Users Hebben Geen Username en Password!

Veel software engineers beweren object-georiënteerd te werken. Ze automatiseren bedrijfsprocessen voor klanten in een taal als Java of C# en menen hiermee voldoende grond te hebben voor deze bewering. Een nadere beschouwing van de implementaties die ze produceren leert echter vaak dat ze de basisconcepten van objectoriëntatie niet of nauwelijks toepassen. Daarnaast is het modelleren van de bedrijfsprocessen – tijdens of voor de bouwfase – een klassiek ondergeschoven kindje. Software architects, verantwoordelijk voor het ontwerp van de te bouwen systemen, blijven vaak op een te technische manier kijken naar het op te lossen probleem. Business architects ontplooien hun activiteiten daarentegen doorgaans op een te vaag of abstract niveau.

In dit artikel presenteren we onze visie op het belang van modelleren in software engineering trajecten. Hierbij kijken we naar verschillende basisconcepten die hierin een sleutelrol spelen, alsmede verschillende activiteiten die onder deze noemer ontplooid kunnen worden.

Modellering

Veel gebruikte definities voor model in de software engineering spreken vaak van een abstractie van de werkelijkheid die door versimpeling overzicht biedt op complexe systemen. In onze optiek is modellering echter meer een vertaling van de werkelijkheid naar een systeem. De abstractie is tot op zekere hoogte noodzakelijk, maar er zou niet in alle gevallen sprake moeten zijn van een versimpeling. Automatisering van een bepaald bedrijfsprobleem wordt vaak aangevlogen door van begin af aan een versimpelde voorstelling te maken van de werkelijkheid. Hierdoor valt er snel een gat tussen wat het bedrijf werkelijk doet en de software-representatie van het bedrijf in het systeem (van oudsher bestempeld als business-IT misalignment).

Het is dus van belang de bedrijfsspecifieke problematiek zo goed mogelijk in een model te vangen. Uiteraard is het mogelijk om bepaalde onduidelijkheden op wat hoger abstractieniveau te beschrijven, maar een volledig uitgemodelleerd systeem bevat in feite geen of nauwelijks versimpelingen.

Een volledig uitgemodelleerd systeem bevat in feite geen of nauwelijks versimpelingen

Concepten

We richten ons hier voornamelijk op het modelleren volgens een object georiënteerde stijl. Een aantal basisconcepten die in deze stijl een belangrijke rol spelen bespreken we hier kort.
In onze optiek is het belangrijkste kernconcept in deze stijl de combinatie van gedrag en statusinformatie die elk object representeert. Objecten worden opgedeeld in classes, en deze opdeling zou hoofdzakelijk gemaakt moeten worden naar verantwoordelijkheden voor bepaalde kernconcepten of actoren binnen een bedrijfsdomein. Dit heeft met techniek totaal niets te maken. Waar objectoriëntatie vaak ingezet wordt om technische vraagstukken op te lossen zien wij juist de kracht van dit paradigma in de automatisering van op het eerste gezicht complexe bedrijfsprocessen. Door juist de kernactoren in een organisatie te onderkennen en deze via duidelijk gedefinieerde grenzen (interfaces) te laten samenwerken om zo een bepaald probleem op te lossen zijn zeer efficiënte en compacte maar volledige modellen te realiseren.

Een belangrijke basisregel in object georiënteerd ontwerp is de zogenaamde active/passive rule

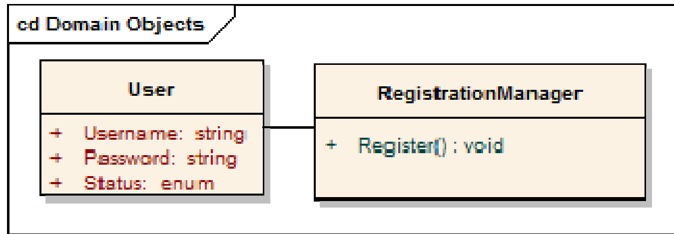
Een belangrijke basisregel in object georiënteerd ontwerp is de zogenaamde active/passive rule. Actieve objecten in de buitenwereld zijn passief in het model en andersom. Een andere manier van het omschrijven van deze basisregel is de encapsulatie: een object is verantwoordelijk voor zijn eigen interne status (state). Via publiek beschikbare methoden biedt elk object een interface aan de buitenwereld waarmee zijn interne status al dan niet gemanipuleerd kan worden.

Een tweede belangrijke basisregel in object orientatie is de chain of responsibility. Uitgangspunt bij het modelleren is dat objecten verantwoordelijkheden zoveel mogelijk delegeren aan andere objecten om zichzelf zo leeg mogelijk te houden. Een eigenschap van betere objectgeoriënteerde modellen is dan ook dat er een uniforme verdeling van logica over de verschillende classes bestaat. Door samenwerking met andere objecten – aangeduid als collaborations – lossen netwerken van objecten het gestelde probleem stap voor stap op.

Een derde concept dat we hier expliciet willen noemen is de door Dijkstra reeds in de jaren '60 geponeerde 'separation of concerns'. Dit concept hangt nauw samen met de active/passive rule, maar we stippen hem hier expliciet aan omdat het belangrijk is dat verantwoordelijkheden op een juiste manier gescheiden worden. Deze regel gaat op meerdere niveaus op – van objecten tot groepen van objecten.

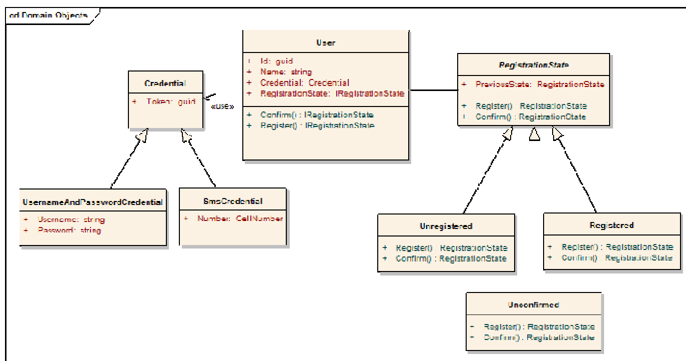
Om meer grip op bovenstaande concepten te krijgen zijn er verschillende bronnen van inspiratie beschikbaar. Een van de meest gebruikte bronnen zijn zogenaamde Design Patterns, geïntroduceerd in de software engineering door 'The gang of four' [GAM 95]. Dit werk biedt een overzicht van toepassingen van principes uit de objectoriëntatie, inclusief concrete implementaties. Inmiddels zijn er minimaal zoveel boeken over design patterns als dat er patterns zijn bedacht.

Toepassing
Hoe passen we deze concepten toe in een concreet voorbeeld? Laten we gewoon eens met een allerdaags voorbeeld beginnen. Veel architecten of software engineers hebben in het verleden wel eens een authenticatiemodule moeten uitwerken. Dit soort functionaliteit zou op meer meer procedureel gerichte manier als in onderstaande figuur geïmplementeerd kunnen worden.



Het userobject zou in dit geval een gebruikersnaam en wachtwoord voeren. Daarnaast zou de status (voor registratie/verificatie) in een statusveld beschikbaar zijn. Een vrij recht-toe-recht-aan benadering, echter wat gebeurt er als er nieuwe authenticatiemethodes ondersteund moeten worden? Denk bijvoorbeeld aan een vingerafdruk of irisscan.

Een tweede aspect dat minder gracieus OO ontwerp genoemd kan worden in dit model is de RegistrationManager class. Deze dirigeert het registratieproces van de gebruiker – dit staat haaks op de chain of responsibility en heeft meer weg van procedureel programmeren dan objectgeoriënteerd. Bij simpelere scenario's kan dit heel goed en snel werken, echter zodra er complexere processen gemaakt moeten worden neemt de vereiste effort exponentieel toe. Martin Fowler [FOW 03] refereert aan deze stijl ook wel als 'Transaction Script'.



Als we de concepten toepassen die we eerder genoemd hebben krijgen we een heel ander model voor onze authenticatiemodule, zoals in bovenstaand ontwerp is te zien.

Allereerst bevat de *User* class geen verwijzingen meer naar specifieke authenticatie-technische onderdelen als username of password. Gekozen is om een nieuwe class te introduceren die verantwoordelijk gemaakt wordt voor het authenticeren van gebruikers. De (abstracte) *Credential* class biedt een standaard interface voor diverse mogelijke implementaties van het authenticatiescenario. Op deze manier kan door subclassing van de *Credential* class gekozen worden voor een fingerprint authenticatie, username/pasword authenticatie of wat al niet meer. Dit is een mooi voorbeeld van hoe goed ontwerp flexibiliteit kan verhogen zonder alles configurabel te maken.

Een tweede interessant onderdeel is de structuur die gekozen is om een registratieproces af te handelen. Registratie is typisch een proces dat middels een aantal sequentiële stappen een aantal gegevens van een gebruiker verzameld via verschillende kanalen (email, sms, etc.). Na elke stap verandert de status van de gebruiker. Aangezien een dergelijk proces vaak verschillende eisen heeft (registratie voor een social networking site is vaak wat minder kritisch dan registatie bij een bank) zouden we graag een patroon introduceren dat ons in staat stelt het registratieproces elke keer anders te implementeren waarbij we wel met de standaard *User* class interface kunnen blijven werken. De gekozen structuur is die van een *State pattern* als beschreven in het standaard werk over design patterns [GAM 95]. Dit patroon stelt ons in staat om transities de we uitvoeren op een user te delegeren naar een state object met verschillende verschijningsvormen. Elke transitie (*Registreer()*, *Verifieer()*) heeft een nieuwe state tot gevolg waar deze nieuwe state zijn voorganger kent. Op deze manier een keten aan states laten beslissen wat de volgende stap in het proces zou moeten zijn.

Conclusie
We hebben in dit artikel kort gesproken over een aantal kernbegrippen binnen object georiënteerd ontwerp. Daarnaast is het belang van design patterns benadrukt en hoe deze standaard blokjes van functionaliteit kunnen bijdragen aan een beter ontwerp. Afsluitend hebben we laten zien hoe we een veel voorkomend probleem op een andere manier kunnen oplossen waardoor we winnen aan flexibiliteit. We hebben hiermee laten zien dat platte procedurele automatisering van bedrijfsprocessen tot moeilijker uitbreidbare en minder herbruikbare oplossing leidt en dat toepassing van object-georiënteerd ontwerp een extra dimensie aan modellen kan toevoegen.

Referenties

* [GAM 95] Gamma et al. *Design Patterns.* Wiley, 1995

* [FOW 03] Martin Fowler. *Patterns of Enterprise Application Architecture.* Addison-Wesley, 2003