

Een kritische blik op Cyclomatic Complexity

S.J.M. Slagter, Universiteit van Amsterdam, 2009

Abstract

Met het toenemende gebruik van analyse tools en metrieken speelt het bepalen van de complexiteit van software een steeds belangrijkere rol. Hierbij wordt veelal gebruik gemaakt van de McCabe's Cyclomatic Complexity. In dit essay bespreken we de manier waarop dit complexiteitscijfer tot stand komt en in hoeverre dit een juist beeld geeft van de complexiteit van software.

Keywords: cyclomatic complexity, cc, McCabe, metriek, metric, complexiteit, complexity

Inleiding

Het meten van complexiteit in software is een lastig begrip[5]. Vraag een aantal ontwikkelaars om aan te geven wat software complex maakt en je krijg waarschijnlijk uiteenlopende antwoorden. Om het wel of niet complex zijn van software niet af te laten hangen van een persoonlijke mening, zijn er ook een aantal wetenschappelijk methodes die de complexiteit van software in kaart brengen. Een van deze methodes is Cyclomatic Complexity van Thomas McCabe uit 1976[1].

Cyclomatic Complexity is een veelgebruikt cijfer voor het aanduiden van de complexiteit van software[10] en wordt onder andere gebruikt tijdens de ontwikkelfase, voor testen, voor refactoring en rewriting van bestaande code, tijdens code reviews en voor maintenance doeleinden[5]. Aangezien vooral maintenance een steeds belangrijkere rol speelt in de software lifecycle (zie tabel 1.) en 50% van de kosten voor maintenance gaat zitten in het analyseren van de software[9] is het belangrijk dat de gegevens volledig en correct zijn.

| Jaar | Maintenance kosten |
|------|--------------------|
| 2000 | > 90% |
| 1993 | 75% |
| 1990 | 60 - 70% |
| 1988 | 60 - 70% |
| 1984 | 65-75% |
| 1981 | > 50 % |
| 1979 | 67% |

Tabel 1: Aandeel software maintenance kosten t.o.v. totaal gemaakte kosten in software lifecycle[8]

In dit essay gaan we dieper in op Cyclomatic Complexity en werpen we een kritische blik op de manier waarop dit cijfer tot stand komt en in hoeverre dit nu altijd een correct beeld geeft van de complexiteit van software zodat er gedegen conclusies aan dit cijfers kunnen worden onttrokken. Aan de hand van de voor- en nadelen van Cyclomatic Complexity en een aantal voorbeelden zal worden aangetoond dat complexiteit lang niet zo triviaal kan worden aangeduid als dat Cyclomatic Complexity wellicht doet vermoeden.

1. Complexiteit

Wat is nu eigenlijk complexiteit? Allereerst is er een belangrijk verschil tussen design complexiteit en code complexiteit. Design complexiteit gaat bijvoorbeeld vooral over het aantal modules in de software en hoe deze met elkaar zijn verbonden (coupling) of over de fan-in of fan-out van een module. Code complexiteit gaat meer over de kwaliteit van de onderliggende code[11]. Bevat de code bijvoorbeeld veel nesting en/of variabelen dan wordt de code al snel complexer bevonden dan wanneer dit niet zo is[11]. Dit is echter vaak een subjectief beeld wat ervoor zorgt dat code complexiteit een moeilijk definieerbare vorm van complexiteit is. Immers, code wat voor de ene ontwikkelaar goed is te begrijpen, kan voor een andere ontwikkelaar wél erg complex zijn. Dit kan bijvoorbeeld te maken hebben met ervaring in het domein of in de betreffende ontwikkelomgeving.

Het is namelijk bekend dat de meest complexe delen van software veel business logica bevatten [18] en daarom voor ontwikkelaars die hier onbekend mee zijn, sneller complex worden bevonden.

Complexiteit hangt in ieder geval niet altijd af van de omvang van een klasse of methode (simpele code kan immers ook erg lang zijn) maar omvang kan wel een belangrijke indicatie zijn (zie hoofdstuk 4). Echte complexiteit hangt echter meer af van de afhankelijkheden binnen een bepaald stuk code. Onderzoek heeft uitgewezen dat het menselijk brein gemiddeld zo'n 7 dingen tegelijk kan onthouden met een marge van 2 [17]. Is de code dus zodanig opgebouwd dat de ontwikkelaar meer dan 7 dingen (variabelen, code paths, logische beslissing, etc.) moet onthouden dan neemt de kans op fouten toe[5].

Om complexiteit niet af te laten hangen van een subjectieve mening van bijvoorbeeld een ontwikkelaar of code reviewer is het nodig om de complexiteit op een objectieve manier te meten. Een van deze complexiteitsmetrieken is de Cyclomatic Complexity van Thomas McCabe. Hij bedacht een manier om de complexiteit van code in kaart te brengen op basis van het aantal mogelijke executiepaden door de code. Om aan de hand van Cyclomatic Complexity te bepalen of een bepaalde routine of methode complex is, wordt over het algemeen de volgende indeling aangehouden:

| CC | Complexiteit |
|--------|-------------------|
| 0 - 5 | Weinig complex |
| 6 - 10 | Gemiddeld complex |
| > 10 | Erg complex |

Tabel 2: Indeling Cyclomatic Complexity[6]

In deze tabel is te zien dat deze complexiteitsindeling in overeenstemming is met de capaciteit van het menselijk brein. Deze indeling is echter wel alleen van toepassing voor een routine of methode terwijl Cyclomatic Complexity ook wel wordt toegepast op klassen, modules of hele systemen. Uiteraard ligt de Cyclomatic Complexity in dat geval een stuk hoger en worden er andere indelingen gehandhaafd.

2. McCabe's cyclomatic complexity

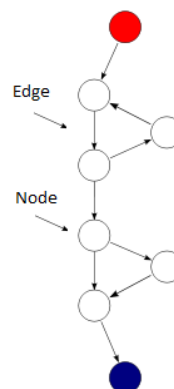
Zoals gezegd is Cyclomatic Complexity een manier om de complexiteit van code te bepalen. Dit wordt gedaan aan de hand van een formule dat het aantal verschillende executiepaden door dat stuk code berekent. Om dit te kunnen doen wordt er gebruik gemaakt van de control flow van de code waarbij de edges(E), nodes(N) en verbonden componenten (P) in kaart worden gebracht doormiddel van een zogenaamde control flow graph (zie afbeelding 1.) [1,13].

```
private int Sum(List<int> numberList){
    int result = 0;

    foreach (int n in numberList){
        result += n;
    }

    if (result > int.MaxValue){
        result = 0;
    }

    return result;
}
```



Afbeelding 1: Code voorbeeld met control flow graph.

In bovenstaande afbeelding zien we een stukje voorbeeld code met daarnaast de control flow graph zoals deze kan worden getekend aan de hand van dit voorbeeld. Deze graph heeft 9 edges(E), 8 nodes (N) en 1 connected component (P), in dit geval de graph zelf. Om hiervan de Cyclomatic Complexity te berekenen gebruiken we de formule van McCabe: $CC = E - N + 2P$, oftewel $CC = 9 - 8 + 2 = 3$. De Cyclomatic Complexity van dit stukje voorbeeldcode komt dus uit op 3. Kijkend naar de code zelf kunnen we dit ook handmatig berekenen door 1 te nemen voor de methode zelf, 1 voor de foreach loop en 1 voor het if-statement.

De berekening van de Cyclomatic Complexity is dus gebaseerd op het tellen van ieder conditionele statement plus 1 voor de methode zelf. Zie onderstaande tabel voor een overzicht van de conditionele statements waar McCabe's Cyclomatic Complexity rekening mee houdt:

| Constructie: | Effect: | Reden: |
|-----------------|-----------|---|
| If - then / ? | +1 | Een (verkort) if-statement is een beslissing |
| Elseif-then | +1 | Elseif is een nieuwe beslissing |
| Else | 0 | Else voegt geen nieuwe beslissing toe, deze zit bij het if-statement. |
| Select-Case | +1 / case | Elke case introduceert een nieuwe beslissing |
| Case-else | 0 | Else introduceert geen nieuwe beslissing |
| For[each] -next | +1 | Er wordt bij het starten van de loop een beslissing gemaakt |
| Do-loop | +1 | Er is een beslissing voor het uitvoeren van de do-loop |
| While | +1 | Er is een beslissing voor het uitvoeren van de while |

Tabel 3: Conditionele statements voor cyclomatic complexity[1,15,18]

Opvallend aan dit lijstje is dat er geen rekening wordt gehouden met mogelijke variaties in de executiepaden door het gebruik van eventuele logische operatoren zoals and en or. Omdat Cyclomatic Complexity is van origine bedoeld voor een procedurele programmeertaal ontbreekt ook een ander belangrijk concept voor het meten van (moderne) object georiënteerde software, namelijk de afhandeling van excepties doormiddel van try-catch. Hiervoor is een variatie op het originele Cyclomatic Complexity cijfer bedacht. De zogenaamde CCN2. Dit is tegenwoordig dan ook de meest gebruikte variant voor het meten van de Cyclomatic Complexity[18]. Zie onderstaande tabel voor de aanvulling van CCN2 op de originele Cyclomatic Complexity van Thomas McCabe.

| Constructie: | Effect: | Reden: |
|--------------|---------|---|
| And / && | +1 | Iedere and levert een nieuwe variatie op |
| Or / | +1 | Iedere or levert een nieuwe variatie op |
| Xor | +1 | Iedere xor levert een nieuwe variatie op |
| Catch | +1 | Iedere catch brengt een nieuwe beslissing met zich mee; "als deze fout voorkomt, doe dan dit" |

Tabel 4: Aanvulling conditionele statements voor cyclomatic complexity[15,18]

Cyclomatic Complexity, uitgaande van de 'moderne' variant, heeft een aantal voor- en nadelen ten opzichte van andere methodes zoals de Halstead metrieken[24]. De volgende hoofdstukken gaan hier verder op in.

3. Voordelen van mccabe

Om te beginnen is Cyclomatic Complexity een simpel cijfer waar veel mensen zich in kunnen vinden. Er komen geen hele ingewikkelde berekeningen aan te pas en het geeft direct inzicht in complexiteit van software [5]. Aan de hand hiervan kan goed overleg worden gepleegd en worden bepaald waar en hoe eventueel de code moet worden aangepast om de complexiteit terug te dringen.

Daarnaast is het met Cyclomatic Complexity mogelijk om eerder naar complexiteit te kijken dan wanneer het project is afgerond zoals dit bijvoorbeeld bij de Halstead metrieken het geval is [11]. Dit voordeel zorgt ervoor dat er bijvoorbeeld al tijdens de ontwikkelfase rekening gehouden kan worden met complexiteit en zodat er op dat punt al veranderingen kunnen worden doorgevoerd om de software zo eenvoudig mogelijk te houden. Dit zou bijvoorbeeld kunnen door het opsplitsen van code.

Een ander voordeel is dat Cyclomatic Complexity een indicatie geeft van de gebieden waar het meeste aandacht voor testen moet plaatsvinden. In sommige gevallen wordt het cijfers zelf gebruikt als indicatie voor het aantal testcases dat voor een bepaalde methode moet worden geschreven [10].

4. Nadelen van de mccabe

De eenvoudigheid van Cyclomatic Complexity is wellicht ook wel meteen het grootste nadeel ervan. Doordat McCabe's Cyclomatic Complexity zich alleen richt op het aantal mogelijke executiepaden binnen een routine of methode zegt het ook niet veel meer dan dat[1,6,11].

Zo wordt er bijvoorbeeld geen rekening gehouden met de Lines Of Code (LOC), data complexity of nesting. Laten we als voorbeeld eens naar de volgende code kijken:

```
private void Export(string fileType, string data){
    if (fileType == "csv")
        ExportCSV(data);

    if (fileType == "doc")
        ExportDoc(data);

    if (fileType == "xml")
        ExportXml(data);
}
```

Afbeelding 2: Code voorbeeld, simpel gebruik van if-statements

Dit stukje code heeft een Cyclomatic Complexity van 4 en is dus volgens McCabe geen complexe code[1]. In dit geval zullen de meeste ontwikkelaars daar ook in meegaan en geen problemen hebben met het aanpassen van deze code. Echter, laten we nu eens naar een tweetal andere stukjes code kijken:

```
private void Export(File file, string data){
    if (file.Exists){
        if (file.Extension == ".xml"){
            if (data.Length > 0) { ExportXml(file, data); }
        }
    }
}
```

Afbeelding 3: Code voorbeeld, gebruik van geneste if-statements

```
private bool Export(File file, string data){
    bool result = false;
    if (file.Exists){
        result = (data.Length > 0) ? ExportCSV(data) : false;
    }
    return result;
}
```

Afbeelding 4: Code voorbeeld, gebruik van verkorte if-statements

Deze, nog steeds redelijk eenvoudige stukjes code hebben ook een Cyclomatic Complexity van 4 maar zijn door het gebruik van nesting en verkorte if-statements al direct een stuk ingewikkelder geworden. Ook al betreft het hier natuurlijk een subjectieve complexiteit[16], deze voorbeelden geven wel aan dat er lang niet altijd blind vertrouwd kan worden op de Cyclomatic Complexity van een stukje code.

Natuurlijk zijn dit nog slechts voorbeelden met een beperkte omvang. Uit verschillende studies [19, 20, 23] blijkt echter dat de omvang, gemeten in Lines Of Code (LOC), en Cyclomatic Complexity een opmerkelijke relatie met elkaar hebben. Beide metrieken blijken namelijk hetzelfde te meten en, ongeacht de gebruikte ontwikkelomgeving, paradigma of methode, evenredig toe te nemen naarmate de omvang van de software toeneemt. Dit geeft dus aan dat bovenstaande voorbeelden nog steeds opgaan voor software systemen met een grotere omvang. Dit terwijl McCabe zelf Cyclomatic Complexity juist als alternatief van Lines Of Code (LOC) zag [1,19].

Ongeacht de omvang van het systeem is het punt hier dat er geen gebruik gemaakt wordt van andere metriecken die juist in combinatie met elkaar een veel completer en correcter beeld geven [7, 11, 15]. Voor deze metriecken kun je bijvoorbeeld denken aan:

| Metriek | Omschrijving |
|---------------------|---|
| Lines Of Code | Het aantal lines of code dat een methode gebruikt. Hoe hoger het aantal lines of code, hoe kleiner de kans op complexiteit [21]. |
| Fan-in / fan-out | Het aantal in- en/of outputs van een methode. Hoe hoger dit getal ligt, hoe groter de kans op complexiteit. |
| Number of variables | Het aantal gebruikte variabelen binnen een methode. Hoe hoger dit getal ligt, hoe groter de kans op complexiteit. |
| Development hours | De besteedde tijd aan een methode. Wanneer dit een relatief korte periode betreft maar wel een complexe methode dan is de kans op complexiteit hoger. |
| Error rates | Het aantal gevonden problemen in een stuk code. Hoe hoger dit getal ligt hoe groter de kans op complexiteit. |

Tabel 5: Overige metriecken

Het blijkt dus dat ook Cyclomatic Complexity zelf meer alleen een beeld van de omvang van software dan dat het echt een compleet beeld geeft van de complexiteit van deze software of onderdelen daarvan [15,20]. Een goede vergelijking van Kearney en Sedlmeyer voor het bepalen van complexiteit gaat als volgt [22]: ".Even when a large correlation is discovered, the finding often has little practical values .. the weight of basketball players is highly correlated with scoring ability. This does not mean that we should choose players by weight ..."

5. Tools

Voor het berekenen van de Cyclomatic Complexity zijn veel verschillende tools beschikbaar. Voor de gegevens uit dit Essay hebben we gebruikt gemaakt van de standaard developer metrics in Visual Studio 2008 en Studiotools van Exactmagic [12]. Opvallend hierbij is dat de Cyclomatic Complexity niet bij iedere tool op hetzelfde cijfer uitkomt en elke tool eigenlijk een eigen interpretatie heeft van McCabe's Cyclomatic Complexity. Het is dus van belang om hier rekening mee te houden wanneer Cyclomatic Complexity een belangrijke rol speelt in het verzamelen van gegevens voor bijvoorbeeld onderhoudsanalyse doeleinden.

6. Conclusie

Zoals is gebleken is de Cyclomatic Complexity van McCabe lang niet voldoende om een volledig en juist beeld te geven van de complexiteit van software. Het richt zich slechts op een klein onderdeel van het totaal aan informatie wat de complexiteit van software bepaald. Ook het gegeven dat het aantal Lines Of Code zo nauw samenhangt met Cyclomatic Complexity versterkt dit beeld. Echter, in de eenvoud van McCabe's Cyclomatic Complexity zit zowel het voor- als nadeel. Zo kan het prima gebruikt worden voor het bepalen van complexiteit tijdens de ontwikkelfase en kan het goed helpen als indicatie van het aantal testcases dat een methode minimaal zou moeten hebben. Echter, aan de andere kant, geeft het geen informatie over data complexiteit, het gebruik van variabelen, het aantal fan-in of fan-out's, etc., terwijl juist ook deze informatie het beeld van code complexiteit compleet kunnen maken welke. Zeker voor het bepalen van de complexiteit van legacy systemen heeft deze informatie een grote meerwaarde.

Andere complexiteitsmetriecken zoals de Halstead complexiteit geven wel een completer beeld van de complexiteit maar hebben vaak weer als nadeel dat deze pas achteraf kunnen worden bepaald. Het blijkt dat beide complexiteitsmetriecken voor een andere toepassing kunnen worden ingezet. Het is dus niet zo dat deze metriecken elkaar uitsluiten maar juist elkaar op verschillende vlakken kunnen aanvullen.

Daarnaast is er nog een groot grijs gebied voor het bepalen van complexiteit in de beleving van de ontwikkelaar. Het gaat hier om de subjectieve complexiteit. Omdat mensen nu eenmaal allemaal van elkaar verschillen zal dit verschil waarschijnlijk altijd blijven bestaan. Dit betekent dat er dus ook voor iedereen altijd een andere mate van complexiteit zal gelden.

McCabe's Cyclomatic Complexity is dus geen slechte metriek maar moet, net als alle andere metrieken, wel in de juiste context worden gebruikt en waar nodig met andere metrieken worden aangevuld. In mijn optiek is Cyclomatic Complexity meer een indicatie van de waarschijnlijkheid van de aanwezigheid van complexiteit. In combinatie met andere metrieken kan dan pas goed de echte complexiteit worden bepaald.

Bronnen

1. McCabe, T. (1976). *A Complexity Measure*. *IEEE Transactions on Software Engineering*
2. Myers, J. (1977). An extension to the cyclomatic measure of program complexity
3. Glover, A. (2004). *Code Improvement Through Cyclomatic Complexity*
4. Glover, A. (2006). *In pursuit of code quality: Monitoring cyclomatic complexity*
5. Harrison, M. (2008). *Managing complexity*
6. McConnell, S. (2004). *Code Complete: A practical handbook of software construction*
7. Bucknall, M. (2005). *Cyclomatic Complexity*
8. Koskinen, J. (2003). *Software Maintenance Costs*
9. Meijer, S. (2004). *Beheer kosten- en baten in de greep*
10. Smacchia, P. (2009). Rambling on Cyclomatic Complexity
11. Marco, L. (1997). Measuring software complexity
12. Studiotools (2009). *Opgehaald van <http://www.exactmagic.com/products/studiotools/>*
13. Wikipedia (2009), *Cyclomatic Complexity*
14. Wikipedia (2009), *Control flow*
15. Aivosto.com (2009). Complexity metrics
16. Daniel, G. Magel, K (1981), The subjective nature of programming complexity
17. Miller, George A (1955), The magical number seven, plus or minus two
18. Pichler, M. (2009), The value of complexity metric - Cyclomatic Complexity
19. Shepperd, M (1988), A critique of cyclomatic complexity as a software metric
20. The free library (2009), Cyclomatic complexity and lines of code: emperical evidence of a stable linear relationship.
21. Watson A.H, McCabe T.J. (1996), Structured testing: a testing methodology using the cyclomatic complexity metric
22. Kearney, R., Sedlmeyer, W.T. (1986), Software complexity measurement
23. Kim Y.W. (2003), Efficient use of code coverage in large-scale software development
24. Bailey C.T., Dingee W.L. (1981), A software study using halstead metrics