Versie 1.0

ir. K.J. van der Lelij

SEMESTER SE-S2 – AGILE & OO PROGRAMMEREN
OPT3 - ONTWERPEN, PROGRAMMEREN & TESTEN – DEEL 3

Cheat Sheet Code Smells

DE HAAGSE HOGESCHOOL

Aanpak

Waar start je als je code stinkt?

Eigenlijk maakt het niet zo heel veel uit waar je begint. Als je uit onderstaande tabel een willekeurige ingang neemt (bijv. *switch-statements* als *code smell*) en daarvoor een goede aanpak selecteert (bijv. *replace conditional with polymorphism*), dan ruim je daarmee een *code smell* op en wordt je code stap voor stap overzichtelijker en daardoor beter onderhoudbaar naar de toekomst toe.

En soms ruim je met de ene *code smell* meteen ook (een stukje van) een andere *code smell* op (als je bijv. met de aanpak *replace type code by subclasses* de *code smell primitive obsession* opruimt, ruim je meteen ook *code smell switch-statements* op die plek op). Over het algemeen genomen kun je zeggen dat je bij het opruimen van andere *code smells* meteen ook een stukje *shotgun surgery* opruimt.

Het heeft ook een beetje te maken met de aanpak die je kiest.

Als je waterval gebruikt als methode, kun je tijd reserveren om alle *code smells* (in de gehele applicatie) op te ruimen. Bij waterval plan je zoiets meestal aan het einde van een project in; vlak voordat je het project afrondt en het resultaat oplevert bij een klant.

Bij Scrum gaat dat net iets anders. Daar heb je twee belangrijke momenten waarop je besluit om *code smells* op te ruimen. Bij Scrum kies je er aan het einde van elke sprint bewust voor om een tastbaar resultaat aan de klant op te leveren. Als je een hele sprint zou reserveren voor het opruimen van *code smells* zou de klant geen veranderingen zien. Dat jij dat verschil wel ziet, verandert niets aan het feit dat het opruimen van *code smells* geen nieuwe features oplevert voor de klant. Wanneer werk je dan wel aan het opruimen van *code smells* als je Scrum gebruikt?

- Je zou in je *Definition of Done* op kunnen nemen dat er in de opgeleverde code geen *code smells* voor mogen komen. Jijzelf controleert bij oplevering of dat het geval is en je bent je dus voortdurend bewust van dit aandachtspunt (met andere woorden: je probeert te voorkomen dat er *code smells* in je code komen als je code schrijft). Maar als jij dan toch *code smells* over het hoofd ziet, maak je in GitHub gebruik van een *pull request* die wordt opgepakt door een collega. En ook die collega controleert – voordat jouw *feature branch* wordt opgenomen in de *development branch* – of er in jouw opgeleverde code toch nog *code smells* zijn achtergebleven en accepteert jouw *pull request* alleen als er in zijn of haar ogen geen *code smells* meer voorkomen in de code.
- Maar onder druk wordt alles vloeibaar. Soms sluipen er dus toch *code smells* vanuit jouw *feature branches* naar de *development branch* (en met een beetje pech ook gewoon naar de *master*). Dan is er een ander moment in Scrum waarbij jij *code smells* op kunt ruimen. Voordat je van een *feature* inschat hoeveel tijd je ervoor nodig hebt om die *feature* te realiseren (tijdens bijv. *backlog refinement* of tijdens de *sprint planning*) bekijk je ook de code. Als je tijdens die inspectie constateert dat er toch nog *code smell(s)* zijn achtergebleven in het stukje code dat jij aan moet passen voor de feature, neem je het opruimen van die *code smell(s)* mee bij de inschatting van het werk dat moet gebeuren voor de realisatie van die feature.

En dan nu de cheat sheet (op de volgende pagina.

Cheat Sheet

Code Smell	*Kenmerk van de* code smell	*Aanpak*
Large Class	Je *class* (laten we de class hier Old noemen) heeft veel *properties*. ► Screencast met voorbeeld	Verplaats een aantal bij elkaar horende *properties* van Old naar een nieuwe *class* (laten we die *class* New noemen) en vervang de oude *properties* in Old door één *property* van het type New. Deze aanpak noem je *extract class*.
Long Parameter List	Algemeen kenmerk van deze *code smell* is dat er veel parameters worden meegegeven aan een methode.	Als je de berekening van de waarde van deze parameters ook in de methode zelf kunt doen of als consequent een property als parameter wordt meegestuurd, wordt de bepaling van de waarde van de parameter verplaatst naar de methode (en verdwijn(-t/-en) er dus parameters uit de lijst die niet meer meegegeven hoeven te worden).
	De waarde van een parameter wordt bepaald voordat de methode wordt aangeroepen, terwijl deze waarde ook binnen de methode bepaald kan worden. ► Screencast met voorbeeld	De lokale variabele(n) (waarvan de waarde ook – zonder dat daarvoor weer (een) extra parameter(s) in de methode nodig is/zijn – binnen de methode bepaald kan worden) en de berekening van de waarde(s) word(-t/-en) verplaatst naar methode. Overbodige parameters worden verwijderd. Deze aanpak noem je *replace parameter with method call*.
	Er worden twee of meer parameters meegegeven, maar eigenlijk horen ze bij elkaar (bijv. de cijfers en letters van een postcode of een start- en einddatum). ► Screencast met voorbeeld	Als twee of meer parameters eigenlijk bij elkaar horen, maak je een *class* (bijv. Postcode of DateRange) of maak je gebruik van een bestaand type in Java (bijv. Period). Daarna vervang je de parameters door één parameter van het type van de (nieuwe) *class* Postcode, DateRange of Period). Deze aanpak noem je *introduce parameter object*.

Code Smell	*Kenmerk van de* code smell	*Aanpak*
Switch-Statements	Als je in je code een *switch* (of een gecompliceerde `if-then-else`) hebt gebruikt, dan is vaak gekozen voor een oplossing die ook OO opgelost had kunnen worden.	Hieronder worden twee methodes beschreven waarmee je de *switch* met een *Object Oriented* oplossing kunt vervangen.
	De switch komt voor in een methode van een *class* en wordt gebruikt om onderscheid te maken tussen verschillende types van die *class* (deze *code smell* komt dan ook vaak voor in combinatie met *Primitive Obsession*). ► *Screencast met voorbeeld*	Maak class *abstract* en benoem voor elk type een *subclass*. Maak methode in *class abstract* en implementeer methode in elke *subclass*. Vervang de *switch* nu door één aanroep van methode en gebruik een object van het type *class*, zodat Java op basis van polymorfisme kan bepalen welke geïmplementeerde methode gebruikt moet worden. Deze aanpak noem je *replace conditional with polymorphism*.
	Er worden zoveel parameters aan de methode meegegeven, omdat het eigenlijk gaat om een samenstelling van meerdere methodes. Om die combinatie te kunnen realiseren wordt een `switch-statement` of een `if-then-else` gebruikt. ► *Screencast met voorbeeld*	Splits methode in twee of meer methodes en geef aan elke methode z'n eigen parameter(s) mee. Een voorbeeld: aan methode worden een type en een waarde meegegeven. Afhankelijk van het type worden vervolgens voor type 'hoogte' de hoogte ingesteld en voor type 'breedte' de breedte. Om deze *code smell* op te ruimen, splits je deze methode op in twee methode (setHoogte en setBreedte) en geef je aan elke methode alleen de value mee. Deze aanpak noem je *replace parameter with explicit methods*.

Code Smell	*Kenmerk van de* code smell	*Aanpak*
Primitive Obsession	Er wordt gebruik gemaakt van primitieve typering (bijv. int of String) om onderscheid te maken tussen verschillende types objecten. Dat kan ook *Object Oriented* worden opgelost.	Beide oplossingen hieronder om *primitive obsession* op te ruimen maken (net als *replace conditional with polymorphism*) gebruik van polymorfisme.
	In een *Class* wordt een *property* van een primitief type gebruikt en worden constantes van dat zelfde primitieve type gebruikt om verschillende types van een kenmerk van de class te onderscheiden (bijv. bloedgroep van een persoon). ► *Screencast met voorbeeld*	Voeg een class New toe en voeg de verschillende types (van bijv. bloedgroep) als public statische constante van het type New toe aan New. Neem daarna de volgende stappen: - Voeg een constructor toe aan New, waarmee je zinvolle objecten van het nieuwe type aan kunt maken. - Initialiseer de constantes m.b.v. deze constructor. - Vervang het primitieve type van *property* in Class door het type New en zorg dat dit type in de *constructor* of in een *setter* van Class een waarde krijgt. - Verwijder de primitieve constantes. - Omdat de constantes in New een waarde hebben, kun je parameter een waarde geven met: `parameter = New.CONSTANTE_X.` Deze aanpak noem je *replace type code with class*.

Code Smell	*Kenmerk van de* code smell	*Aanpak*
Primitive Obsession	In een *Class* wordt een *property* van een primitief type gebruikt en worden constantes van dat zelfde type gebruikt om verschillende types van *Class* te onderscheiden (bijv. types klanten). *Replace type code with class* ruimt dus *primitive obsession* voor een kenmerk van een class op en *replace type code with subclasses* ruimt *primitive obsession* voor het type van de class zelf op. ► *Screencast met voorbeeld*	Voeg een *abstract class* ClassType toe en voeg voor elk type een subclass van dit abstracte type toe en implementeer daarin *abstract* methodes uit ClassType. Vervolgens: - Kun je de primitieve constantes verwijderen uit Class. - Kun je het type van *property* wijzigen naar ClassType. - Kun je de *switch* vervangen door één aanroep, waarbij op basis van polymorfisme wordt bepaald welke implementatie van de methode gekozen moet worden. Deze aanpak lijkt dus heel sterk op *replace conditional with polymorphism* (zoals hierboven beschreven). Je noemt deze aanpak in dit geval *replace type code with subclasses*.
Duplicate Code	Algemeen kenmerk is dat een stukje code op één of meer plekken letterlijk wordt herhaald. Gevolg is dat je bij een wijziging van de code die code op twee of meer plekken aan moet passen (ondertussen ben je dat vergeten).	Je kunt deze *code smell* op verschillende manieren opruimen; afhankelijk van het karakter van de *code smell*.
	Een *property* komt in alle *subclasses* voor. ► *Screencast met voorbeeld*	Verplaats de *property* naar de *superclass*. Deze aanpak noem je *pull up field*.
	In alle *subclasses* is (een deel van) de code van de *constructor* identiek. ► *Screencast met voorbeeld*	Verplaats *duplicate code* naar de *constructor* van de *superclass* en roep deze *constructor* in subclasses aan met *super (…)*. Deze aanpak noem je *pull up constructor body*.

Code Smell	*Kenmerk van de* code smell	*Aanpak*
Duplicate Code	In je code wordt steeds de zelfde actie uitgevoerd, maar die actie wordt onder verschillende voorwaarden gebaseerd op verschillende waardes. ► *Screencast met voorbeeld*	Kijk of je deze code kunt vervangen door een eenvoudiger en slimmer algoritme, waarin de actie onder alle voorwaarden wordt uitgevoerd op verschillende waardes (zie voor een mooi voorbeeld van het herhaald opruimen van deze *code smell* de internet-pagina '*Practicing TDD using the Roman Numerals kata*', waar het inderdaad lukt om een slim en eenvoudig algortme te vinden dat voor alle romeinse cijfers werkt). Deze aanpak noem je *substitute algorithm*.
	Een aantal `if`-*s* (en mogelijk `else`-*s*) leiden allemaal tot dezelfde (*duplicate*) code. ► *Screencast met voorbeeld*	Verplaats de voorwaarden die tot hetzelfde resultaat leiden allemaal naar een *private boolean* methode en roep deze methode aan in plaats van de verschillende `if`-*s* (en `else`-*s*). Deze aanpak noem je *consolidate conditional expression*.
	In alle `then`-*s* komt dezelfde code telkens terug (niet als geheel, maar als onderdeel van de `then`-code). ► *Screencast met voorbeeld*	Verwijder de *duplicate code* uit de de `if-then-else`, en neem de code eenmalig op na het afsluiten van de `if-then-else`. Deze aanpak noem je *consolidate duplicate conditional fragments*.
Long Method	Algemeen probleem van een lange method is dat je moet scrollen om te checken wat de methode precies doet.	Je kunt deze code smell op verschillende manieren opruimen; afhankelijk van het karakter van de code smell.
	Een aantal regels code horen bij elkaar, omdat ze een stukje functionaliteit realiseren. ► *Screencast met voorbeeld*	Pak deze regels code op, plaats ze in een private methode in dezelfde class en geef deze method een sprekende naam. Controleer welke variabelen door de verplaatsing rood gekleurd zijn (ze zijn binnen de methode onbekend) en vervang deze variabele door: - Een parameter die aan de methode wordt meegegeven of - Een lokale variabele die alleen binnen de methode gebruikt wordt. Roep vervolgens op de plek waar je de regels code zojuist verwijderd hebt de nieuwe methode aan en geef de parameters de juiste waarde. Deze aanpak noem je *extract method*.

Code Smell	*Kenmerk van de* code smell	*Aanpak*
Long Method	In je code gebruik je een lokale variabele, waarvan je de waarde berekent (vaak zijn daar meer regels code voor nodig). Het resultaat van die berekening geef je mee aan een methode. ► *Screencast met voorbeeld*	Je kunt de berekening van de lokale variabele uitbesteden aan een private methode. Op de plekken waar je in je oude code gebruik maakte van de waarde van deze lokale variabele, kun je nu de methode aanroepen waaraan je de berekening van de lokale variabele hebt uitbesteed (en die je een sprekende naam hebt gegeven waaruit blijkt wat die methode berekent). Deze aanpak noem je *replace temp with query*.
	Voordat je een methode aanroept, verzamel en verwerk je de gegevens van een object in een (aantal) lokale variabele(n). De waarde van die lokale variabelen (waarin je de gegevens van een object hebt verzameld) geef je vervolgens als aparte parameters mee aan een methode. ► *Screencast met voorbeeld*	Verwijder de vertaling van de gegevens uit een object in één of meer lokale variabelen uit je code voor de methode-aanroep. Geef het object vervolgens als parameter mee aan de methode. Verzamel en verwerk de gegevens van het object tenslotte in de methode die je hebt aangeroepen. Deze aanpak noem je *preserve whole object*.

Code Smell	*Kenmerk van de* code smell	*Aanpak*
Long Method	In de code van een class (laten we die class hier Origineel noemen) staat een gecompliceerde berekening die een stuk van je methode in beslag neemt. ► *Screencast met voorbeeld*	Verplaats de code voor de berekening vanuit de huidige class naar een nieuwe class (laten we die class hier Berekening noemen). Doorloop daarvoor de volgende stappen: - Voeg de nieuwe class Berekening toe waaraan je de bestaande berekening uit Origineel uitbesteedt. - Kopieer de properties die je nodig hebt voor de berekening vanuit Origineel naar Berekening. - Voeg een constructor toe aan Berekening, waaraan je een parameter van het type Origineel meegeeft, waarin je waarde van de properties uit Origineel kopieert naar de properties van Berekening. - Voeg een methode (die we hier berekening noemen) toe aan Berekening en kopieer hierin de code voor de berekening. - Op de plek waar de berekening in Origineel werd uitgevoerd, vervang je de code voor de berekening door het aanmaken van een object van het type Berekening waaraan je het object this (van het type Origineel) meegeeft. - Roep daarna de methode berekening aan. Deze aanpak noem je *replace method with method object*.
	In je code staat een `if-then-else` of een `switch` met een gecompliceerde voorwaarde. ► *Screencast met voorbeeld*	Verplaats de voorwaarde naar een private methode met een boolean return-type en vervang de complexe voorwaarde in de orinele code door de aanroep van de private methode. Deze aanpak noem je *decompose conditional*.

Code Smell	*Kenmerk van de* code smell	*Aanpak*
Temporary Field	Algemeen kenmerk van deze code smell is dat een property wordt gebruikt die maar in één methode wordt gebruikt. Het is dus eigenlijk geen property, maar een tijdelijke (lokale) variabele.	Het resultaat van onderstaande stappenplannetjes is bijna hetzelfde: je verplaatst de 'tijdelijke' property naar een aparte class en de methode gaat mee.
	Zie hierboven. De aanpak hiernaast kies je als je in methode alleen gebruik maakt van property. ► *Screencast met voorbeeld*	De 'tijdelijke' property wordt gekopieerd naar een class New. Daarna: - Wordt property in Old vervangen door een property van type New. - Wordt methode uit Old gekopieerd naar New. - Wordt de code van methode in Old vervangen door de aanroep van de gekopieerde methode in New. Deze aanpak gebruik je ook voor het opruimen van de code smell *Large Class* en wordt *extract class* genoemd.
	Zie hierboven. De aanpak hiernaast kies je als je in methode niet alleen gebruik maakt van property, maar ook van de waarde van andere properties. Ook de waarde van deze properties moet dan naar class New worden gekopieerd. ► *Screencast met voorbeeld*	De aanpak voor het opruimen van deze code smell hebben we al gebruikt voor het opruimen van code smell *Long Method*; specifiek de aanpak die daar *replace method with method object* wordt genoemd. Voor het stappenplan voor deze aanpak voor het opruimen van *Temporary Field* wordt verwezen naar het stappenplan dat hierboven is uitgewerkt.
Divergent Change	Er is teveel code per class verzameld. Vaak ontstaat daardoor ook nog eens *Duplicate Code* in verschillende *subclasses*. Het probleem is dat door één wijziging in de class op heel veel andere plekken in zo'n class code hersteld moet worden als gevolg van een kleine wijziging.	Voeg een *superclass* toe, waarin je gemeenschappelijke *properties* en methodes onderbrengt die je slim hebt gekopieerd uit verschillende *subclasses*. Deze aanpak noem je *extract superclass*.

Code Smell	*Kenmerk van de* code smell	*Aanpak*
Shotgun Surgery	Algemeen kenmerk van deze *code smell* is dat een kleine wijziging in je code leidt tot veel onverwachte wijzigingen in andere *Classes*.	Voor deze *code smell* zijn, afhankelijk van het type probleem verschillende methodes beschikbaar om de *code smell* op te ruimen.
	Een methode wordt meer gebruikt in een andere *Class* Other dan in de eigen class Self.	Verplaats de methode van Self naar Other. Deze aanpak noem je *move method*.
	Een *property* of variabele wordt meer gebruikt in een andere *Class* Other dan in de eigen class Self.	Verplaats de *property* van Self naar Other. Deze aanpak noem je *move field*.
	Een *Class* Eenvoudig heeft geen verantwoordelijkheden (en dat lijkt in de toekomst niet te veranderen), maar wordt wel gebruikt in een andere *Class* Other (en niet in andere *Classes*).	Neem Eenvoudig als *inline Class* (ofwel lokale *Class*) op in Other. Deze aanpak noem je *inline class*.