.NET column

Ik geef al jaren trainingen over en met behulp van UML. De voordelen van deze notatietechniek ten opzichte van de 'oudere' technieken zijn al lang bekend: het is een standaard (beheerd door de Object Management Group), kent notatie voor objectoriëntatie-constructies en is behoorlijk consistent in het gebruik van bepaalde symbolen in diverse schema's. De cursussen die ik geef, beschrijven - variërend in detail - welke symbolen waar voor staan en leggen de relatie tussen de verschillende diagrammen uit.

Model

In de praktijk is goed te merken dat UML niet altijd wordt toegepast zoals je zou verwachten wanneer je de boeken leest. Ik heb altijd aangenomen dat dat lag aan ontbrekende kennis van UML en/of het gemis aan discipline bij het maken van documentatie. De eerste aanname was niet uit de lucht gegrepen; vaak heb ik gezien dat ontwerpers/programmeurs net weten wat een klassediagram is, maar zelden andere diagrammen kennen. Terwijl UML het klassediagram min of meer afleidt uit de overige diagrammen.

Mijn tweede aanname, dat ontwerpers/programmeurs niet gewend zijn aan het opschrijven van een ontwerp, is ook gebaseerd op de praktijk. Vaak wordt eerst gegrepen naar Visual Studio om alvast wat te coderen en volgt, in het beste geval, de documentatie pas later. Zelden wordt UML als een hulpmiddel voor het ontwerpen gezien en wordt het in het uiterste geval toegepast bij het opschrijven van wat geprogrammeerd is. Dat is jammer, want de software die we maken wordt steeds complexer en tekentechnieken kunnen ons helpen om op een hoger abstractieniveau na te denken over onze programmatuur. Programmeertalen bieden die mogelijkheid hooguit in beperkte mate.

De afgelopen week ben ik gaan inzien dat er nog meer aan de hand is. UML is te algemeen. Als ik een klassediagram krijg voorgeschoteld, heb ik aan het diagram alleen niet genoeg om te kunnen begrijpen wat er staat. Het probleem zit juist in de mogelijkheid die UML biedt om op verschillende abstractieniveaus te werken. Zo kan een klasse een afbeelding zijn van een C#-klasse, maar ook van een assembly of van een logische applicatielaag. Om te weten wat er precies bedoeld wordt moet er documentatie gemaakt en gelezen worden. Weg is het voordeel van het visueel modelleren, het diagram verliest zijn toegankelijkheid omdat we eerst weer moeten lezen. In veel gevallen wordt dit opgevangen door naamgeving. Vaak gaat dit goed, maar er zit een behoorlijk risico aan vast. Het niet definiëren van begrippen die in een ontwerp gebruikt worden kan leiden tot misverstanden en dus tot fouten. De interpretatie van het diagram is te vrij geworden.

Wat is dan een oplossing? Microsoft timmert met de DSL Toolkit aan de weg. Een DSL (Domain Specific Language) kan door een ontwerper/ontwikkelaar worden gedefinieerd en daarmee eenduidig een betekenis krijgen voor een specifiek domein. Het voordeel van een dergelijke specifieke definitie is dat er geen (of minder) misverstand kan ontstaan over de betekenis van het diagram. Doordat de betekenis zo precies is kan daarom ook met grotere zekerheid op basis van een met een DSL gemaakt diagram code gegenereerd worden.

Helaas blijft mijn eerste probleem met UML ook voor een DSL bestaan en zal misschien wel in hogere mate gelden voor een DSL: we zullen meerdere DSL's moeten maken en leren en dat is een hoop werk. De tijd die we winnen door het genereren van de code zouden we daar wel eens aan kunnen verliezen.

Erno de Weerd
Info Support. Over deze column kan verder gediscussieerd worden op http://blogs.infosupport.com/ernow