Refactoring.md

Refactoring

Clean code

Het belangrijkste doel van refactoring is alle technische erfenis bekampen (*technical debt*). Refactoring wil een hoop rommelcode omvormen tot een zuiver geheel dat gekenmerkt wordt door een doorzichtig en eenvoudig ontwerp.

Eigenschappen van propere code (*clean code*):

Propere code is onmiddellijk te begrijpen voor programmeurs

Foute namen voor variabelen, te grote classes, te lange functies - dit alles maakt je code slordig en achteraf moeilijk te doorgronden.

Propere code bevat geen copieen

Telkens wanneer je code verveelvoudigt, moet je achteraf ook telkens in elk copie dezelfde aanpassing doen! Je moet dit niet alleen onthouden: dit vraagt ook meer inspanning en vertraagt je ontwikkelproces.

Propere code is zo eenvoudig mogelijk en bevat een minimaal aantal classes en te verplaatsen code

Minder code en eenvoudigere code betekenen dat je minder in je hoofd moet houden. Minder code betekent minder bugs. Code betekent verantwoordelijkheid en belasting: houd code kort en eenvoudig ("een goede programmeur is zo lui mogelijk").

Propere code doorstaat alle testen

Je weet dat je code slecht in elkaar steekt wanneer je slechts 95% van je unit test suite succesvol kan uitvoeren. Je weet dat je in de problemen zal geraken wanneer je 0% *code coverage* hebt.

Propere en geteste code is makkelijker en goedkoper te onderhouden!

Technical debt

Iedereen doet zijn best om aanvankelijke propere code te schrijven. Wanneer precies wordt code dan "vuil"? De metafoor “technical debt” (technische erfenis) werd voor het eerst aangedragen door Ward Cunningham. Je kan een lening vragen aan de bank: deze laat je toe sneller te kopen dan je normaal zou kunnen doen. Je kan echter makkelijk zoveel lenen en kopen dat je uiteindelijk je lening niet meer kan terugbetalen. Hetzelfde gebeurt met code: je kan snel ontwikkelen door code te copieren, een extra parameter in te voeren, de eerste naam te kiezen die in je opkomt voor je variabele of method, het schrijven van kleine unit testen vermijden, de code slecht opdelen over bestanden en bibliotheken, geen documentatie voorzien, ga zo maar door, tot je op een punt komt dat ontwikkelen zeer moeizaam wordt en je verplicht bent om de ontstane mesthoop te herwerken en toch testen te schrijven: op dat moment betaal je je schuld zwaar uit en moet je door je onvoorziene tijdsbesteding toegeven dat je niet goed bezig was.

Oorzaken van *technical debt*

Business druk

Soms is de druk in een project zo groot dat je features uitbrengt die nog niet volledig afgerond en getest zijn. Op dat moment moet later aangepast worden met patches en "lappen" om de onvolkomen punten in je code te verbergen.

Onbegrip voor de gevolgen van *tecnhical debt*

Management ziet vaak de waarde van refactoring minder. Als bovendien de programmeur dit niet doet, ontstaat er een manier van werken waarbij de code uiteindelijk toch moeilijk onderhoudbaar blijkt. Op dat moment begint men dan vaak "opnieuw", vaak in een andere omgeving. Nodeloos te stellen dat dit ook zeer kostelijk is (business wil wel betalen voor nieuwe software, maar niet nog een keer voor wat al gerealiseerd werd, en dus moet de vernieuwing vaak verdoken gebeuren onder grote druk).

Onderlinge afhankelijkheid niet beperken

Het project begint te lijken op een "monoliet", eerder dan het product van individuele modules die samenwerken en elkaar versterken. Een wijziging heeft in dat geval vaak impact op het geheel in plaats van op een deel. Samenwerken in een team wordt veel moeilijker omdat het werk van de ene dat van de andere beinvloedt.

Gebrek aan testen

Het gebrek aan onmiddellijke feedback wanneer je iets ontwikkeld hebt, goed of fout, kan grote gevolgen hebben: je "vergeet" het stuk code, het gaat in productie en kan grote problemen veroorzaken. Op dat moment moet er snel een stuk code aangepast worden om het gebrek te verbergen - iedereen weet ervan, de gebruiker heeft er last van, het ontwikkelingsproces valt even stil en de kwalijke erfenis wordt groter.

Gebrek aan documentatie

Dit vertraagt het betrekken van nieuwe collega's. Op dat moment moeten sleutelfiguren opleiding geven en kunnen ze niet verder werken.

Gebrek aan interactie tussen teamleden

Indien er niet regelmatig kennis uitgewisseld wordt, opereren de leden van een team op basis van verouderde uitgangspunten en inzichten. Dit geldt nog meer wanneer nieuwe collega's onvoldoende opgeleid worden: zij introduceren dan zonder het vaak te willen een andere manier van werken, een ander begrip van het domein, ...

Gelijktijdige ontwikkeling in branches gedurende lange tijd

Te lang apart doorwerken op een versie van dezelfde code betekent dat er verhoudingsgewijs veel langer gewerkt moet worden om de verschillen opnieuw samen te brengen.

Uitgestelde refactoring

De vereisten van een project worden voortdurend bijgestuurd. Na verloop van tijd moeten stukken code eveneens verwijderd of bijgestuurd worden. Des te langer hiermee gewacht wordt, des te groter is de impact want intussen wordt er steeds nieuwe code bijgeschreven die gebruik maakt van de aan te passen code.

Gebrek aan *compliance monitoring*

Niemand bewaakt de kwaliteit van het geheel: het gevolg is dat elke programmeur naar eigen kennis, inzicht en goesting implementeert.

Incompetentie

Wanneer de programmeur er gewoon niet in slaagt goede code te produceren ... .

Wanneer "refactoren"

Rule of Three

- Wanneer je iets voor de eerste keer doet, doe het gewoon
- Wanneer je iets voor de tweede keer moet doen, huiver een keer, maar doe het gewoon opnieuw
- Wanneer je iets voor de derde keer moet doen, start refactoring

Wanneer je een feature toevoegt

Refactoring helps andermans code begrijpen. Wie na jou komt, heeft er trouwens ook voordeel van. Een nieuw feature is bovendien makkelijker toe te voegen aan propere code. you understand other people’s code.

Wanneer je een bug fixt

Leg niet gewoon een "lap", maar refactor. Je krijgt een beetje extra tijd toegemeten, het wegwerken van de bug wordt geapprecieerd en je verbetert tegelijk de basis voor de toekomst.

Tijdens een code review

Dikwijls de laatste kans om code op te kuisen voor deze in productie komt. Vier ogen zien altijd meer dan twee ... . Vaak het moment om kleine dingen op te kuisen zodat je achteraf meer tijd hebt voor de ernstige problemen.

Hoe refactoren

- Als een sequentie van kleine wijzigingen die de bestaande code telkens iets verbeteren terwijl het geheel blijft werken. Kan het niet op deze manier, dan moet vaak een groter stuk code volledig herschreven worden.
- Geen nieuwe functionaliteiten introduceren bij het refactoren.
- Alle unit testen moeten blijven slagen.

Wat moet refactored worden? ("smells")

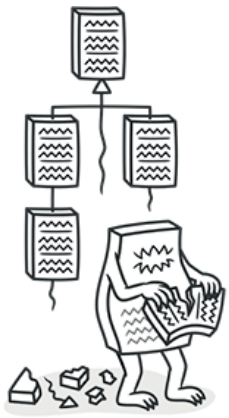
Op dit moment vind je hieronder slechts een summier overzicht: wanneer begint code "te stinken" en is opkuisen aan de orde. Later zullen we op deze aspecten dieper ingaan.

Bloaters

Stukken code, functies en classes die zo groot geworden zijn dat er nog moeilijk mee kan gewerkt worden. Vaak ontstaat dit pas na verloop van tijd.

- Long Method
- Large Class
- Primitive Obsession
- Data Clumps
- Long Parameter List

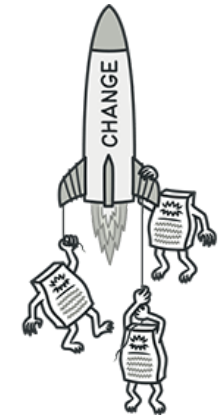
Object-Orientation Abusers



Foutief toegepaste OO-concepten:

- Alternatieve classes met verschillende interfaces
- Refused Bequest
- Temporary Field
- Switch Statements

Change Preventers



Teken aan de wand: wanneer je op een (1) plaats iets moet aanpassen, moet je plots op veel andere plaatsen ook aanpassingen doorvoeren.

- Divergent Change
- Parallel Inheritance Hierarchies
- Shotgun Surgery

Dispensables

Wat overbodig is, moet weg. Geen stukken code in commentaar, bijvoorbeeld: hiervoor dient je versiebeheersysteem (vb. Git).

- Comments
- Duplicate Code
- Data Class
- Lazy Class
- Dead Code
- Speculative Generality

Couplers

Te sterke koppeling tussen classes of omgekeerd, teveel delegatie in plaats van koppeling.

- Feature Envy
- Incomplete Library Class
- Middle Man
- Inappropriate Intimacy
- Message Chains

Refactoring technieken

Op dit moment vind je hieronder slechts de grote categorieen. We zullen hier later verder op ingaan.

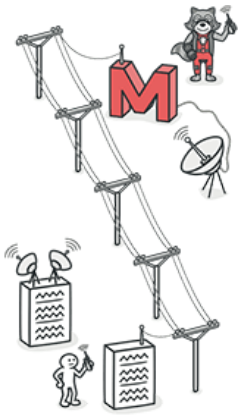
Composing Methods

Moving Features between Objects

Organizing Data

Simplifying Conditional Expressions

Simplifying Method Calls



Dealing with Generalization