Ewart Nijburg en **Patrick Vorgers** zijn beiden technisch architect bij Ordina.

Refactoring werkt!

DE VOORDELEN VAN REFACTORING ALS ONDERDEEL VAN HET SOFTWAREONTWIKKELPROCES

Het was even wachten, maar met de introductie van Visual Studio 2005[1] kan ook de .NET-ontwikkelaar, out-of-the-box gebruik maken van refactorings. Refactoring is het proces van het verbeteren van de code tijdens het ontwikkelen. Maar hoe kunnen deze refactorings ons als ontwikkelaar nu helpen om kwalitatief betere code te schrijven?

Zonder dat we als ontwikkelaars er ons van bewust zijn, refactoren we op de een of andere manier al. Als we een methode een duidelijkere naam geven en alle aanroepen van deze methode aanpassen, zijn we aan het refactoren. Als we een lange methode opknippen in meer methodes die beter te begrijpen zijn, zijn we aan het refactoren. Eigenlijk is er dus niets nieuws onder de zon, want deze aanpassingen doen we als ontwikkelaar toch dagelijks? Ja en nee, want tijdens het refactoren ben je als ontwikkelaar bezig om op een *gestructureerde* manier de interne structuur van de programmatuur te verbeteren, zodat deze beter leesbaar, onderhoudbaar en herbruikbaar is zonder hierbij nieuwe functionaliteit te realiseren. Veel voorkomende aanpassingen zoals het aanpassen van een methodenaam zijn beschreven in refactorings. Een refactoring bestaat uit een aantal vooraf gedefinieerde stappen, waardoor het altijd mogelijk is een stap ongedaan te maken als tijdens het refactoren fouten zijn geïntroduceerd. Deze stapsgewijze controleerbare aanpassingen geeft ons als ontwikkelaar de mogelijkheid om met vertrouwen welke refactoring dan ook uit te voeren.

Refactoring wat en hoe

Refactoring is het proces dat de interne structuur van de code verbetert, zodanig dat het externe gedrag niet wordt veranderd.

Refactoring wordt vaak achterwege gelaten omdat de impact van de wijziging niet kan worden ingeschat. Dit is per definitie een verkeerde reden om niet te refactoren. Het is immers zo dat met behulp van refactorings, op een gestructureerde en gecontroleerde manier de interne structuur van de code wordt aangepast zonder het externe gedrag te veranderen. Refactoring heeft dus geen impact op het functioneren van de software. De praktijk leert echter dat dit veelal wel het geval is. De voornaamste redenen hiervoor zijn:

- Het stappenplan van een refactoring, of de refactoring zelf, is onbekend, waardoor niet duidelijk is hoe een wijziging gecontroleerd doorgevoerd kan worden.
- Refactoren wordt gecombineerd met het toevoegen van nieuwe functionaliteit. Dit is in tegenspraak met de belangrijkste regel van refactoren dat het externe gedrag niet mag worden aangepast.
- Verscheidene refactoringsstappen worden tegelijk doorgevoerd zonder tussentijdse validatie, waardoor aan het einde niet meer helder is wat er nou eigenlijk allemaal is gewijzigd. Mensen zijn in het algemeen niet in staat om meer dan zeven[2] plus/min twee dingen in het kortetermijngeheugen te stoppen. Daarna gaat al snel het overzicht verloren.
- Handmatig refactoren is in sommige gevallen een saai werkje. Ontwikkelaars verliezen al snel hun concentratie met als gevolg dat er fouten worden gemaakt.

Alle genoemde redenen zijn terug te voeren op de ontwikkelaar. Een ontwikkelaar moet bewuster met refactoring om leren gaan, zodat er controle over de wijziging wordt gecreëerd. Ook zal het deels wegnemen van deze punten - door het proces te automatiseren - een heel scala aan mogelijkheden bieden.

Geen nieuwe functionaliteit

Om de introductie van fouten te beperken is het toevoegen van functionaliteit tijdens het refactoren niet toegestaan. Als ontwikkelaar ben je bezig met het refactoren of met het implementeren van functionaliteit. In de praktijk wordt snel tussen deze taken gewisseld, aangezien de meeste refactorings uit een klein aantal stappen bestaan. Vooral het combineren van refactorings en het even snel toevoegen van functionaliteit tijdens het refactoren is een valkuil waar veel ontwikkelaars intrappen. Het komt meer dan eens voor dat een ontwikkelaar aan een kleine aanpassing begint en dat hij na een uur tot de conclusie komt dat de wijziging als een olievlek is verspreid over het hele systeem. Als ontwikkelaar hoop je dan altijd maar op het beste en ga je flink aan de slag om de wijziging in een zo kort mogelijke tijd door te voeren. Dit is een typisch scenario waarbij een aantal refactoringsstappen of zelfs hele refactorings en nieuwe functionaliteiten tegelijkertijd worden doorgevoerd. Op zich valt er nog mee te leven dat het systeem voor een bepaalde periode 'open' ligt. Het gevolg van een grote wijziging is echter vaak dat de bestaande testcases (als deze er al zijn) na de wijziging niet meer werken. Dit heeft tot gevolg dat het externe functioneren van de software niet goed kan worden geverifieerd. Op dat moment zitten we met een probleem, omdat we niet weten welke wijziging de oorzaak is van het falen van de testcases. Het resultaat is dat er uren wordt besteed aan het oplossen van een bug die waarschijnlijk binnen tien minuten had kunnen worden opgelost als de wijziging was opgedeeld in een aantal overzichtelijke stappen. Op deze manier is het falen van een testcase altijd te herleiden naar een specifieke wijziging.

Testcases

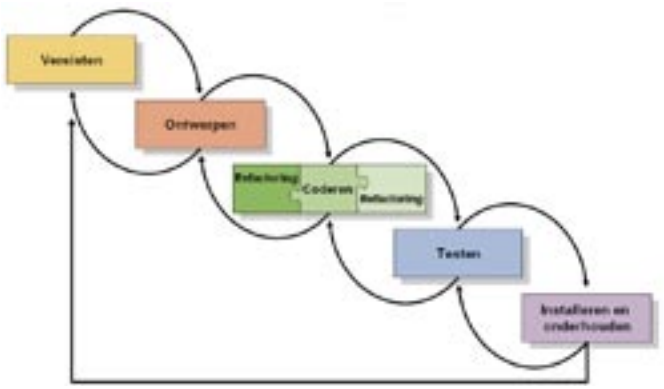
Het uitvoeren van vooraf gedefinieerde testcases na elke refactoringsstap lijkt veel werk. Maar unit-testen kunnen helpen bij de vroegtijdige opsporing van fouten en besparen uiteindelijk veel debug-tijd. Met Visual Studio 2005 is het mogelijk gebruik te maken van een gestandaardiseerd test-framework. Testcases voor methodes zijn met een druk op de knop te genereren waarbij de ontwikkelaar alleen nog het specifieke testgeval hoeft te coderen.

Het uitvoeren van testcases is dus net zo eenvoudig geworden als de compile-run-ontwikkeling die we als ontwikkelaar al gewend zijn.
Met de introductie van refactorings in Visual Studio 2005 heeft de ontwikkelaar nu de mogelijkheid om refactorings als 'Extract Method', 'Encapsulate Field', 'Rename Method' enzovoort automatisch te laten uitvoeren. De globale search-and-replace-acties komen met de 'Rename Method/Variable/Class…'-refactorings te vervallen. De wijziging kan nu snel en eenvoudig worden doorgevoerd. Belangrijker is dat we nu zeker weten dat deze correct wordt doorgevoerd. De uitvoering van de testcases komt niet te vervallen door het automatiseren van refactorings. We kunnen nu echter 'veilig' meer refactorings achter elkaar doorvoeren. Mocht er om wat voor reden dan ook toch een fout zijn geïntroduceerd, dan zijn de wijzigingen via de undo-functie snel weer terug te draaien.

Toepassen en herkennen

Het herkennen van stukken code die geschikt zijn voor refactoring is een kwestie van ervaring. Het mooie van deze ervaring is dat de meeste ontwikkelaars deze intuïtief al bezitten. Het dupliceren van code is een voorbeeld waarvan elke ontwikkelaar weet dat het niet gewenst is. Soms kan het eenmalig dupliceren van code helpen bij het halen van de deadline. Maar als ontwikkelaar weet je dat je daar op een gegeven moment een boete voor moet betalen. Dit wordt ook wel de design-debt genoemd. Wordt deze design-debt nooit betaald en bestaat de code uit veel van deze 'foute' ontwerpbeslissingen, dan is op een gegeven moment de code niet meer te onderhouden, te lezen en te hergebruiken. Het blijven verbeteren van de interne structuur van de code zorgt er dus voor dat een ontwikkelaar niet voor het dilemma komt te staan waarbij hij eigenlijk beter opnieuw kan beginnen, omdat de huidige code gewoonweg niet meer te onderhouden is. Het dupliceren van code is één van de vele voorbeelden waaraan mogelijke refactorings kunnen worden herkend. In het boek 'Refactoring' van Martin Fowler worden deze ook wel 'Bad Smells'[3] genoemd.

Voorbeelden van deze 'Bad Smells' zijn:
Duplicated Code - Het dupliceren van code is een typisch voorbeeld dat iedereen herkent. Dezelfde code moet op verschillende plekken worden bijgehouden en over het algemeen duurt het niet lang voordat een fout wordt geïntroduceerd, omdat de code op de ene plek wel wordt aangepast maar op de andere plek niet.
Long Method - Het ontstaan van een lange methode is iets wat heel geleidelijk gebeurt. Elke keer wordt een stukje functionaliteit toegevoegd, waarbij deze eigenlijk in een aparte methode had moeten worden gezet.
Large Class - Het ontstaan van een grote klasse is vergelijkbaar met het ontstaan van een lange methode. Alleen vindt het nu plaats op klasseniveau. Op een gegeven moment moet de klasse worden opgedeeld in meer klassen met elk zijn eigen verantwoordelijkheid.
Long Parameter List –Een lange parameterlijst kan op verschillende plaatsen voorkomen en ontstaat over het algemeen doordat nieuwe functionaliteit wordt toegevoegd aan een methode of event. Het is eigenlijk iets dat werd gebruikt voor de introductie van objectoriëntatie en kan dan ook met behulp van objectoriëntatie worden bestreden. Het is bijvoorbeeld mogelijk om enkel een compleet object als parameter door te geven in plaats van allemaal aparte variabelen. Hierdoor wordt het overzichtelijker wat een methode nodig heeft en is het gebruik behoorlijk vereenvoudigd. Het is immers veel vanzelfsprekender om properties op een object van waarden te voorzien dan om losse parameters door te geven.
Speculative Generality - Sommige ontwikkelaars hebben de neiging om elke klasse zo flexibel mogelijk op te zetten. De standaard reactie is dan ook vaak van 'Maar in de toekomst gaan we er *misschien* dit of dat mee doen'. Hierdoor worden klassen onnodig complex te begrijpen en te onderhouden. Door gebruik te maken van refactorings kunnen we veilig deze overhead weghalen. Mocht



Afbeelding 1. Softwareontwikkelproces inclusief refactoring

in een later stadium de functionaliteit wel gewenst zijn, dan kunnen we deze op dat moment eenvoudig met behulp van refactoring weer toevoegen.
Switch Statements - Het probleem van switch-statements is het volgende. Als er een waarde wordt toegevoegd aan bijvoorbeeld een enumeratie, dan moet op elke plek waar deze enumeratie wordt gebruikt het switch-statement worden aangepast. Door gebruik te maken van polymorphisme in plaats van enumeraties is het doorvoeren van een wijziging veel eenvoudiger. De verantwoordelijkheid hoeft dan nog maar op één plek te worden geïmplementeerd.
Comments - Commentaar in de code kan een indicatie zijn van een 'Bad Smell'. Het is niet zo dat er geen commentaar meer in de code moet worden gezet. Maar commentaar wordt vaak gebruikt om onduidelijke code te verhelderen, waarbij het toepassen van een aantal refactorings hetzelfde resultaat zou hebben.

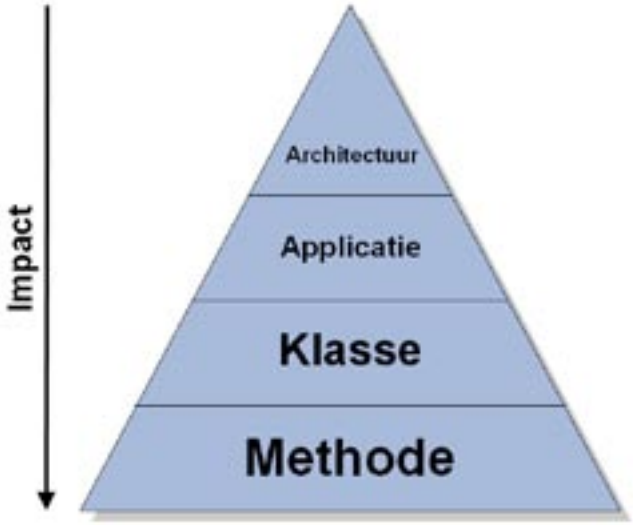
De meeste 'Bad Smells' liggen voor de hand waardoor het vinden van verbeterpunten in de code eenvoudig is. Sommige zijn echter niet zo eenvoudig en vergen meer ervaring. Het mooie van refactoren is dat je er altijd nog mee kunt beginnen, onafhankelijk van de staat van de code. Het toepassen van één refactoring heeft vaak tot gevolg dat je vanzelf een nieuwe refactoring ziet. Dit betekent nog niet dat je deze refactoring ook moet uitvoeren. Kijk altijd of een refactoring toegevoegde waarde heeft ten opzichte van waar je mee bezig bent.

Softwareontwikkelproces

Ieder softwaresysteem is onderhevig aan wijzigingen, zonder deze wijzigingen zou het systeem al snel zijn economische waarde verliezen. Omdat de opeenstapeling van veranderingen de structuur van het systeem verstoort, is het resultaat al snel een toenemende complexiteit van dat systeem. Door gebruik te maken van refactoring in het ontwikkelproces kan deze complexiteit flink worden teruggebracht. Naast dat refactoring de kwaliteit van de interne structuur van een systeem verhoogt, is het ook een communicatiemiddel tussen ontwikkelaars, architecten en reviewers.

Refactoring als communicatiemiddel – elk type refactoring heeft een unieke naam en stelt de gebruikers ervan in staat een heel verhaal te communiceren zonder het hele verhaal ook werkelijk te vertellen. De communicatie tussen reviewers en ontwikkelaars wordt veel makkelijker, omdat met weinig woorden kan worden verteld wat een reviewer verwacht van de ontwikkelaar. De vereiste is natuurlijk wel dat iedereen op de hoogte is van de betreffende refactoring.

Refactoring als voorbereiding op nieuw ontwerp – in het softwareontwikkelproces komt het geregeld voor dat een design-beslissing in een opvolgende iteratie niet aansluit bij de tot dan toe gerealiseerde implementatie. Refactoring biedt dan veelal de oplossing omdat het de ontwikkelaar de mogelijkheid biedt de code geschikt te maken voor het nieuwe design, zonder de externe functionaliteit te hoeven aanpassen.



Afbeelding 2. Refactoring-classificatie

Refactoring als testvoorbereiding – het testen van een systeem wordt makkelijker naarmate de complexiteit van een systeem afneemt. Doordat tijdens het refactoren gebruik wordt gemaakt van testcases, zal door het refactoren van bijvoorbeeld redundante code het aantal testcases afnemen, waardoor het onderhouden van de testcases makkelijker wordt.

Refactoring als afsluiting – we hebben het allemaal wel eens gedaan, code gedupliceerd uit gemak, omwille van tijd of bij gebrek aan visie. Naarmate de tijd vordert, kom je dan tot de conclusie dat je een flink aantal zaken dubbel moet gaan doen bij toekomstige aanpassingen of het onderhoud. Dit probleem wordt verergerd doordat het onderhoud over het algemeen niet wordt uitgevoerd door de initiële ontwikkelaars. Onnodige complexiteit binnen het systeem is dan van directe invloed op de tijd die nodig is om het systeem te leren kennen en begrijpen. Het is in ieders belang om de interne structuur van een systeem zo simpel, overzichtelijk en onderhoudbaar mogelijk te houden. Refactoring biedt hier een uitkomst. Door refactoring integraal onderdeel te laten uitmaken van het softwareontwikkelproces kan een hogere mate van kwaliteit worden behaald. Er zijn diverse plaatsen waar refactoring kan worden ingepast in het ontwikkelproces. Volgens ons zijn er twee plaatsen waar refactoring de meeste winst oplevert; voor een coderingsiteratie als voorbereiding op een nieuw ontwerp en na de coderingsiteratie ter voorbereiding op het testen of het toekomstig onderhoud. Afbeelding 1 geeft een voorbeeld van een iteratief ontwikkelproces met geïntegreerde refactoring.

Refactorings in .NET tooling

Refactoring	Borland Delphi 2005		Visual Studio 2005 Beta 2		Refactor! for Visual Basic 2005 Beta 2
	Delphi	C#	VB	C#	VB
Rename Type	✓	✓	✓	✓	
Rename Variable	✓	✓	✓	✓	
Rename Method	✓	✓	✓	✓	
Extract Method	✓			✓	✓
Encapsulate Field				✓	✓
Extract Interface				✓	
Reorder parameters				✓	✓
Promote local variable to parameter				✓	
Extract Property					✓
Create Overload					✓
Reverse Conditional					✓
Simplify Conditional					✓
Introduce Local					✓
Introduce constant					✓
Inline Temp					✓
Replace Temp with Query					✓
Split Temporary Variable					✓
Move initialization to declaration					✓
Split initialization from declaration					✓
Move declaration near reference					✓

Afbeelding 3. Refactorings in .NET-tooling

Classificatie

Refactorings zijn op verschillende manieren te classificeren. Door op voorhand naar de impact/scope van een refactoring te kijken is een erg handige methode. Een refactoring zoals 'Extract Method' heeft tot gevolg dat er wijzigingen op klasseniveau worden doorgevoerd. Een refactoring zoals 'Rename Variable' heeft echter alleen maar wijzigingen op methodeniveau tot gevolg. De impact/scope van de 'Rename Variable'-refactoring is dus kleiner dan die van 'Extract Method'. Afbeelding 2 geeft een overzicht van de impact die refactorings kunnen hebben op de software.

Methode - Refactorings op methodeniveau wijzigen de implementatie van één methode. Code buiten de methode wordt niet aangepast.

Klasse - Refactorings op klasseniveau wijzigen de implementatie van één klasse. Hiertoe wordt bijvoorbeeld de 'Rename Method'-refactoring gerekend. De 'Rename Method' kan echter, afhankelijk van de zichtbaarheid van de methode voor andere klassen, ook wijzigingen in andere klassen tot gevolg hebben. In dit geval is de impact dus niet op klasseniveau, maar op applicatieniveau. Refactorings kunnen dus, afhankelijk van de context waarin ze worden toegepast, in meer classificatielagen thuishoren.

Applicatie - Refactorings op applicatieniveau wijzigen de implementatie van verscheidene klassen. De introductie van een nieuwe basisklasse in een bestaande hiërarchie is een voorbeeld waarbij meer klassen deelnemen aan de refactoring; ze worden immers afgeleid van de nieuwe basisklasse.

Architectuur - Refactorings op architectuurniveau zijn vergelijkbaar met die op applicatieniveau. Refactorings op architectuurniveau hebben echter als doel de implementatie van een ontwerp aan te passen (algoritme/strategie). Het naar een design-pattern[4] toe refactoren is een typisch voorbeeld van een architectuurrefactoring.

Door gebruik te maken van de beschreven classificatie is op voorhand de impact/scope van een refactoring bekend. Hiermee kan dus, afhankelijk van de status van het project, worden besloten om de refactoring nu, later of helemaal niet uit te voeren. We refactoren niet om het refactoren, maar proberen in de beschikbare tijd een zo goed mogelijk product op te leveren. Dit kan betekenen dat er soms, door niet te refactoren, aan kwaliteit/flexibiliteit moet worden ingeleverd.

Ondersteuning in .NET-tooling

Refactorings kunnen in de vorm van stappen worden uitgedrukt. Dit maakt ze geschikt voor automatisering. Een geautomatiseerde refactoring is snel, effectief, secuur, reproduceerbaar en heeft alleen maar kennis van de code die er al is. Er zal dus nooit nieuwe functionaliteit worden toegevoegd. Sinds kort bevatten de nieuwste versies van zowel Visual Studio als Borland Delphi dan ook refactoring-functionaliteit out-of-the-box. Op zich geen vreemde zaak, omdat de compilers van beide omgevingen uitstekend in staat zijn de contextgevoelige informatie die nodig is voor refactoring aan te leveren. Denk maar eens aan het hernoemen van een property op een klasse. Hier mogen alleen maar de referenties naar en binnen die specifieke klasse worden aangepast en al het andere, met eenzelfde naamgeving, moet worden genegeerd. Voor een compiler is dit kinderspel, omdat deze al exact weet welke property waar wordt gebruikt.

De refactorings die zijn geïntroduceerd in de ontwikkelomgevingen bevinden zich voornamelijk op de methode- en klasseniveaus en richten zich op het uit handen nemen van repetitieve en foutgevoelige taken. Afbeelding 3 geeft een overzicht van de refactorings die

door Borland en Microsoft zijn toegevoegd aan de ontwikkelomgevingen. Op zich is het aanbod van refactorings, die Microsoft en Borland in de ontwikkelomgevingen aanbieden, niet spectaculair. Door gebruik te maken van de mogelijkheden die Microsoft en Borland in hun ontwikkelomgevingen hebben ingebouwd, kunnen externe partijen met add-ins zelf refactoring-functionaliteit aan de IDE toevoegen. Borland heeft hiervoor de OpenTools API geïntroduceerd en Microsoft biedt via het Visual Studio Industry Partner (VSIP) programma een SDK aan die kan worden gebruikt. Developer Express Inc is een van de bedrijven die zo'n add-in heeft geschreven. Omdat de Visual Basic .NET-entiteit in Visual Studio veel minder refactoring-functionaliteit bevat dan de C#-entiteit heeft Developer Express samen met Microsoft een gratis add-in voor Visual Studio ontwikkeld. Deze add-in voegt diverse refactorings op methode- en klasseniveau toe aan de Visual Basic.NET-entiteit en is te downloaden via de volgende link: http://msdn.microsoft.com/vbasic/downloads/2005/tools/refactor/. Hetzelfde bedrijf heeft ook een professional versie uitgebracht die naast een flink aantal nieuwe refactorings voor C# ook de mogelijkheid biedt om eigen ontwikkelde refactorings toe te voegen aan Visual Studio.

Stevig houvast

De auteurs zijn van mening dat refactoring een vast onderdeel zou moeten zijn van het softwareontwikkelproces omdat het, mits goed uitgevoerd, altijd loont en niet veel tijd hoeft te kosten. Refactoring biedt een stevig houvast voor ontwikkelaars en het team waarin zij meewerken. Hoewel de refactorings die door Microsoft en Borland zijn toegevoegd aan de ontwikkelomgevingen nog niet zo spectaculair zijn, bieden ze toch al genoeg mogelijkheden om de complexiteit van een bestaand of nieuw systeem te vereenvoudigen. Het wachten is op add-ins die het mogelijk maken om meer refactorings geautomatiseerd door te voeren op applicatie- en architectuurniveau, omdat daar de meeste winst valt te behalen.

Ewart Nijburg en **Patrick Vorgers** zijn beiden als technisch architect werkzaam bij de Technology Consulting-afdeling van Ordina Software Integration & Development (www.ordina.nl). Hun specialisaties zijn softwarearchitecturen, high availability en software performance engineering. Voor vragen en opmerkingen kun je ze bereiken op ewart.nijburg@ordina.nl en patrick.vorgers@ordina.nl

Referenties:

- Design patterns: Elements of reusable code, Gamma et al. ISBN 0201633612
- Refactoring: Improving the design of existing code, Fowler et al. ISBN 0201485672
- Martin Fowler: www.refactoring.com
- MSDN: http://msdn.microsoft.com/vbasic/downloads/2005/tools/refactor/
- DevExpress Refactor Pro for C#: http://www.devexpress.com/Products/NET/

Noten

(1) Visual Studio 2005 ondersteunt standaard refactoring voor C#.

(2) Miller 1956, http://www.well.com/user/smalin/miller.html

(3) Zie Martin Fowler 'Refactoring – Improving the design of existing code' pag. 75 'Bad Smells'

(4) Een design-pattern is een beschrijving van een veelvoorkomend probleem inclusief de oplossing van de kern van het probleem en wel zo dat deze herbruikbaar is. Zie Gamma et al. 'Design patterns, elements of reusable code'.