Software architectuur

Andy Verkeyn
Mei 1999 - Maart 2000 - April 2003

Derde versie

Andy.Verkeyn@rug.ac.be
http://allserv.rug.ac.be/~averkeyn

Object orientatie

Het OO (object oriented) paradigma ziet een applicatie als een verzameling van objecten die samenwerken. Een object is een abstractie van een entiteit of concept uit het domein van de applicatie. Elk object heeft een status (attributen) en een gedrag (methodes). Tijdens een applicatie communiceren de verschillende objecten door elkaar boodschappen te sturen.

Voorbeeld voor een bankapplicatie

> Basisobjecten zijn hier bv. een Klant en een Rekening. Elke klant heeft als eigenschappen een naam, een adres en een aantal rekeningen. Relevant gedrag van een klant voor een bankapplicatie is (dit zijn de methodes van klant) bv. het veranderen van zijn adres en het aanvragen van een nieuwe rekening. Een rekening heeft een bepaald nummer en een bedrag. Bewerkingen (methodes) zijn allerlei transacties zoals bv. overschrijven, stortingen,... alsook het opvragen van de huidige rekeningstand. Tijdens de werking van de applicatie zal een klant bv. naar één van zijn rekeningen de boodschap sturen om de stand op te vragen (en dus de methode uit te voeren).

Objecten met een gelijkaardige structuur (zelfde eigenschappen en gedrag) worden gegroepeerd tot een klasse. De beschrijving van een klasse vormt als het ware een blauwdruk voor de objecten van die klasse. Elk object is dus een instantie van een bepaalde klasse.

Voorbeeld

> Piet en Jan zijn twee objecten van de klasse Persoon. Ze beschikken over dezelfde attributen (naam, adres, leeftijd, echtgenote,...) maar hebben verschillende waarden voor deze attributen.

Als gebruiker van een object zijn we enkel geïnteresseerd in de manier waarop we met dit object kunnen werken, met andere woorden, in de methoden die we van buiten uit kunnen oproepen. Deze publieke methoden worden de interface van een object of de klasse genoemd. In OO is deze interface volledig gescheiden van de concrete implementatie. De interface vertelt ons 'WAT' een object kan doen, de implementatie vertelt de computer 'HOE' een bepaalde methode moet uitgevoerd worden.

Tot slot, biedt OO ons ook nog de mogelijkheid om relaties tussen klassen uit te drukken, zoals bijvoorbeeld de associatie-, aggregatie-, compositie en overervingsrelatie.

De basisfilosofie van OO is altijd: het verhogen van het abstractieniveau. We werken niet met kunstmatige datastructuren en losse functies, maar met “levende” objecten zoals die in de werkelijkheid voorkomen. Een bijkomend doel is het minimaliseren van het wateroppervlak-golf effect: wijzigingen of toevoegen in één stuk van een applicatie mag geen sneeuwbal effect van veranderingen in andere stukken code veroorzaken (wat de onderhoudbaarheid en uitbreidbaarheid van applicaties sterk doet toenemen).

Applicatie ontwerp

Inleiding

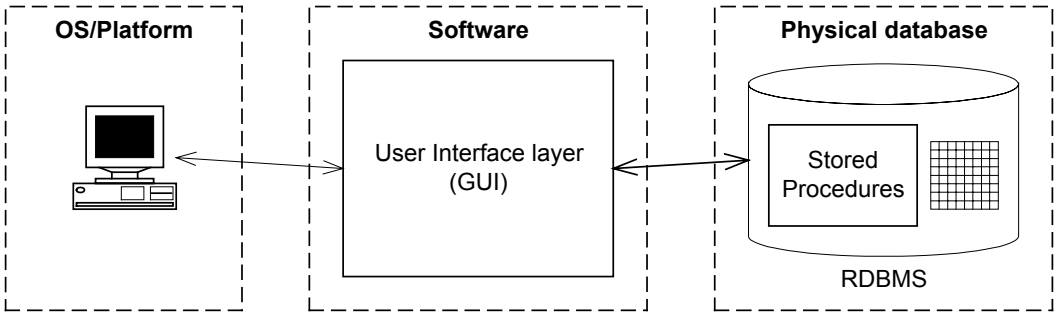
Een (transactionele) applicatie werkt meestal op dezelfde manier:

- we voeren als gebruiker gegevens in (via de zogenaamde user interface)
- de ingevoerde gegevens worden door het programma verwerkt (applicatie kernel)
- resultaten die tussen twee oproepen van een programma moeten bewaard blijven (persistente data), worden opgeslagen op een bepaald medium. Hiervoor kan bv. beroep gedaan worden op een relationele databank (RDBMS, Relational DataBase Management System).

Nu al kan opgemerkt worden dat er binnen de informatica wereld een grote verscheidenheid bestaat aan grafische user interfaces (de meest geavanceerde vorm van user interfaces) of GUI (Graphical User Interface), en data opslag technieken. Zo heeft elk besturingssysteem (Windows, Unix, OS/2, Mac,...) zijn eigen manier om een GUI op te bouwen en verschilt ook de toegangsmanier tot data van databank tot databank (zelfs binnen de relationele databanken). Hiervoor een adequate oplossing vinden is een basisprobleem binnen de software engineering.
Onder portabiliteit verstaat men de eenvoud waarmee men een bepaald programma kan herwerken voor een ander platform of databank. Zo kan het bijvoorbeeld nodig zijn een applicatie opnieuw te compileren met een andere instelling, of moet men bepaalde stukken code herschrijven voor het nieuwe platform. In het ergste geval moet men de applicatie volledig opnieuw maken. De ultieme vorm van portabiliteit is totale platformonafhankelijkheid, waarbij men aan het programma helemaal niets moet veranderen om het op een totaal ander platform te laten draaien.

Gezien deze opsplitsing naar functionaliteit is het logisch om de objecten van een applicatie ook in een aantal lagen (groepen) te gaan opsplitsen. Elke laag vormt hierbij een min of meer onafhankelijke, logische eenheid (separation of concerns) binnen het programma en is dus geen willekeurige groepering van objecten.
Een programma bestaat dan uit de opeenstapeling van de verschillende lagen, waarbij elke laag de functionaliteiten uit de onderliggende laag gebruikt en resultaten teruggeeft aan de laag erboven. Op die manier is elke laag een client van de onderliggende laag en een server voor de bovenliggende laag (client/server paradigma).

Two-tier



Figuur 1: Two-tier architectuur

Het twee lagen model (data driven approach) maakt enkel onderscheid tussen de user interface en methodes die de eigenlijke RDBMS aanspreken (zie Figuur 1). De user interface bevat dan meestal tekstveld-objecten (of andere user interface elementen) waar men SQL statements aanhangt. Deze statements beelden de inhoud van de GUI elementen rechtstreeks af op de kolommen van een relationele databank tabel. Verwerking van de ingevoerde gegevens gebeurt voor een deel in de user interface en voor een deel in stored procedures (en triggers) van de databank.
Het spreekt voor zich dat er bij de minste wijzigingen aan de UI of data organisatie grote stukken code moeten herschreven worden, wat tot hoge onderhoudskosten leidt. Dit zijn nu echter componenten die beide inherent vaak kunnen veranderen, onder andere voor het gebruikersgemak (van de UI) of om performantieredenen (van de databank). De portabiliteit van dergelijke applicaties is meestal nihil. Om deze redenen is deze aanpak dus zeker niet geschikt voor serieuze software projecten.
Deze twee lagen architectuur is het uitgangspunt van 4GL (Fourth Generation Language) en RAD (Rapid Application Development) omgevingen.

Three-tier

Inleiding

Een betere aanpak is het drie lagen model (business model approach) waarbij de business logica samengebracht wordt in een aparte tussenlaag. Dit model bestaat uit:

- User interface (functioneel model): De user interface laag concentreert zich op de interacties met de gebruiker en neemt vaak de vorm aan van een GUI (bv. een typische Windows applicatie).
- Business (service) logic (proces model): Deze laag bevat de kern van de applicatie, waarin de invoer van de gebruiker verwerkt wordt, en voert dus in feite de functionaliteit van de applicatie uit: de reden waarom dit programma geschreven werd. Dergelijke functionaliteiten worden ook business processen genoemd.
- Data storage (informatie model): Zorgt voor het bewaren (persistent maken) van gegevens. Dit kunnen de gegevens zijn die de gebruiker invoerde in de user interface, de resultaten van de berekeningen of combinaties van beide.

Dit model wordt typisch geimplementeerd in een 3GL (Third Generation Language) OO general-purpose programmeertaal zoals bv. C++ of Java.

User interface

Het drie lagen model ziet een user interface niet als een interface naar de databank maar als een middel om te gebruiker te laten communiceren met de eigenlijke kern van de applicatie.

Een applicatie kan immers ook over meerdere user interfaces beschikken, bv:

- GUI versie voor Windows
- GUI versie voor OS/2
- WWW versie
- Console (text georienteerde) versie
- ...

Door deze user interface elementen in een aparte laag te implementeren (en dus te scheiden van de eigenlijke applicatie logica), kan men een programma gemakkelijk porteren naar een ander platform dat meestal over een totaal andere manier beschikt om een dergelijke user interface op te bouwen (bv. Windows versus Unix). Een ander platform kan zelfs over totaal verschillende user interface componenten beschikken, hoewel de filosofie uiteraard telkens gelijklopend is. De enige laag die dan bij het porteren moet herschreven worden is deze user interface laag.

Er zijn zelfs frameworks beschikbaar die ook dit overbodig maken. Ze bieden een uniforme manier om een GUI aan te maken en mappen die zelf naar een specifiek platform. Een applicatie die gebruik maakt van, bv., het Zinc Application Framework (voor C(++)) kan door een eenvoudige hercompilatie geporteerd worden (de mapping gebeurt tijdens de compilatie fase).
Nog een eenvoudiger manier is het gebruik van het Java AWT framework dat zelfs deze hercompilatie overbodig maakt (de mapping gebeurt tijdens de run-time fase). Dit is mogelijk door de inherente platformonafhankelijkheid van de Java taal.
Men is gewaarschuwd dat dergelijke frameworks ook niet altijd zaligmakend zijn en soms problemen en onstabiliteiten vertonen. Bovendien bieden ze vaak een 'grootste gemene deler' van de bestaande platformen wat het ontwerp van een geavanceerde user interface kan beperken.

Business model

Het business model is het centrale en belangrijkste deel van een three-tier applicatie en blijft meestal stabiel omdat bedrijven hun manier van werken zelden radicaal veranderen (tenzij een bedrijf zijn werking volledig op zijn kop zet).

Een business model bestaat uit:

- business objecten: abstracties van de entiteiten en concepten uit het probleemdomein.
- business process: reflecteert de werking van het bedrijf

Een business model is volledig onafhankelijk van een bepaalde applicatie en staat ook volledig los van een bepaalde user interface en data storage. Het is een model van het probleemdomein.

Een applicatie is een implementatie van 1 of meerdere business processen. De realisatie van de business processsen gebeurt door het uitwisselen van boodschappen tussen de business objecten.

Business objecten kunnen meestal niet herbruikt worden voor verschillende applicaties (omdat de abstracties verschillen, al hebben ze dezelfde naam: een persoon abstractie voor een bank is niet gelijk aan een persoon abstractie voor een kapperszaak).
Herbruik kan eventueel wel in aanmerking komen binnen hetzelfde bedrijf omdat daar hetzelfde probleemdomein gemodeleerd wordt.
Meestal bestaat een business object echter uit kleinere componenten die wel kunnen herbruikt worden, zoals bv. een adres, telefoonnummer,...

Persistentie

De data persistentie laag is nog altijd een software laag die boven het data medium zelf staat, waarin de data effectief zullen bewaard worden. Het is voor de business objecten de toegangspoort tot persistentie. Het is deze data storage laag die met het gekozen data medium gaat communiceren.

Dit zorgt ervoor dat enkel en alleen deze laag moet gewijzigd worden wanneer voor een ander medium gekozen wordt. Op dit vlak zijn er een aantal mogelijkheden (van gemakkelijk naar moeilijk in een OO taal):

- geserialiseerde objecten (die eventueel via een netwerk verstuurd worden)
  Hierbij worden de objecten 'samengedrukt' tot een vlakke structuur die in een gewoon bestand wordt opgeslagen. Dit databestand kan dan eventueel via een netwerk verstuurd worden. Serialisatie is standaard aanwezig in Java.
- OODBMS
  Dit persistentie medium vereist de oproep van een store methode in de databank zelf.
- RDBMS
  Wanneer men objecten (met een potentieel complexe structuur) in een (vlakke) relationele databank wil bewaren, moet men zelf de attributen van de objecten naar de kolommen van een of meerdere tabellen mappen. De verschillen tussen het objectmodel en het relationele model noemt men de 'impedance mismatch'. Door de grote verscheidenheid van relationele databanken (onder andere de onderlinge verschillen van de SQL taal) is dit het moeilijkste medium.

Ook als men de layout van de databank wijzigt, moet men enkel deze laag aanpassen.

Wat relationele databanken betreft, zijn er bibliotheken beschikbaar die een uniforme API (Application Programming Interface) aanbieden en zelf de vertaling naar de onderliggende, specifieke databank uitvoeren die op dat moment ingesteld is. Voorbeelden van dergelijke bibliotheken zijn voor C(++) ODBC (Open DataBase Connectivity, een standaard ontwikkeld door Microsoft) en voor Java JDBC (Java DataBase Connectivity). Wanneer men in de data storage laag zo'n bibliotheek gebruikt, is de applicatie volledig onafhankelijk van de onderliggende (relationele) database (ook niet-relationele databronnen zoals bv. een tekstbestand en MS-Excel worden ondersteund).

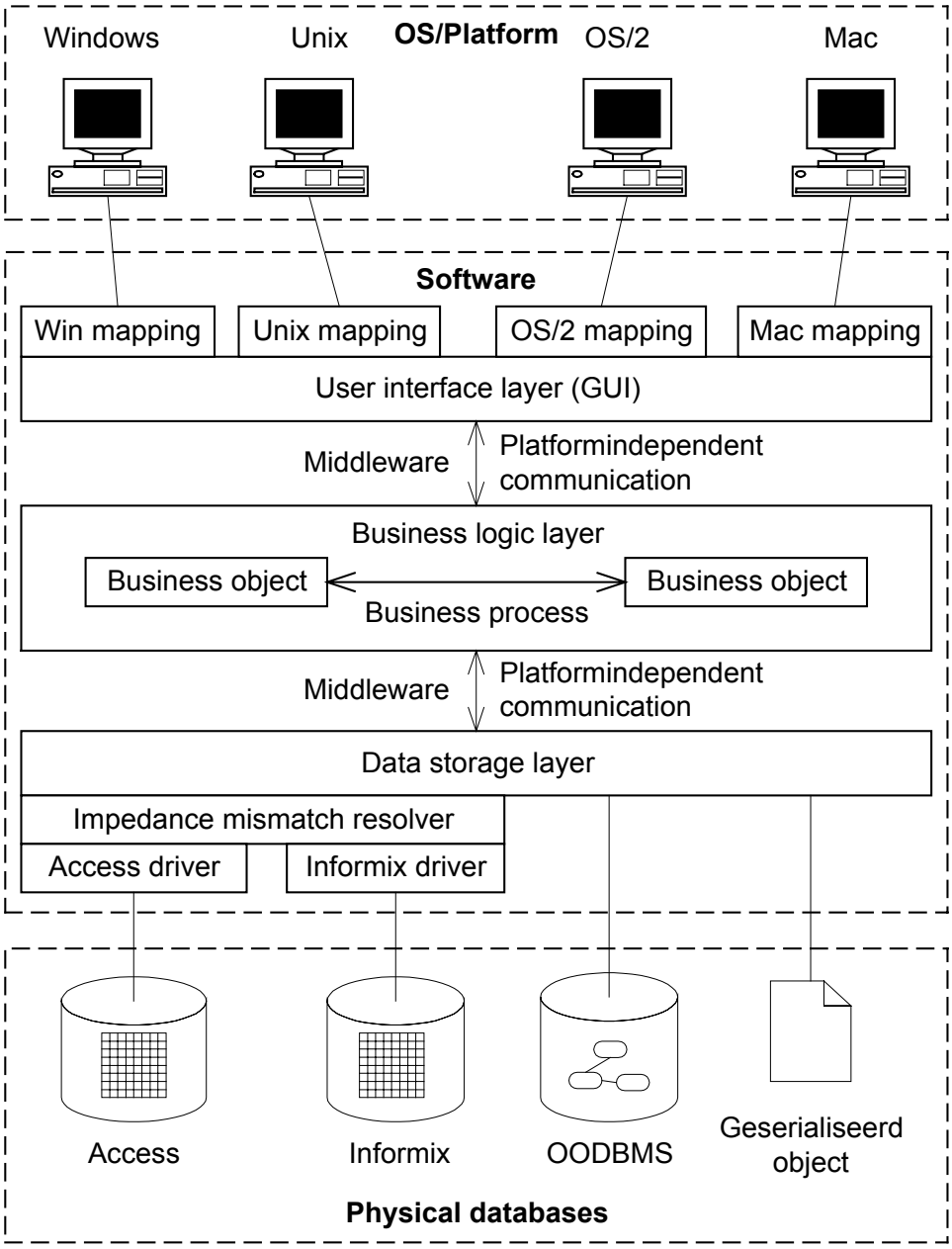
Interacties

De user interface vormt eigenlijk de motor van de applicatie en zal enerzijds informatie van het business model opvragen en anderzijds, indien de gebruiker erom vraagt, een dialoog tussen de business objecten onderling starten (een business process). De resultaten van het business process worden door de user interface aan de gebruiker getoond.

Indien nodig kan het model beslissen om zijn toestand (of bepaalde resultaten) persistent te maken. Hiervoor zal het model beroep doen op de diensten van de data storage. De vraag om zich persistent te maken, kan het model ook krijgen vanuit de user interface, bv. wanneer de applicatie afgesloten wordt.

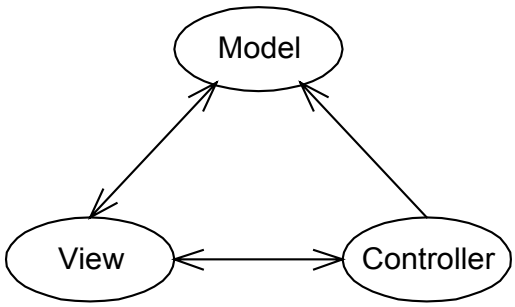
Het feit dat men op een logisch, conceptueel niveau deze lagen van elkaar scheidt, betekent echter niet dat ze niet in 1 uitvoerbaar ('executable') programma kunnen samenzitten. Langs de andere kant is het mogelijk om deze lagen werkelijk op fysisch verschillende computers te laten werken. De bibliotheek die voor de communicatie tussen de lagen over het netwerk zorgt, noemt men middleware. Platformonafhankelijke middleware moet de mogelijkheid bieden om -in elke laag-met verschillende netwerkprotocolen te werken. Sommige middleware laat zelfs toe de lagen in verschillende programmeertalen te schrijven. Voorbeelden van middleware bibliotheken zijn Java RMI (Remote Method Invocation), DCOM (Distributed Component Object Model, ontwikkeld door Microsoft) en CORBA (Common Object Request Broker Architecture, ontwikkeld door de OMG, Object Management Group, een standardisatie consortium van honderden bedrijven)

Het resulterend 3-tier schema is te zien in Figuur 2.

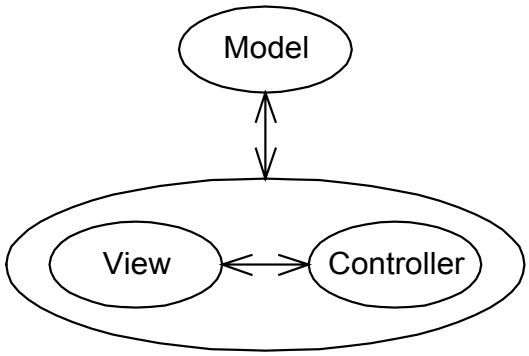


Figuur 2: Three-tier architectuur

User interface ontwerp



Figuur 3: Model-View-Controller



Figuur 4: Document-View

Model-View-Controller (MVC)

Inleiding

Een eerste paradigma voor het ontwerpen van user interface componenten zoals bv. een tekstveld, een knop, een lijst, een scrollbar, een combobox,.. werd bedacht door de ontwikkelaars van de Smalltalk programmeeromgeving (PARC Xerox, Palo Alto Research Center), tevens de pioniers op het gebied van GUI.

Hierbij wordt er, net zoals bij de business model approach, opnieuw onderscheid gemaakt tussen een invoercomponent (controller), de verwerking van de ingevoerde gegevens (het model) en één of meerdere uitvoercomponenten (views), maar dit keer gebeurt dit op een kleinere schaal (zie Figuur 3):

- model: beheert een of meer objecten en houdt de huidige toestand (status) bij. Een model bezit methodes om de toestand op te vragen (accessors, getters) en biedt mogelijkheden om die te veranderen (modifiers, setters).
- view: beheert een rechthoekige ruimte van het scherm (venster) en toont de data (status van het model) aan de gebruiker. Er kunnen meerdere views tegelijkertijd actief zijn (bv. een cijfertabel en een grafiek)
- controller: interpreteert de acties van de gebruiker op de view en wijzigt op basis van die acties (de status van) het model. De mogelijkheden van de controller kunnen dus door de view bepaald zijn.

Voorbeeld: scrollbar

- model: de minimumwaarde, maximumwaarde en huidige waarde.
- view: tekent de pijltjes (links en rechts) en de ‘thumb’ die aangeeft welk deel momenteel zichtbaar is
- controller: interpreteert muis kliks op de knoppen en het verslepen van de ‘thumb’ en verandert op basis daarvan de huidige positie in het model. Hier is echter ook informatie uit de view voor nodig, namelijk om het increment te bepalen (klein venster: groot increment, groot venster: klein increment).

Interacties

Het is de verantwoordelijkheid van het model om de actieve views op de hoogte te brengen als zijn toestand veranderd is, zonder ook maar één veronderstelling te maken

omtrent de aard van de view. Hij geeft enkel zijn toestand door aan een object dat daarin geïnteresseerd is. Deze communicatie gebeurt bij voorkeur met het event/listener model (het Observer design pattern): in het begin van de applicatie registreren view objecten (listeners) zich bij het model. Als het model wijzigt, stuurt het naar al deze listeners een event object dat de gewijzigde toestand bevat. Indien nodig, mogen de views ook een referentie naar het model waarvan ze listener zijn bevatten, bv. om meer gedetailleerde informatie omtrent de toestand van dit model op te vragen.

Uiteraard moet ook de controller een referentie naar het model bijhouden, om de door de gebruiker gevraagde bewerkingen op het model effectief te kunnen uitvoeren.

Het kan eveneens voorkomen dat wijzigigen in de toestand van het model andere acties op het model toelaten. Het is de verantwoordelijkheid van de view om de controller hiervan op de hoogte te brengen (dergelijke bijkomende manipulatie mogelijkheden kunnen immers afhankelijk zijn van de representatievorm). Opnieuw kan men voor deze communicatie het event/listener model gebruiken. Soms is het zelfs nodig om in de controller een referentie naar een view bij te houden, zodat de controller gedetailleerde informatie kan opvragen, bv. op welke plaats op de view de cursus zich bevindt.

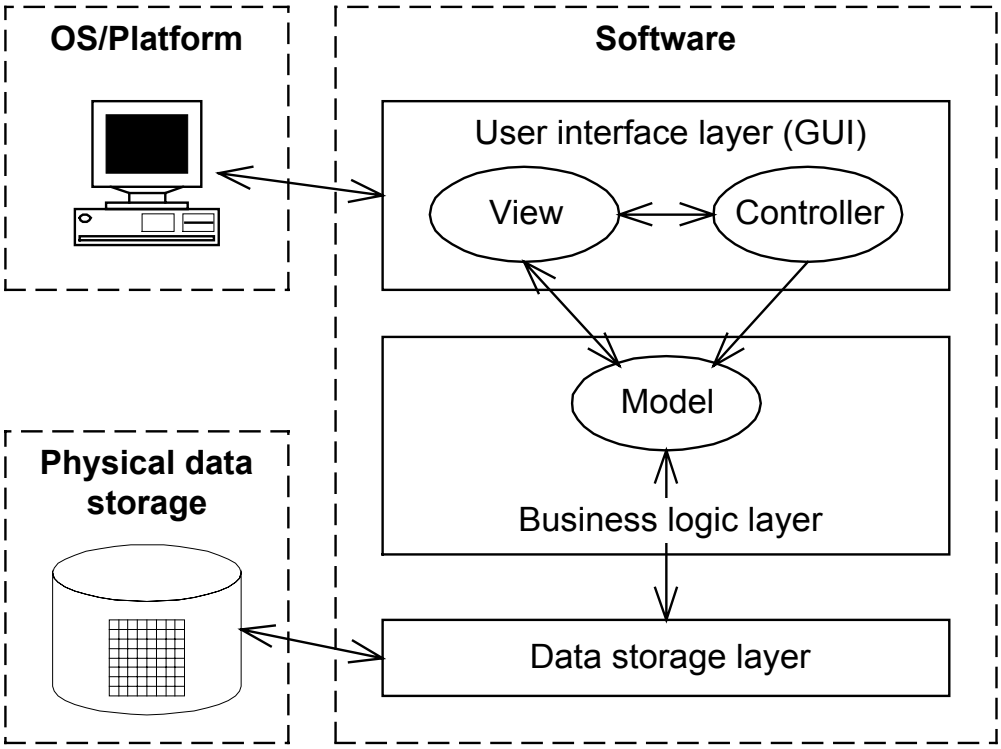
Document-View

Deze 3-delige architectuur is in de praktijk echter niet altijd gemakkelijk te verwezenlijken, omdat de view en de controller eigenlijk heel nauw samenhangen.

Tijdens het ontwerp van de Windows GUI bibliotheek MFC (Microsoft Foundation Classes), voerde Microsoft een 2-delige architectuur in waarbij men de view en de controller combineerden in één object. Dit noemt men de document-view architectuur of de modified MVC (zie Figuur 4).
De Java AWT/Swing GUI bibliotheek steunt ook op deze document-view architectuur, waarin men de view aanduidt als UI Delegate. Het is echter wel mogelijk om het model op te vragen en zelfs te wijzigen, zodat men eigenlijk de flexibiliteit van het MVC model behoudt.

Gecombineerde architectuur

De manier waarop de MVC architectuur binnen de business model approach past is getoond in Figuur 5.



Figuur 5: Gecombineerde architectuur

In de praktijk kunnen er zich echter nog wel onduidelijkheden voordoen om elementen binnen deze architectuur te doen passen. Een voorbeeldje...

Een tekstinvoer veld van een GUI is als GUI component eigenlijk veel te algemeen. Beter is een aantal standaardcomponenten gebruiken die de invoermogelijkheden meer beperken, bv. een numeriek invoer veld, een datum invoer veld,... rechtstreeks in het model van de GUI component (dit kan in de Java Swing bibliotheek met de setDocument methode). Wanneer deze (syntax beperkte) invoer dan aan het business model van de applicatie doorgegeven wordt, kunnen er daar nog bijkomende semantische controles gebeuren (bv. als de datum geldig is binnen de specifieke applicatie). Bemerk hierbij de twee niveaus van modellen: binnen de GUI component en binnen de globale applicatie (business logic laag).

Soms kan het ook nodig zijn om specifiek voor een bepaalde GUI component, bv. een JTable in Swing, een "model" te voorzien dat een deel van de business logica encapsuleert om bruikbaar te zijn voor deze component. (bv. AbstractTableModel). Dit stukje van het model is meestal afhankelijk van de gebruikte GUI library (bv. Swing) en kan bijgevolg onder de GUI-laag geclassificeerd worden.