TOOLS

Iedereen maakt fouten. Foutloze code is een illusie. Enige discipline, codereviews en goede tooling kunnen helpen het aantal fouten drastisch te beperken. Voor meer houvast geef ik een beeld van een aantal belangrijke code inspection tools, en geef ik de – in mijn ogen – beste keuzes, en de situaties waarin wat het best kan worden ingezet.

Schrijven van foutloze code is een illusie

Overzicht van Code Inspection Tools

Er bestaan ontzettend veel tools die qua functionaliteit (deels) vergelijkbaar zijn met de door ons genoemde. De selectie die wij hebben gemaakt is gebaseerd op jarenlange praktijkervaring met en tests van kwaliteitgerelateerde tooling. Een snelle zoekactie met behulp van Google (op Java code quality tools of Java code inspection tools), of een bezoek aan Open Source Code Analyzers in Java brengt je redelijk snel up-to-date met deze en andere tools.
Om het overzichtelijk te houden, geef ik alleen een overzicht van de aanbevolen tools, niet alle geteste tools.

Codekwaliteit

De redenen om code van goede kwaliteit af te leveren zijn legio – denk maar aan je eigen gemopper als je onderhoud moet doen op 'slechte' code. Nog erger dan slechte code zijn programmeerfouten die tot productiefouten kunnen leiden – runtime fouten, maar ook security breaches.
Code van goede kwaliteit is eenvoudiger te onderhouden en daarmee ook goedkoper qua onderhoud.

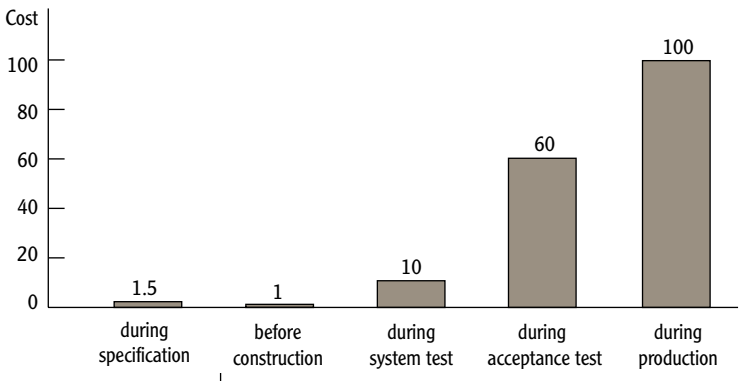
Een codereview, uitgevoerd door ervaren ontwikkelaars is een goed begin. Het voordeel hiervan is dat de gereviewde er veel van kan leren, omdat hij kan uitleggen waarom bepaalde dingen niet kunnen, of hoe ze beter kunnen.

Natuurlijk zijn codereviews en het gebruik van de beschreven tools alleen niet genoeg. Er zullen echt goede unit, functionele, performance en stress testen nodig zijn. Dit valt echter buiten de scope van dit artikel.

Meten van codekwaliteit

Het uitvoeren van codereviews kan voor een deel worden vervangen door het gebruik van tools. Deze kunnen enerzijds een beeld geven van codeerfouten, anderzijds van (een beperkt aantal) mogelijke runtime fouten. Hoe minder fouten en hoe eerder in een project ze gevonden worden, hoe beter het uiteindelijke resultaat, en hoe minder tijd het oplossen van de fouten kost (zie figuur 1).

Een groot deel van de codeerfouten kan worden gevonden door middel van 'statische code-analyse'. Hierbij wordt alleen de code bekeken; het programma draait niet. Denk hierbij aan:



Figuur 1: overzicht van de kosten van het oplossen van een fout in de verschillende stadia van een project.

Mylène Reiners
is software-architect bij Atos Origin.

Een goede mix van code inspection tools helpt je om fouten eenvoudiger op te sporen.

- voldoet de code aan de eisen van de code conventies (formattering, CamelCase)?
- is er geen duplicaat code?
- worden er geen veel voorkomende fouten (initialisatie fouten, delen door 0, onbereikbare code) gemaakt?
- is de code niet te complex (diepte nesting, grootte classes en methods)?
- liggen er geen performance problemen op de loer(variabelen die nooit gebruikt worden, serializable op een té hoog niveau)?
- is de code goed gedocumenteerd (Javadoc)?

Een deel van de runtime fouten kan worden gevonden via 'dynamische code-analyse'. Bij dynamische code-analyse wordt het programma uitgevoerd. Denk hierbij aan (de omschrijvingen zijn in het Engels, omdat er geen echt goede Nederlandse equivalenten voor zijn):

- Memory Leaks;
- Invalid Pointers;
- Memory Corruption;
- Memory Overflow;
- Reading/Writing Uninitialized Memory;
- Unused Variables/Arguments;
- Data Formatting Problems;
- Unexpected Errors;
- Invalid Arguments;
- Invalid System Calls.

Een goede mix van code inspection tools helpt je om veel van de hierboven genoemde fouten op te sporen. Belangrijk in dit geval is te onthouden dat tools niet altijd de correcte resultaten melden. De mogelijkheid bestaat dat er valse positieven (foutmeldingen bij correcte code) of valse negatieven (geen foutmeldingen bij incorrecte code) worden gegeven. Bij de keuze van de tools is hier goed naar gekeken. De valse positieven en negatieven komen niet onacceptabel vaak voor, maar ze komen voor. Blijf dus altijd kritisch!
Om een goede mix van tools te vinden, zou idealiter zowel statische als dynamische code-analyse plaats moeten vinden. Goede dynamische code-analyse-tools zijn echter schaars, zeker in de Open Source-wereld. Dan kom je al snel op bytecode manipulatie en dan is de analyse maar een bijproduct. In de onderstaande lijst zul je daarom helaas maar één (proprietary) product tegenkomen.

Daarnaast is er nog een aantal (bijzonder) 'nice to haves'. In dat opzicht moet je denken aan:

- Suggesties voor oplossing en/of correctie;
- Uitbreidbaarheid, herbruikbaarheid, onderhoudbaarheid, efficiëntie;
- Configureerbaarheid;
- (Grafische) rapportage;
- Integratie met andere tools, IDE e.d;
- Snelheid;
- Goede documentatie.

Een overzicht van de tools

Alle beschreven tools voldoen aan zoveel mogelijk van de hierboven beschreven eisen en onderscheiden zich van de niet gekozen tools vooral in het voldoen aan de 'nice to haves'. Deze maken het leven van een programmeur zoveel gemakkelijker…

Checkstyle

Checkstyle is een Open Source, statisch code-analysetool om verschillende aspecten van de code te controleren op lay-out, class design problemen, dubbele code en (een beperkt aantal) bug patterns.
Checkstyle maakt gebruik van een configuratiefile. Er zijn kant en klare configuratiefiles die je kunt gebruiken, bijvoorbeeld de 'Code Conventions for the Java Programming Language' (sun_checks.xml). Je kunt ook je eigen configuratiefiles maken waarmee voor jouw project belangrijke controles worden uitgevoerd.
Het is een goede gewoonte om Checkstyle meteen op je code los te laten en de problemen zo snel mogelijk op te lossen. Allereerst werkt dit het beste en het snelste, en daarnaast leer je welke fouten je in de rest van je code meteen kunt vermijden – en misschien merk je dat voor jouw project bepaalde controles niet relevant zijn, zodat je die meteen kunt uitschakelen, en daardoor niet meer gestoord wordt (zorg wel dat je heel zeker weet dat dit kan).

PMD

PMD is een Open Source, statisch code-analysetool om Java sourcecode te analyseren op lay-out en om mogelijke problemen te vinden zoals bugs, dead code, suboptimale code, te complexe code en dubbele code. Dit laatste wordt uitgevoerd door de CPD (Copy/Paste Detector) module Er is een gedeeltelijke overlap met Checkstyle, maar beide hebben ook zoveel eigens, dat ze elkaar absoluut niet uitsluiten, eerder aanvullen. Kort gezegd focust PMD meer op mogelijke defects. Net als Checkstyle verdient het aanbeveling om voor alle geschreven code meteen de PMD checks uit te voeren.
PMD maakt gebruik van verschillende rulesets (via kant-en-klare configuratiefiles) om de code te analyseren. Op het moment zijn er bijna dertig verschillende rulesets. Het gebruik van PMD valt en staat met het kiezen van de juiste, afhankelijk van je project. De 'Basic' ruleset biedt een goed startpunt, maar controleer ook de overige, of je die misschien kunt (moet?) toepassen. Ook kunnen eigen rules gedefinieerd worden.

Simian

Simian (Similarity Analyser) controleert code op codeduplicatie, onafhankelijk van de programmeertaal, met als doel het voorkomen van copy en paste fouten. Deze fouten kunnen vervelende effecten hebben voor het onderhoud van de code. Bij

aanpassingen in de code wordt vaak de duplicate code over het hoofd gezien.
Simian is een alternatief voor de CPD module van PMD, en is, in tegenstelling tot PMD, niet gratis voor projecten of enterprise toepassingen.

FindBugs
FindBugs is een Open Source, statische code-analysetool en vindt veelvoorkomende en potentiële fouten in Java-code. De fouten die FindBugs zoekt, zijn gebaseerd op keuzes van de ontwikkelaar en constructie van zijn/haar code. Denk aan: null-pointer verwijzingen, redundante vergelijkingen, onjuiste implementatie van equals() methode en code die performance issues kan veroorzaken. FindBugs analyseert de Java bytecode (.class files) i.p.v. de sourcecode.
Voor FindBugs geldt hetzelfde als voor de twee hiervoor beschreven tools: je kunt je eigen 'bug detectors' schrijven en draai het vanaf het begin van je project op je code.

Fortify SCA en PTA.
Fortify biedt twee tools, waarvan de eerste, Source Code Analyzer (SCA) als alternatief voor Checkstyle, PMD en FindBugs gebruikt kan worden. Fortify is sterk in het voorkomen van bugs. Ook hiervoor kun je je eigen checks schrijven.
Als geld geen rol speelt, is dit een aanrader, zeker in combinatie met de tweede: Program Trace Analyzer (PTA), die dynamische code-analyse doet en gewoonlijk wordt ingezet tijdens een Quality Assurance test, en zo tijdens het testen nog extra informatie geeft over (mogelijk toekomstige) fouten.
Als je deelneemt aan een Open Source-project, kun je trouwens je code door Fortify laten controleren op fouten (met SCA). Je krijgt dan een gedetailleerd overzicht van de fouten in je code en een korte melding wordt gemaakt op hun site.

JDepend en Metrics.
Metrieken kunnen erg veel zeggen over de kwaliteit van je project. Immers met het kwantificeren van de mate van uitbreidbaarheid, herbruikbaarheid en onderhoudbaarheid kun je een erg goed beeld van de kwaliteit van de code (en het ontwerp) krijgen.
JDepend is een Open Source-tool dat de relatie tussen Java-packages weergeeft door de .class files binnen deze packages te analyseren. Op basis van deze analyse worden er design kwaliteit matrices gegenereerd die iets vertellen over de kwaliteit van de architectuur zoals uitbreidbaarheid, herbruikbaarheid en onderhoudbaarheid. Binnen de matrix worden per package bijvoorbeeld het aantal abstracte en concrete classes en interfaces, het aantal packages dat afhankelijk is van het betreffende package etcetera vermeld.

Ook Metrics is een tool om statistieken voor een project te verzamelen. Onder andere de volgende eigenschappen worden inzichtelijk gemaakt: Aantal classes, aantal subclasses, diepte in overerving, LoC (Lines of Code), complexiteit en cohesie binnen een class. Naast deze statistiek kan Metrics ook een grafisch overzicht van de afhankelijkheden die packages met elkaar hebben genereren.

Voor de meeste projecten volstaat het om een keuze te maken tussen een van beide tools, omdat ze elkaar voor een groot deel overlappen. Het grootste verschil zit in de weergave en het aantal metingen dat gedaan wordt. JDepend geeft de informatie per package, Metrics per metriek. Daarnaast doet Metrics meer metingen. De ervaring leert dat het goed gebruiken van dit soort tools een redelijk steile leercurve heeft. Voor projecten waar de metrieken heel erg belangrijk zijn, raad ik Metrics aan, in andere gevallen JDepend (omdat die leercurve minder is door het minder aantal metrieken). Alleen echt heel bedrijfskritische applicaties zouden voor alle zekerheid door beiden heen gehaald kunnen worden.

JBoss Tattletale
JBoss Tattletale is een Open Source-tool, die kan helpen bij het inventariseren van problemen die aan het licht kunnen komen in een runtime omgeving zoals ontbrekende of duplicate classes. Het rapporteert onder meer de volgende eigenschappen:
- Afhankelijkheden tussen jars;
- Ontbrekende classes op het classpath;
- Identificeert wanneer een class in meerder jars zit;
- Identificeert wanneer een jar meerder malen binnen een project voorkomt.

Zeker classes die in meerdere jars voorkomen, kunnen in de praktijk tot problemen leiden, die vaak moeilijk te traceren zijn. Dit geldt zeker als bijvoorbeeld de Application Server al een jar gebruikt, die jij ook nodig hebt, alleen met een andere versie. Daarom is het een goed idee om zodra er code gedeployed gaat worden, af en toe JBoss Tattletale te gebruiken.

Ook al is Tattletale een product van JBoss, dat wil niet zeggen dat het alleen in een JBoss-omgeving nuttig is of te gebruiken is. Het kan gebruikt worden in alle omgevingen, dus ook IBM, Oracle (BEA) of Open Source, en dat wordt dan ook geadviseerd!

Sonar
Sonar is een software quality managementtool. De resultaten worden in een webbrowser getoond.
De tool analyseert (naar wens continu) de code en genereert matrices met informatie over de kwaliteit van de code. Denk aan: testcoverage, complexiteit

Een groot voordeel van Sonar is dat een aantal bewezen code inspection tools worden gecombineerd.

Foutloze code kun je niet garanderen, wel betere.

per methode/class, code rule compliance ofwel: in hoeverre respecteert de code de markt standaarden, dubbele code en buildtime. Sonar biedt een algemene matrix (dashboard) met een grafische samenvatting van de belangrijkste informatie. Per project kan dieper de matrix ingezoomd worden voor gedetailleerde informatie.
Een groot voordeel van Sonar is dat verschillende goed bewezen code inspectie tools gecombineerd worden om een totaalplaatje te creëren. Doordat Sonar de code in de continuous build omgeving kan verifiëren, kan op deze manier de kwaliteit van het project in zijn geheel gecontroleerd worden.

Sonar maakt gebruik van een aantal eerder beschreven tools, zoals Checkstyle, en PMD, maar ook JavaNCSS (ook een goed metrics tool, maar iets minder uitgebreid dan JDepend en Metrics) en Cobertura (Code Coverage tool). Een voorbeeld van de eerste detailpagina van Sonar wordt gegeven in Figuur 2.

Figuur 2: Sonar output.

Er is ook een aantal plugins voor Sonar geschreven. Die plugins variëren van niet-standaard ondersteunde kwaliteitstools tot integratie met Continuous Integration en issue tracking tools. Vanzelfsprekend kun je naar believen je eigen plugins schrijven.

Indien gewenst kan er ook voor een commerciële variant van Sonar worden gekozen. In dat geval krijg je ondersteuning bij installatie en gebruik en commerciële plugins. Die laatste zijn vercommercialiseerde versies van de Open Source plugins.

Conclusies en aanbevelingen

Zoals in de inleiding gesteld is het niet eenvoudig om de juiste verzameling van tools te selecteren. Voor elke situatie kan afhankelijk van de wensen en de omgeving de selectie anders uitpakken. De grootte van een project en de mate waarin de betrokkenen contact met elkaar hebben, zal deze keuze sterk beïnvloeden.
Dit geldt in de eerste plaats voor de mogelijkheid van het houden van face-to-face codereviews. Dit is en blijft de beste manier om code van goede kwaliteit af te leveren. Het is echter niet zo, dat codereviews code inspectie tooling overbodig maken. Goed gebruik van de juiste tools maakt de review immers gemakkelijker, omdat al een groot aantal mogelijke fouten zijn afgevangen.

Allereerst adviseer ik iedereen in elk project Sonar te gebruiken. Het is een gemakkelijk te installeren en te gebruiken tool, waarmee je heel snel een overzicht krijgt van de algemene kwaliteit van de projecten.

Ook al gebruikt Sonar onder water een aantal beschreven code inspectie tools, toch zou ik het niet als enige gebruiken.

Tijdens het coderen adviseer ik in alle gevallen ook Checkstyle, PMD en FindBugs (of SCA) gebruiken, omdat ze gemakkelijk te integreren zijn in een standaardwerkwijze, en de engineer per sourcecodebestand zijn of haar fouten meteen kan zien en aanpassen. Sonar is mijns inziens daarvoor té zeer gericht op de overallkwaliteit.
Daarnaast adviseer ik ten stelligste om regelmatig, in elk geval elke oplevering te inspecteren met JBoss Tattletale. Veel voorkomende fouten als redundante libraries of classes kunnen zo voorkomen worden.

Sonar geeft, zoals eerder gezegd, ook metrieken weer. Omdat het leren omgaan met de daaraan gelieerde tools nogal een steile leercurve heeft, is het vaak moeilijk om hiervoor tijd te krijgen. Aan te bevelen valt om de cijfers in Sonar te blijven controleren, en als ze té hoog worden, JDepend in te zetten.
Wanneer metrieken echt belangrijk zijn, kan Metrics ingezet worden, en eventueel een combinatie van beide bij superbelangrijke business critical projecten.

Tenslotte kan PTA ingezet worden tijdens Quality Assurance (acceptatie) testen. Hiervoor dient echter budget vrijgemaakt te worden.
Met al deze aanbevelingen is foutloze code nog steeds niet gegarandeerd, maar in elk geval betere code – en dat is heel wat waard. «

Referenties

- Open Source Code Analyzers in Java: http://java-source.net/open-source/code-analyzers
- Jupiter: http://code.google.com/p/jupiter-eclipse-plugin/
- Crucible: http://www.atlassian.com/software/crucible/
- Checkstyle: http://checkstyle.sourceforge.net/
- Code Conventions for the Java Programming Language: http://java.sun.com/docs/codeconv/
- PMD: http://pmd.sourceforge.net/
- Simian: http://www.redhillconsulting.com.au/products/simian/
- FindBugs: http://findbugs.sourceforge.net/
- Fortify: http://www.fortify.com/
- Fortify SCA results voor Open Source projecten: https://opensource.fortify.com/teamserver/welcome.fhtml
- JDepend: http://www.clarkware.com/software/JDepend.html
- Metrics: http://metrics.sourceforge.net/
- JBoss Tattletale: http://www.jboss.org/tattletale
- Sonar: http://sonar.codehaus.org/