TECHNISCHE SCHULD

HOE VOORKOM JE EEN CRISIS ;)

Wim Bokkers, januari 2012

De laatste tijd zijn schulden volop in het nieuws. Europa lijdt onder een schuldencrisis. Dit artikel heeft niet de pretentie om dáár iets aan te kunnen veranderen.

Een schuldencrisis-in-het-klein kan ook veroorzaakt worden door **technische** schuld. Hoewel niet bedreigend op grote schaal, kunnen projecten er veel last van hebben. Het is namelijk een schuld waarover rente betaald moet worden in de vorm van inefficiënte projecturen. Voor de meeste projectmanagers blijft deze schuld echter verborgen. En zolang de klant nog in staat is te betalen, hoeven we ons er niet zo druk over te maken.

De parallel trekkend met Europa: zolang het economisch goed gaat, maken we ons niet druk om wat uitstaande schuld. Dat is typisch iets 'voor later zorg'. Maar zodra het economisch minder voor de wind gaat, blijkt die schuld ineens erg wezenlijk en kun je daar behoorlijk mee in je maag zitten.

In slechte tijden kán technische schuld ervoor zorgen dat projecten niet levensvatbaar zijn, omdat de te betalen 'rente' gewoonweg te hoog wordt. De klant is niet meer bereid of in staat om de benodigde prijs te betalen.

Met dit artikel beperk ik me tot het begrip technische schuld binnen het vakgebied software engineering. Daarbij richt ik me op het herkennen van technische schuld en het aflossen van teveel aan technische schuld om hoge rentebetalingen te voorkomen.

WAT IS TECHNISCHE SCHULD?

De term technische schuld (technical debt) is afkomstig van Ward Cunningham, die in 1992 voor het eerst het verband legde tussen technische complexiteit en schuld:

> *Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite... The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation, object-oriented or otherwise.*

Dit verband vloeit voort uit de wet die Meir Manny Lehman in 1980 formuleerde (Lehman's Law):

> *As an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it.*

De term technische schuld is dus ontstaan binnen het vakgebied software engineering. En dat is niet zo vreemd; met software kun je op duizenden manieren hetzelfde resultaat bereiken; daarmee wordt het een programmeur erg makkelijk gemaakt om foute keuzes te maken.

Voor minder goede code wordt een prijs betaald. Op korte termijn kan minder goede code voordeel opleveren, omdat een project daardoor sneller klaar is. Op langere termijn zal de 'schuldenlast' zodanig groeien dat de code niet meer te onderhouden of uit te breiden is zonder grote investeringen in de structuur van de code.

HOE ONTSTAAT TECHNISCHE SCHULD?

Er bestaan grofweg twee oorzaken voor het ontstaan van technische schuld:

- Onbewust
- Bewust

Onbewuste schuldopbouw is meestal het gevolg van onervarenheid en/of te weinig kennis. Deze vorm van schuldopbouw is het gevaarlijkst, omdat niemand het beseft en er dus ook geen maatregelen tegen worden getroffen. Onervarenheid kan daarnaast ook zorgen voor slechte tijdschattingen vanwege het niet onderkennen van projectrisico's als gevolg van slechte code.

Bewuste schuldopbouw ontstaat meestal vanwege tijdsdruk, soms met een commercieel motief: sneller opleveren levert meer geld op. Dit kan een verstandige keuze zijn, mits iedereen er zich van bewust is en er geld gereserveerd wordt om deze schuld op een aanvaardbaar niveau te houden.

Bewuste schuldopbouw kan ook ontstaan vanwege gemakzucht: waarom 'moeilijk' doen als het ook makkelijk kan. Eigenlijk is dit ook een vorm van onkunde, omdat de 'makkelijke' manier op langere termijn juist meer inspanning kost.

HERKENNEN VAN TECHNISCHE SCHULD

Het duurt soms lang voordat je aan de 'buitenkant' merkt dat er sprake is van technische schuld. Zodra de volgende symptomen zich aandienen, is het eigenlijk al te laat:

- Er wordt door programmeurs geklaagd over slechte of bij elkaar gehackte code.
- Elke verandering leidt tot een opeenvolging van nieuwe fouten in de software (en kan showstoppers tot gevolg hebben).
- Fouten die verholpen leken, duiken vaak weer op.
- Er treden regelmatig fouten op in code die eerst prima werkte.

Gelukkig is het ook mogelijk om veel vormen van technische schuld in een veel eerder stadium te herkennen. Door je eigen code regelmatig te (laten) reviewen kunnen de volgende zaken aan het licht komen:

- Code smells
- Anti-patterns

Code smells zijn symptomen in de source code van een programma die mogelijk de indicatie zijn van een dieperliggend probleem.

Anti patterns zijn veelvoorkomende ontwerp-'patronen' die niet effectief zijn en/of contraproductief werken op de langere termijn.

Het vinden van code smells en anti-patterns die daadwerkelijk bijdragen aan de technische schuld is een kunst op zich, want er zijn situaties waarin een afwijkende keuze juist het beste werkt.

AFLOSSEN VAN TECHNISCHE SCHULD

Technische schuld ontstaat beetje bij beetje. Het is de opeenhoping van kleine afwijkingen ten opzichte van een in aanvang smetteloze architectuur.

Het lukt niet altijd om de opbouw van technische schuld op voorhand te voorkomen, omdat de negatieve gevolgen van bepaalde keuzes pas gaandeweg een project duidelijk worden. Het is echter wel van belang om technische schuld af te lossen ruim voordat het uit de hand gaat lopen. Het regelmatig aflossen van technische schuld moet daarom een vast onderdeel vormen van het normale ontwikkelproces:

- **Focus op kwaliteit** van architectuur en ontwerp (niet-functionele eisen!)
- **Review** code regelmatig
- **Refactor** de code vaak (minimaal na elke review)

Volgens Wikipedia betekent refactoring:

> *Code refactoring is "disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior", undertaken in order to improve some of the nonfunctional attributes of the software. Typically, this is done by applying series of "refactorings", each of which is a (usually) tiny change in a computer program's source code that does not modify its functional requirements. Advantages include improved code readability and reduced complexity to improve the maintainability of the source code, as well as a more expressive internal architecture or object model to improve extensibility.*

Bij elke refactoring wordt een deel van de technische schuld afgelost, met als gevolg dat de code beter te onderhouden en uit te breiden is.

Om te kunnen refactoren is wel kennis nodig van de belangrijkste ontwerpprincipes. Deze principes kunnen worden samengevat met het acroniem ‘SOLID’. Het gaat om de volgende principes:

- Single Responsibility Principle (SRP)
- Open-Closed Principle (OCD)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

Deze principes zijn er voornamelijk op gericht om de afhankelijkheid tussen componenten te verkleinen (low coupling) en daarbij componenten te gebruiken die een goed afgebakend doel dienen (high cohesion).

Een ander belangrijk principe is ‘DRY’: Don’t Repeat Yourself.

Verstandige toepassing van deze principes leidt tot uitbreidbare, onderhoudbare en herbruikbare code.

Helaas zijn hier weer mitsen en maren bij te bedenken. Zo kost het maken van generieke en herbruikbare componenten erg veel inspanning. Het is bijvoorbeeld niet nodig of nuttig om een component dat één keer wordt gebruikt, herbruikbaar te maken. Zeker omdat herbruikbaarheid pas getoetst kan worden na veelvuldige toepassing. (Robert Glass’ Rules of Three).

Ook bestaat het gevaar van 'teveel' architectuur door het onnadenkend overal toepassen van deze principes. Dit zorgt dan juist weer voor slecht te begrijpen code die daardoor juist niet goed te onderhouden is. Het is daarom altijd raadzaam om de volgende acroniemen achter de hand te houden:

- YAGNI (You Ain't Gonna Need It)
- KISS (Keep It Simple Stupid)

Er bestaat geen standaardrecept waarmee alle technische schuld uitgebannen kan worden. Elke situatie is uniek en vraagt om een specifiek recept met kennis en ervaring als basis-ingrediënt.

VOORKOM EEN CRISIS

Met een beetje inspanning en alertheid is technische schuld goed binnen de perken te houden en kan een lokale schuldencrisis voorkomen worden. Zowel softwareontwikkelaars als projectleiders moeten wel beseffen dat dit alleen kan wanneer er ruimte vrijgemaakt wordt in de planning om code reviews en refactorings te kunnen uitvoeren.

VERDER LEZEN

Technical debt - Wikipedia
http://en.wikipedia.org/wiki/Technical_debt

Design Debt - Jim Shore
http://jamesshore.com/Articles/Business/Software Profitability Newsletter/Design Debt.html

Paying Down Your Technical Debt – Jeff Atwood
http://www.codinghorror.com/blog/2009/02/paying-down-your-technical-debt.html

Code smell – Wikipedia
http://en.wikipedia.org/wiki/Code_smell

Code refactoring – Wikipedia
http://en.wikipedia.org/wiki/Code_refactoring

SOLID – Wikipedia
http://en.wikipedia.org/wiki/SOLID_(object-oriented_design)