Diederik Krols is Chief Architect bij Real Software

Anti-patterns: van amusement naar werkinstrument

FILTERING NOODZAKELIJK BIJ DYNAMISCHE ANALYSE

Dit is niet het zoveelste artikel over design patterns. Ik ga ervan uit dat een ruime meerderheid van de professionele ontwikkelaars die kent en toepast. Neen, we gaan het hebben over de zogenaamde anti-patterns. Die hebben met de klassieke design patterns gemeen dat ze een reeks courante problemen beschrijven, samen met één of meer oplossingen voor deze problemen.

De reeks courante problemen bevat zaken als schaalbaarheid, flexibiliteit, onderhoudbaarheid, en het beheer van resources. Terwijl de klassieke design patterns ontdekt werden door analyse en synthese van succesvolle ontwerpen, werden de anti-patterns ontdekt door analyse en synthese van mislukte ontwerpen. Anti-patterns beschrijven voor de hand liggende oplossingen die meer schade toebrengen dan het probleem dat ze uit de wereld willen helpen: "An anti-pattern is something that looks like a good idea, but which backfires badly when applied" (quote van Jim Coplien).

Anti-pattern catalogi

Het boek 'AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis' uit 1999 bevat de eerste catalogus van anti-patterns. De patterns werden er onderverdeeld in drie categorieën: ontwikkeling, architectuur, en projectleiding. Hier zijn enkele bekende voorbeelden uit de eerste twee categorieën:

- Spaghetticode: ongestructureerde, langdradige code met door elkaar gevlochten logica.
- Raviolicode: een toepassing die bestaat uit een grote reeks kleine klassen, waardoor het overzicht verloren gaat. Dit is het OO-equivalent van spaghetticode.
- Blob: één klas die het merendeel van de verantwoordelijkheden van een volledige applicatie draagt.
- Golden Hammer: het te pas en te onpas gebruiken van een technologie, alleen maar omdat je er kennis van hebt.

Die eerste anti-pattern catalogus had zijn populariteit grotendeels te danken aan het hoge Dilbert-gehalte. Alle anti-patterns werden uitvoerig gestoffeerd met herkenbare anekdotes van foutieve code, ontwerpen en managementstijlen. En geef toe: elke IT-er leest graag verhalen over mislukte projecten, zolang hij maar niet zelf bij die projecten betrokken is. Na die catalogus zijn anti-patterns wat in de vergeethoek geraakt. Daar is verandering in gekomen dankzij het volwassen worden van refactoring, vooral onder invloed van Kent Beck en Martin Fowler. Het inmiddels overbekende refactoring-boek voorzag de anti-patterns van concretere benamingen: 'Blob' werd bijvoorbeeld gewoon 'Large Class'. Er wordt ook aandacht besteed aan technieken om anti-patterns op te sporen en te vervangen door correcte design patterns. Anti-patterns hebben sindsdien niet alleen maar amusementswaarde, maar ze zijn aan het uitgroeien tot een nuttig werkinstrument voor ontwikkelteams.

Code met een geurtje

Anti-patterns worden ook wel eens '*smells*' genoemd; code waarin anti-patterns voorkomen is dus code met een geurtje. Fowler en Beck beschrijven die geurtjes in een platte lijst en dat is niet erg praktisch. Ik zal daarom proberen om de 22 klassieke smells in categorieën op te delen. Ik heb er ook het 'dead code' pattern aan toegevoegd. De classificatie is als volgt:

1. Overtredingen tegen koppeling en cohesie

Een elegant ontwerp dat wordt gekenmerkt door een lage koppeling en hoge cohesie. Dit is de basis van de hele reeks GRASP (General Responsibility Assignment Patterns) en GoF (Gang of Four) design patterns. De volgende anti-patterns veroorzaken een te hoge koppeling en/of te lage cohesie:

- Kettingaanroep (Message Chains): overdreven gebruik van koppelingen in een method. Deze klassieker onder de anti-patterns is ook bekend als de 'Wet van Demeter'.
- Makelaar (Middle Man): een klasse die zelf niks doet, maar alles delegeert aan een andere.
- Stalker (Feature Envy): een klasse die meer geïnteresseerd is in een externe method dan de klasse zélf die deze method implementeert.
- Ongewenste intimiteiten (Inappropriate Intimacy): twee of meer klassen die in elkaar verstrengeld zijn.
- Onvoltooide symfonie (Incomplete Library Class): een functiebibliotheek die belangrijke functies mist, waardoor iedereen telkens opnieuw de nodige uitbreidingen moet bouwen.

2. Zwaarlijvige code

Het is een beetje als bij ons mensen: code die onnodig omvangrijk is, is waarschijnlijk niet gezond. Deze categorie bevat de volgende anti-patterns:

- Lange method (Long Method): een method die te lang en/of te complex is om te begrijpen.
- Vette kluif (Large Class): een klasse die te veel probeert te doen.
- Drempelvrees (Primitive Obsession): te hard vasthouden aan individuele atomaire waarden in plaats van er een volwaardige klas van te maken. Dit is meteen ook de oorzaak van de twee volgende bad smells.
- Lange argumentenlijst (Long Parameter List): een te lange en dus onoverzichtelijke lijst van parameters.
- Dataklompjes (Data Clumps): groepjes losse gegevens die altijd samen gebruikt worden.
- Veel commentaar (Comments): complexe code moet niet voorzien worden van veel commentaar, maar moet vereenvoudigd worden.

Categorie	Anti-pattern	FxCop Regel(s)
Zwaarlijvige code	Long Parameter List	CA1005: Avoid excessive parameters on generic types
		CA1025: Replace repetitive arguments with params array
Overtollige code	Dead Code	CA1811: Avoid uncalled private code
		CA1804: Remove unused locals
		CA1801: Avoid unused parameters
		CA1812: Avoid uninstantiated internal classes
		CA1806: Do not ignore method results
		CA1823: Avoid unused private fields
OO-overtredingen	Refused Bequest	CA2222: Do not decrease inherited member visibility

Tabel 1: Lijst van FxCop-regels in anti-pattern context

3. Overtollige code

Na verloop van tijd vind je in elke grote applicatie wel code die geen toegevoegde waarde meer heeft. De volgende bad smells behoren tot deze categorie:

- Luie klasse (Lazy class): een klasse die te weinig doet om z'n gewicht te kunnen dragen.
- Dataklasse (Data class): een klasse met enkel getters en setters rond attributen.
- Dubbele code (Duplicate Code): het knip-en-plak-syndroom.
- Kristallen bol (Speculative Generality): code waarvan je hoopt dat hij toekomstige uitbreidingen of hergebruik vergemakkelijkt, maar die voor de rest gewoon in de weg zit.
- Dode code (Dead Code): code die nergens aangeroepen wordt of die zelfs niet aangeroepen kan worden.

4. Starre code

De categorie 'starre code' bevat de smells die zondigen tegen het (ideale) principe dat elke wijziging slechts op één klasse impact zou mogen hebben. Als die smells de boventoon krijgen, krijg je ofwel een flessenhalsprobleem voor je onderhoudsopdrachten, ofwel worden die te uitgebreid. Het is in elk geval een indicatie dat grote delen van het technisch en/of functioneel ontwerp van de toepassing niet meer overeenkomen met de werkelijkheid. Deze categorie bevat twee anti-patterns:

- Zwart schaap (Divergent Change): bij ongeacht welke wijziging moet eenzelfde klas aangepast worden, en
- Spijkerbom (Shotgun Surgery): het omgekeerde, elke kleine functionele wijziging heeft impact op heel veel klassen.

5. Overtredingen tegen objectoriëntatie

Deze categorie bevat een reeks typische gemiste kansen om OO-principes als overerving, inkapseling, en polymorfisme te hanteren. Deze categorie omvat:

- Voorwaardelijke wijs (Switch Statements): zelf de logica bepalen op basis van types, in plaats van polymorfisme te gebruiken.
- Arrivist (Temporary Field): een method-variabele met klasse-scope.
- Parallelle overerving (Parallel Inheritance Hierarchies): telkens wanneer je een subklasse definieert, moet je er ergens anders ook één aanmaken.
- Gedeeltelijke overerving (Refused Request): een subklasse wil niet alles overerven van zijn superklasse.
- Twee-eiige eenlingen (Alternative Classes with Different Interfaces): eenzelfde functionaliteit op twee verschillende manieren implementeren.

Tools met een neus

Weten wat anti-patterns zijn, volstaat natuurlijk niet om ze te vermijden: je moet ze ook kunnen opsporen. Iedereen weet wel wat spaghetticode is, maar bijna niemand staat er bij stil dat je kunt meten of je ze zelf ook niet aan het schrijven bent. Door Visual Studio juist te configureren en te omringen met een beperkt aantal tools kan je die perfect ombouwen tot een ware 'anti-pattern detector'. Daarmee kan je tijdens het ontwikkeltraject regelmatig of zelfs continu een geuranalyse uitvoeren.

Statische analyse

Door een analyse van de source-code of de IL-code in de assemblies kan je vrij gemakkelijk de meeste anti-patterns ontdekken. Ik zal kort drie soorten tools bespreken: tools voor het toetsen van regels, tools voor het berekenen van metrieken en specifieke tools. De meest gebruikte tool voor het toetsen van de code aan een reeks regels is FxCop. Sinds Visual Studio 2005 is die nu helemaal geïntegreerd in de ontwikkelomgeving. Met FxCop is het zeer eenvoudig om bijvoorbeeld overtollige code te detecteren. Tabel 1 bevat een lijst van de anti-patterns die je met FxCop kunt bestrijden. Je kunt aan FxCop ook zelfgeschreven regels toevoegen.

Een tweede categorie tools voor statische analyse bevat de instrumenten waarmee je metrieken voor objectgeoriënteerde software kunt berekenen. Hiermee vind je onmiddellijk alle anti-patterns die overtredingen zijn op OO-principes, diegene die zondigen tegen koppeling en cohesie, en ook zwaarlijvige en overtollige code. Sommige van die metrieken zijn gemakkelijk te berekenen, zoals het aantal lijnen source-code (SLOC) per method en per klasse. Andere metrieken zijn dan weer wat moeilijker: efferente en afferente koppeling (Ce en Ca), directe koppeling tussen klassen (ABC), abstractie (A), instabiliteit (I), de afwijking ten opzichte van de ideale lijn (D), de relationele cohesie van assemblies (H), het gebrek aan interne cohesie van methods (LCOM), en de cyclomatische complexiteit (CC). Tabel 2 bevat een lijst van anti-patterns die je op basis van OO-metrieken kunt ontdekken. Een gedetailleerde analyse van elk van deze metrieken zou te ver voeren, maar wie geïnteresseerd is moet zeker het NASA-artikel uit de lijst met referenties eens doorlezen. Het lijstje met tools voor statische analyse kan volledig gemaakt worden met een categorie 'gerichte tools'. Hierin plaats ik bijvoorbeeld Simian (SIMIlarity ANalyzer), een tool waarmee je heel gemakkelijk gedupliceerde source-code kunt opzoeken.

Dynamische analyse

Wanneer je de ontwikkelomgeving hebt uitgebreid met al deze meetinstrumenten, is de kans groot dat je bedolven wordt onder de resultaten. Enige filtering is dan noodzakelijk, want het is echt onbegonnen werk om constant jacht te maken op alle mogelijke slechte geurtjes. Je moet bijvoorbeeld rekening houden met het ontwerp (in dataset-centric applicaties heeft het geen zin om dataklassen op te sporen en te elimineren), en met de procesfase waarin jij je bevindt (zoeken naar dubbele code doe je nog niet in het begin van het ontwikkeltraject). De beste manier om gericht te zoeken naar slechte geurtjes, is rekening houden met het relatieve belang van de code: de vaak uitge-

Categorie	Anti-pattern	Metriek(en)
Zwaarlijvige code	Long Method	SLOC
	Large Class	SLOC
Overtollige code	Speculative Generality	I, A, D
Starre code	Divergent Change	I, A, D, LCOM
	Shotgun Surgery	I, A, D
OO-overtredingen	Switch Statements	CC

Tabel 2: lijst van OO-metrieken in anti-pattern context

voerde code en de code die cruciale bedrijfsprocessen ondersteunt, verdienen meer aandacht dan de andere. Je zult merken dat hier vaak de 80/20-regel van toepassing is: door 20% van de code op te volgen, dek je 80% van de belangrijke functionaliteit. De op te volgen code vind je door de code-coverage en de call-graph te bekijken van representatieve functionele testen. Dit houdt meteen in dat je niet alleen over een coverage-tool moet beschikken, maar ook over een testing-framework. Wie al overgeschakeld is op Visual Studio 2005 Team System, heeft alle wapens in handen om meteen op jacht te gaan. Wie nog niet is overgeschakeld op die laatste versie, moet nog een beroep doen op externe tools zoals Nunit en NCover. Dynamische analyse helpt je niet alleen om je focus te houden, maar kan ook zelf slechte geurtjes aan het licht brengen: in de gegenereerde call-graphs zullen overtredingen tegen koppeling en cohesie duidelijk zichtbaar worden in de vorm van onverwachte pijlen of wolken van pijlen.

Analyse van wijzigingen

De geurtjes uit de categorie 'starre code' zijn niet op te lossen door te kijken naar de code, of door het runnen van testscenario's. Je kunt ze enkel opsporen door een nauwkeurige opvolging van de evolutie van de source-code: je moet in beeld kunnen brengen op welke locatie in de code aanpassingen uitgevoerd werden als gevolg van een werkopdracht. Gebruikers van Visual Studio 2005 Team System hebben alweer een stapje voor: dankzij het integreren van de methodologie kan je work-items (werkopdrachten) koppelen aan wijzigingen in de source-code (check-ins). Aangezien dit een CMMI-niveau 2 vereiste is, moet het mogelijk zijn om via de rapporteringen van MSF-CMMI een duidelijk beeld te vormen van de impact van wijzigen op de source-code. Daaruit kan je afleiden of je applicatie al dan niet starre code bevat.

De realiteit

Zoals je ziet, kom je met Visual Studio 2005 al een heel eind: change management, dynamische analyse, en een stuk van de statische analyse worden standaard afgedekt. Enkel in het berekenen van OO-metrieken schiet de ontwikkelomgeving nog tekort. Alle informatie kan echter via Reflection opgezocht worden. De anti-patterns die op metrieken gebaseerd zijn, zijn dan ook redelijk eenvoudig op te sporen. Het kan in principe zelfs al vóór de compilatie. Jammer genoeg is deze functionaliteit nog niet vervat in Visual Studio, en moeten we een beroep doen op externe tools, zoals NDepend en Vil. Visual Studio beschikt echter over alle troeven: de Visual Studio Plugin-infrastructuur is het ideale medium voor statische analyse en het berekenen van OO-metrieken. Spijtig genoeg wordt het hiervoor nog niet gebruikt.

Om het gehele anti-pattern plaatje te dekken moeten we dus nog steeds gebruikmaken van externe tools. Dat heeft een aantal nadelen:

- De documentatie van open source-projecten laat vaak te wensen over. Ga bijvoorbeeld maar eens op zoek naar een duidelijke beschrijving van het NDepend-configuratiebestand. Als je dergelijke informatie niet onmiddellijk vind, kan de interesse in de tool vlug bekoeld zijn.
- Sommige van de tools –zoals Simian- zijn zeer generiek en niet echt toegespitst op .NET-projecten. Dit houdt in dat je relatief veel tijd moet steken in de configuratie van de tool. Het heeft bijvoorbeeld geen zin om op zoek te gaan naar dubbele code in de door Visual Studio gegenereerde code, bijvoorbeeld de code voor Typed DataSets en Forms. Het is echter niet eenvoudig om Simian deze te laten negeren.
- Andere tools zijn weer te specifiek: devMetrics is bijvoorbeeld een zeer volledige tool voor het berekenen van allerhande metrieken (zelfs als Visual Studio plugin), maar dan enkel voor C#, en niet voor andere .NET-talen.

Als gevolg hiervan hebben sommige van deze tools, ondanks een grote toegevoegde waarde, nooit echt een groot publiek kunnen bereiken.

Refactoring

Wanneer je tijdens zo'n geuranalyse anti-patterns detecteert, dan komen de klassieke design patterns van pas om het probleem op de correcte wijze op te lossen. Dit valt onder de noemer *Refactoring*. Refactoring werd geïntroduceerd door de thesis van William Opdyke, en de best practices werden in catalogusvorm gebundeld door Martin Fowler. Een reeks van de refactoring patterns is inmiddels ingebouwd in onze ontwikkeltools: ofwel als third party-plugin, ofwel als standaard onderdeel ervan (Visual Studio 2005). Ik zal hierover niet verder uitweiden: het onderwerp is in vorige edities van het .NET Magazine al aan bod gekomen. Oorspronkelijk was refactoring vooral belangrijk tijdens de onderhoudsfase, maar momenteel worden de technieken ook toegepast tijdens de ontwikkeliteraties. Een en ander is mogelijk door de automatisering van refactoring: het wordt hoe langer hoe gemakkelijker om anti-patterns in code op te sporen en te vervangen door correcte design patterns. Ik maak hier graag de vergelijking met Microsoft Word: dat is ook in staat om verbeteringen in teksten in willekeurige talen voor te stellen, en zelfs om die verbeteringen automatisch uit te voeren. Ik verwacht in de komende jaren enorme vooruitgang op het domein van geautomatiseerde refactoring. Automatische refactoring laat toe dat ontwikkelteams snel de binnenkant van hun toepassing kunnen aanpassen, zonder de buitenkant te veranderen. In extreme gevallen kan je zelfs het ontwerp en de architectuur ervan nog drastisch aanpassen op een willekeurig moment in het ontwikkeltraject. De klassieke *Big Upfront Designs* uit bureaucratische methodologieën kunnen daarmee vermeden worden. Dit is – onder andere- de basis van de zogenoemde *Agile*-methodologieën, zoals eXtreme Programming en Scrum.

Een goede neus

In een ideale wereld is het uitvoeren van geuranalyses volledig ingebakken in de ontwikkelomgeving en hoef je geen beroep meer te doen op externe tools. Visual Studio 2005 is sterk verbeterd op het gebied van integratie van statische en dynamische analyse, en change management. Het wordt steeds gemakkelijker om 'slechtgeurende' code te herkennen en om te zetten naar code die wel voldoet aan de regels van de kunst. We staan niet ver meer van de automatisering van dit proces: stilaan zijn anti-patterns aan het uitgroeien van louter amusement naar nuttig werkinstrument.

Diederik Krols is Chief Architect bij het Belgische Real Software. Hij vult zijn dagen met .NET softwarearchitectuur en Continuous Integration, balancerend tussen eXtreme Programming, Scrum, en CMMI. Hij blogt samen met zijn collega's op http:\\www.realdn.net, en solo op http:\\www.dotbay.be.

Referenties

Het boek AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis van William J. Brown, Raphael C. Malveau, Hays W. "Skip" McCormick, Thomas J. Mowbray
Het boek Refactoring: Improving the Design of Existing Code by Martin Fowler
Het artikel Applying and Interpreting Object Oriented Metrics van het NASA Software Assurance Technology Center: http://satc.gsfc.nasa.gov/support/STC_APR98/apply_oo/apply_oo.html