmp) und einmal im ablaufenden Programm (Datei Abb1b.bmp). Delphi-Entwicklungsumgebung (Datei 1c.bmp).
Abb.2: Bild mit Schichten: Presentation-Layer, Application-Layer, Persistance-Layer (2a) und einmal Bild mit Delphi-Schichten (2b) mit Programmschicht, die auf Datendateien zugreift.
Abb.3: Drei Komponenten: Daten, Tabelle, Grafik (wie in Gamma et al.).
Abb.4: Das Auslösen eines Ereignisses durch den Benutzer und Meldung an das Programm.
Abb.5: Das Auslösen eines programmseitigen Ereignisses und Meldung an das Programm.
Abb.6: Reaktive Änderung
Abb.7: Endlosschleife durch reaktive Änderung

Abb.1a


Benutzerverwaltung
Benutzer
Kennwort
Schreiben erlaubt?
ADMIN
ADMIN
Ja
STEFAN
STEFAN
Ja
Schließen
Hilfe

Abb.1b

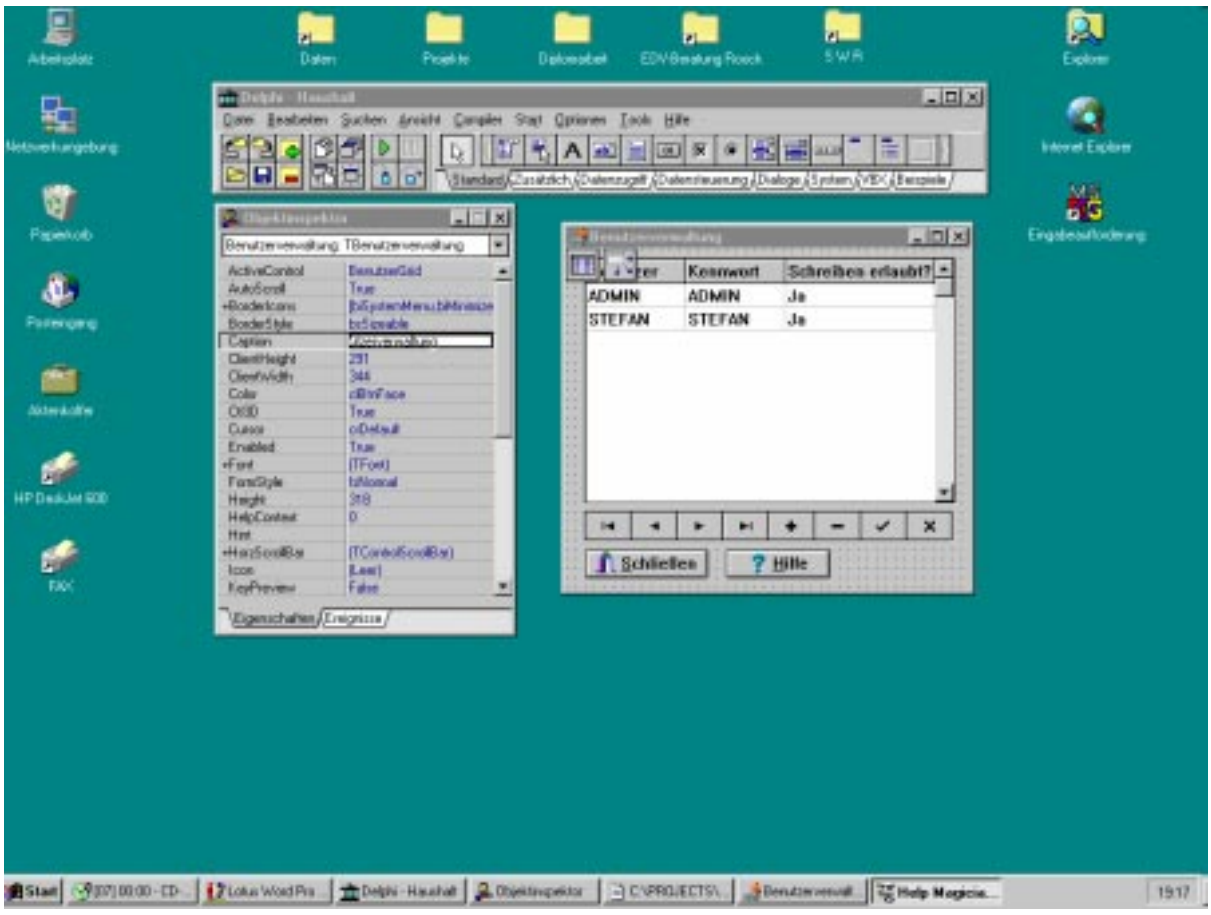

Abb.1c

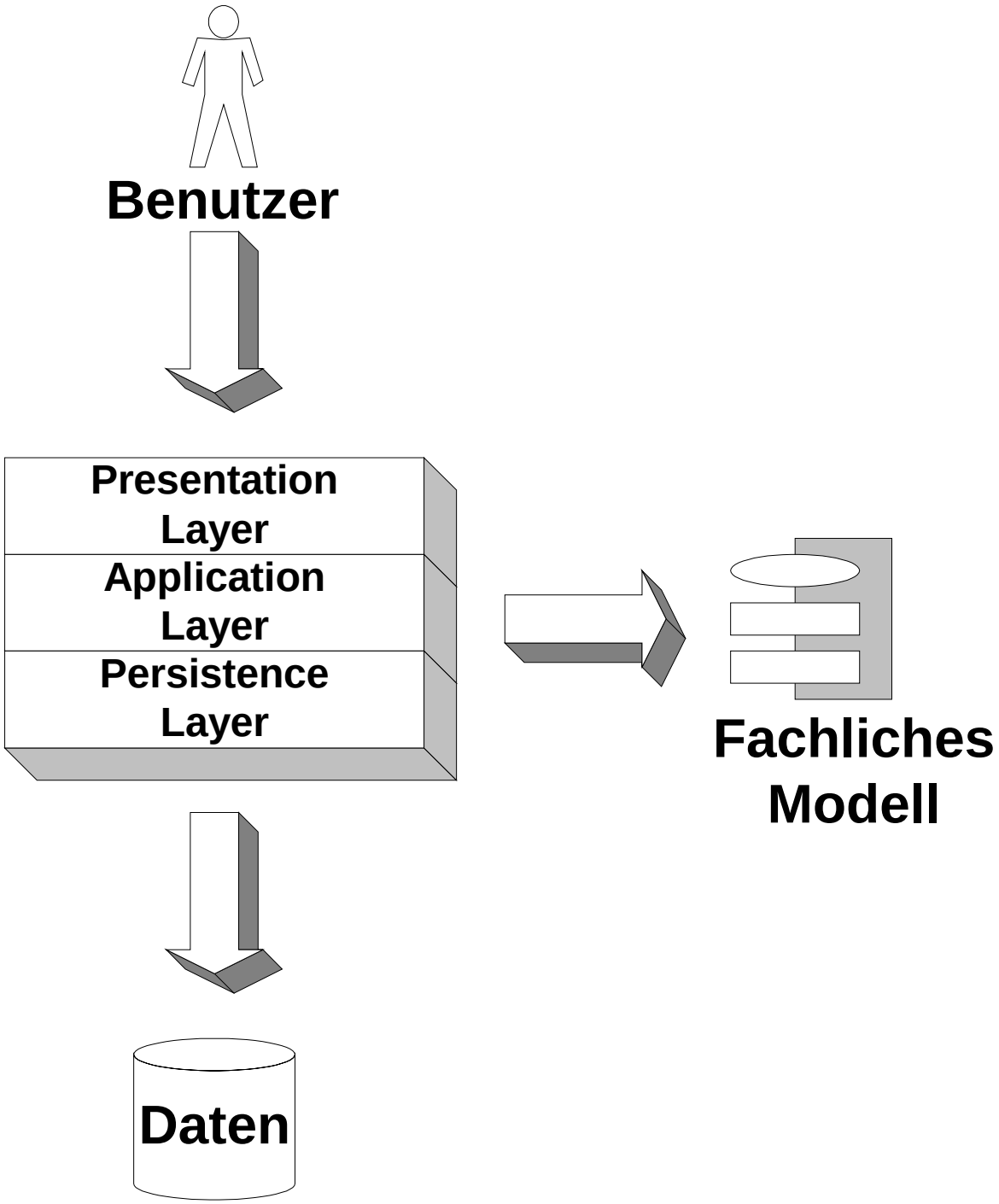

Abbildung 1

Abb.2a

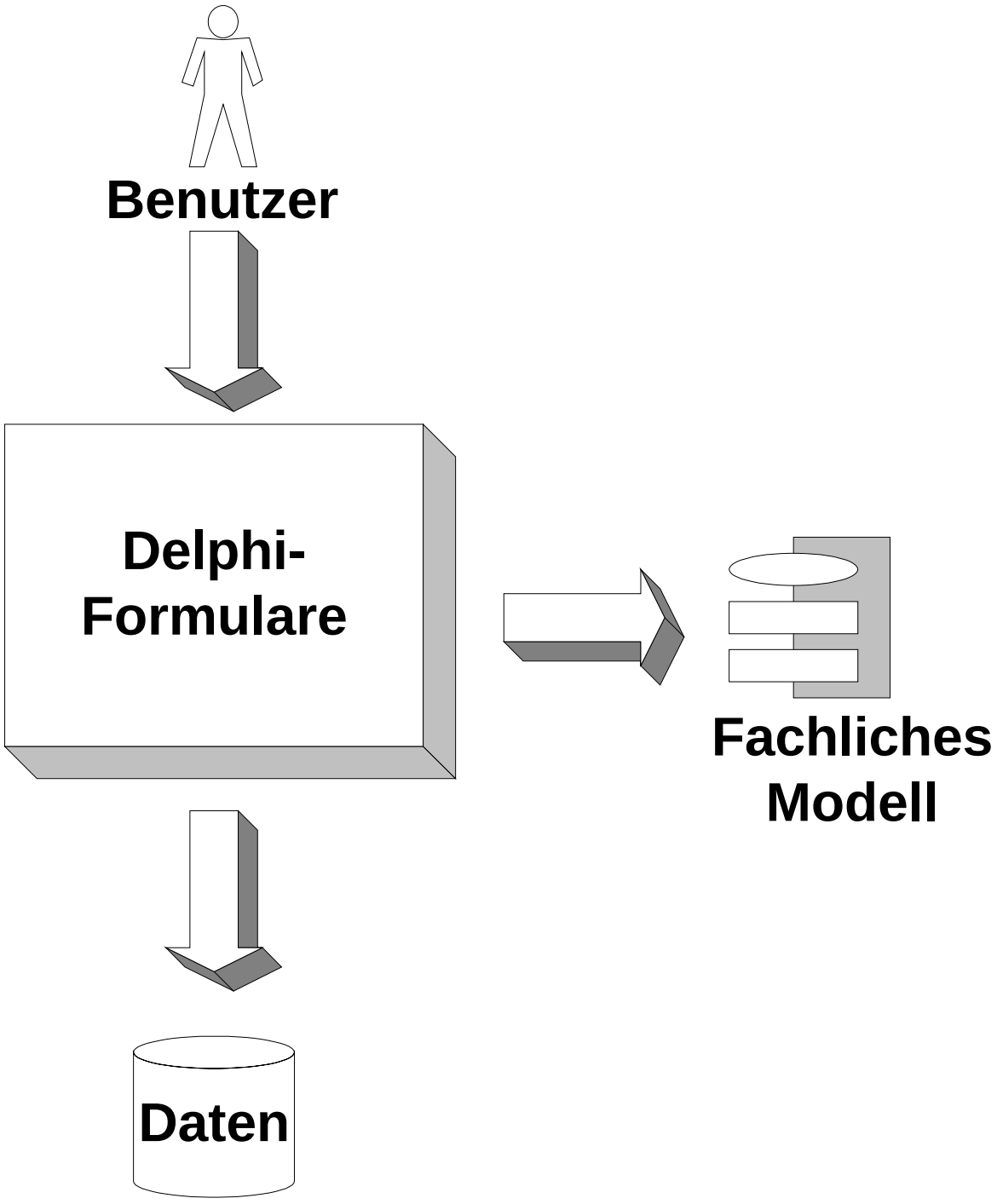

Abbildung 2

Abb.2b

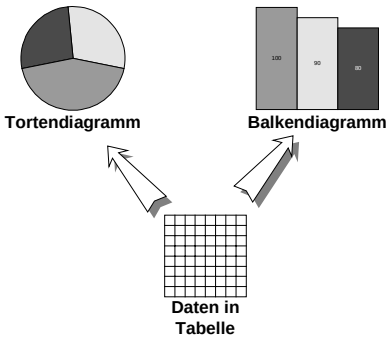

Abbildung 3

Abb.3

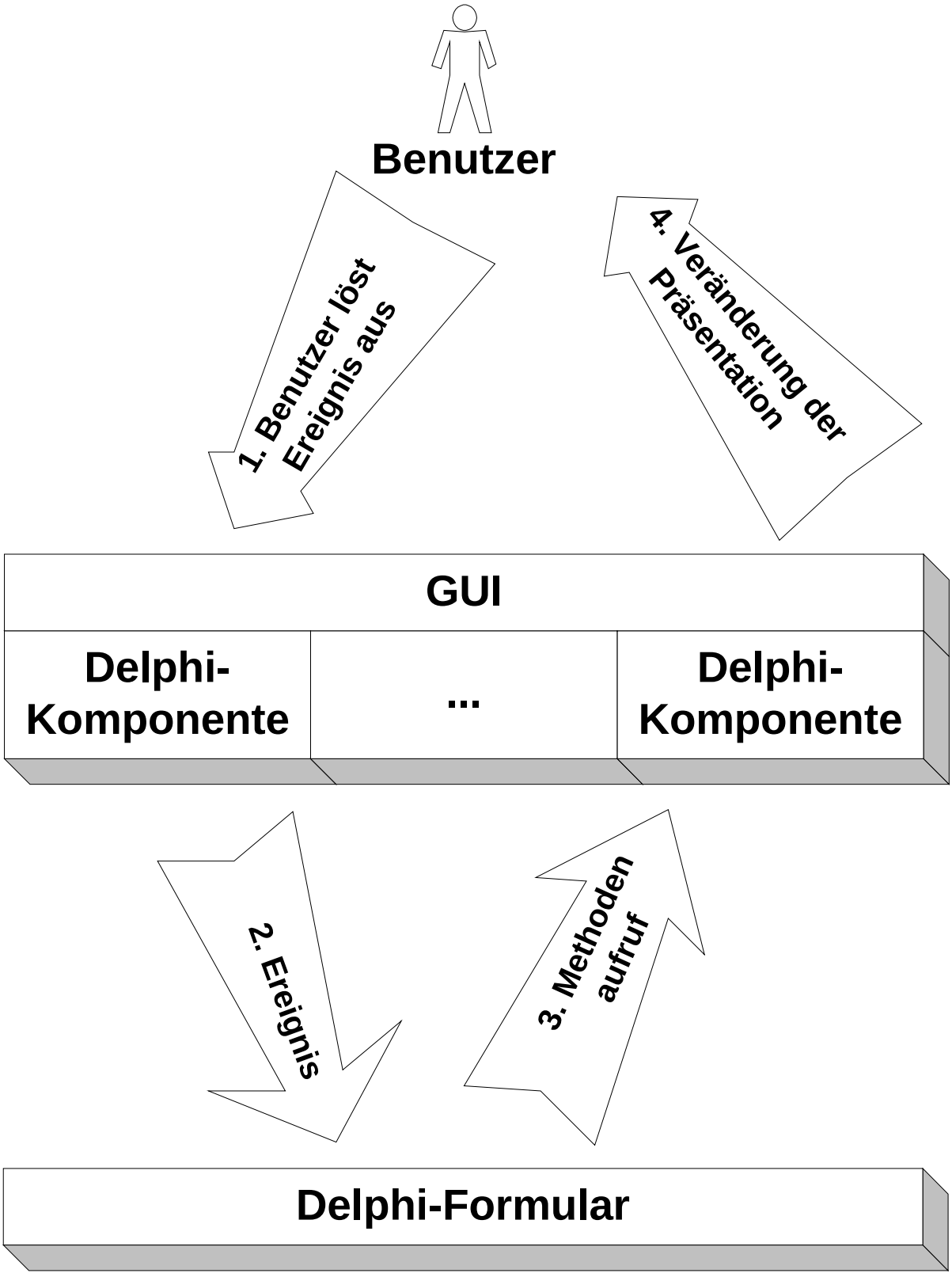

Abbildung 4

Abb.4

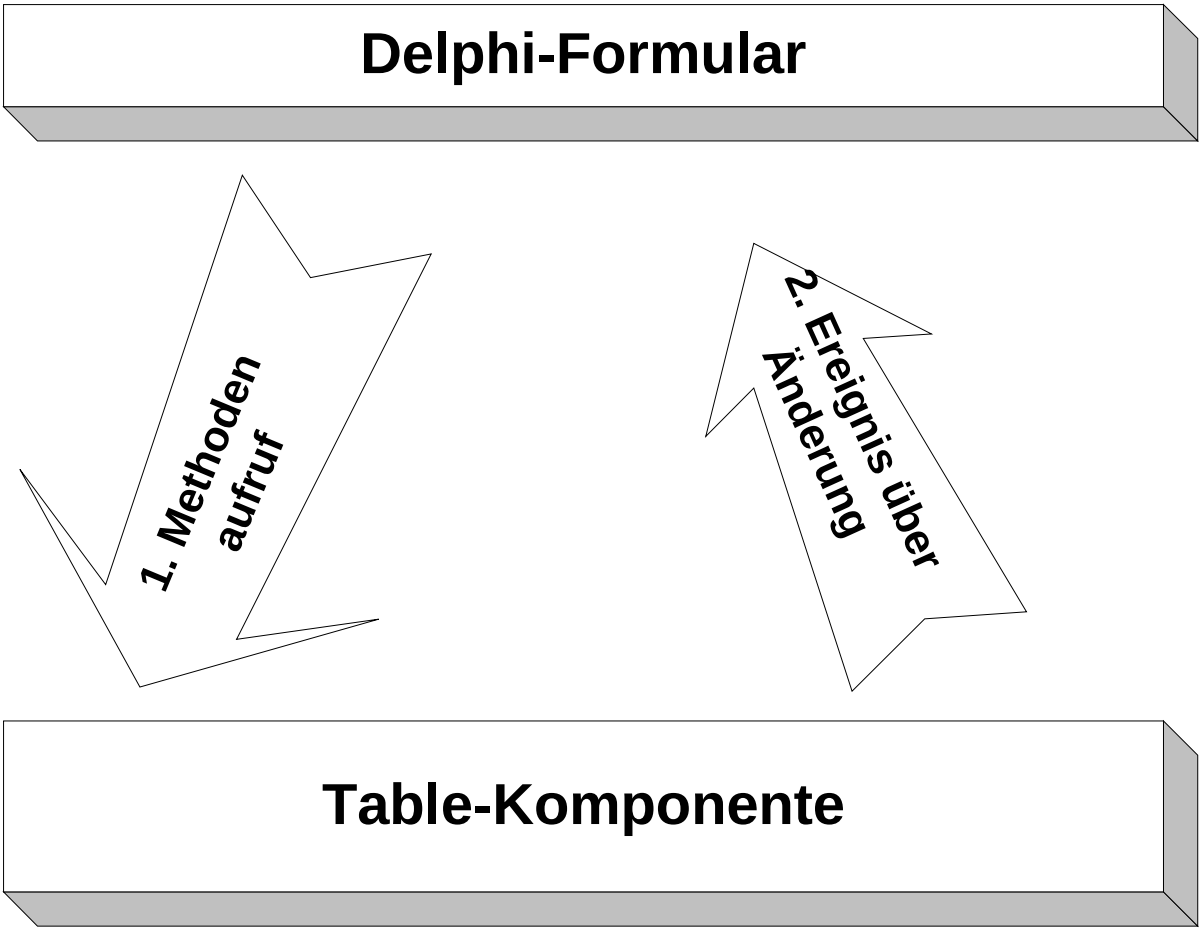

Abbildung 5

Abb.5

Der Subjektzustand wird durch einen Prozeduraufruf verändert.

Daraufhin wird Objekt A benachrichtigt, welches als Reaktion eine Veränderung an Objekt B vornimmt.

Objekt B verändert wiederum das Subjekt C. An dieser Stelle haben wir bereits eine *indirekte reaktive Änderung.*

Jetzt wird Objekt B von der ursprünglichen Änderung benachrichtigt.
Beim Sondieren findet B das Subjekt C nicht mehr in dem signalisierten Zustand vor.

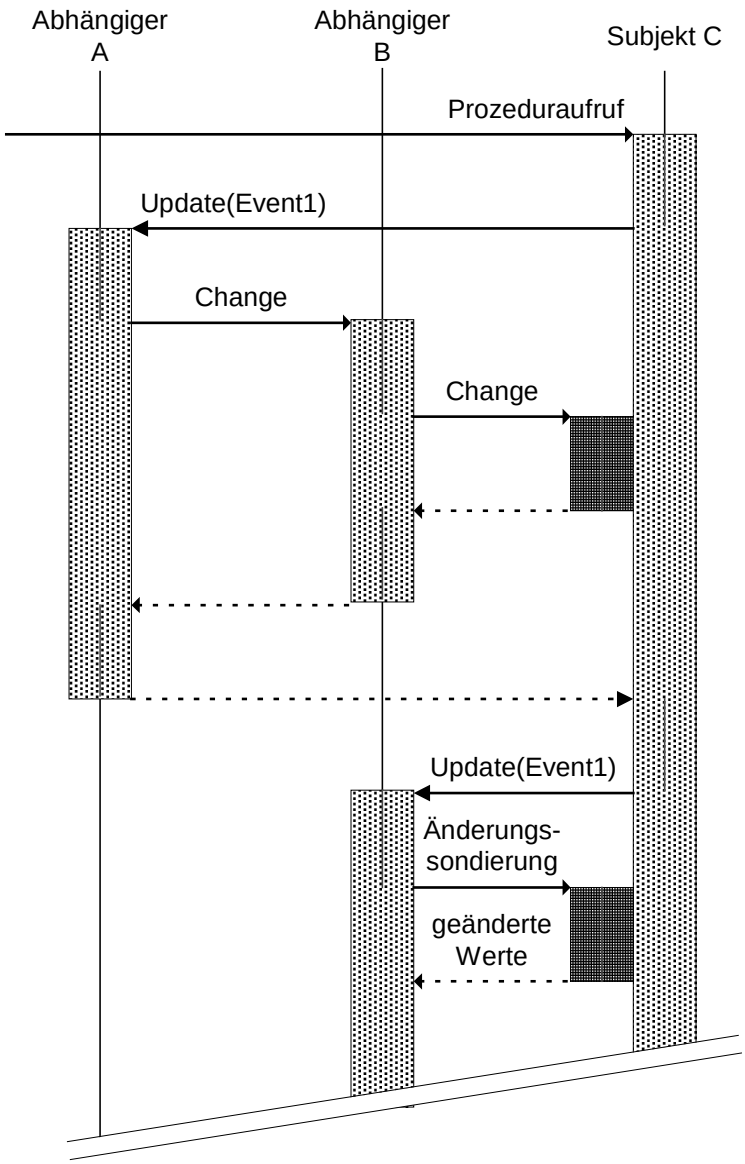

Abbildung 6

Abb.6

Der Subjektzustand wird durch einen Prozeduraufruf verändert.

Daraufhin wird Objekt A benachrichtigt, welches als Reaktion eine Veränderung an Objekt B vornimmt.

Objekt B verändert wiederum das Subjekt C. An dieser Stelle haben wir bereits eine *indirekte reaktive Änderung.*

Jetzt kann durch ständiges Change-Update eine Endlosschleife entstehen.

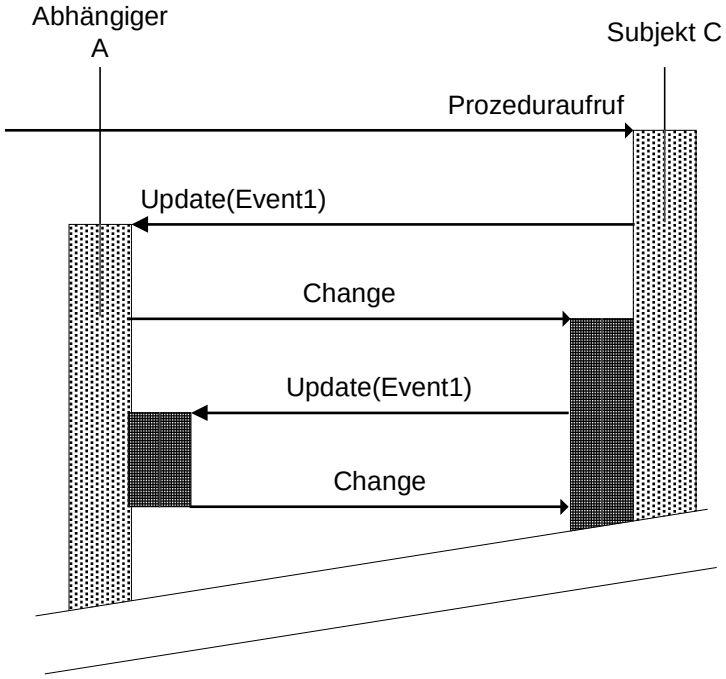

Abbildung 7

Abb.7