Engineer

Teknisk skuld

merkostnaden som en snabb nödlösning i dag ger senare

Vad det är

Teknisk skuld är det framtida priset för en genväg i ett systems design eller kod: ett val som sparar tid i dag och gör senare ändringar långsammare, mer riskfyllda eller omöjliga, ända tills genvägen har tagits bort igen. På engelska heter det technical debt, ofta förkortat till tech debt. Ordet är en metafor, och det valdes för att genvägen beter sig som ett lån. Den köper något nu och tar betalt för det senare, så länge den ligger kvar.

Idén kommer från Ward Cunninghams erfarenhetsrapport på konferensen OOPSLA 1992 om WyCash, ett system för portföljförvaltning skrivet i Smalltalk. Ett litet team hade överblick över ungefär fyra megabyte källkod, och programmets mogna delar hade reviderats eller skrivits om många gånger. Det är den konsolideringen, hävdar rapporten, som gjorde en jämn, gradvis tillväxt möjlig, och när den saknas är det den allvarliga fallgropen: kod som fungerar och gör kunden nöjd men aldrig har fått ordentlig form. Cunningham sa det på en rad: ”Shipping first time code is like going into debt.” Själva rapporten talar bara om skuld och om sin skuldanalogi. Senare arbeten använder hela namnet, och en artikel från Google från 2015 kallar teknisk skuld en metafor som Ward Cunningham introducerade 1992.

Forskare som möttes på Dagstuhl i april 2016 för att enas om vad begreppet täcker skrev en arbetsdefinition: en samling design- eller implementationskonstruktioner som är bekväma på kort sikt men skapar en teknisk kontext som kan göra framtida ändringar dyrare eller omöjliga. De avgränsar verkan till kvaliteter inne i systemet, framför allt hur lätt det är att underhålla och vidareutveckla det. Det skiljer skuld från en funktion som ingen byggde: skuldens pris faller på dem som ändrar systemet och når användarna bara genom dem. Samma rapport behåller det finansiella ordförrådet. Kapitalbeloppet är det genvägen sparade, eller vad en bättre lösning skulle kosta i dag. Räntan finns i två slag: löpande ränta (recurring interest), som betalas i

form av lägre tempo, följdfel och förlorad kvalitet, och växande ränta (accruing interest), det extra priset för varje ny bit programvara som byggs ovanpå koden som inte riktigt stämmer. Skuld finns mest i kod, men också i dokumentation, tester och kända fel.

Använd begreppet i ett beslut

Använd ”Vad är teknisk skuld?” i ett konkret beslut. Vi kan knyta begreppet till systemet ni bygger eller köper. Systemets form utarbetad på papper först, så att delarna passar ihop och kostnaden är känd innan någon skriver kod.

Se våra tjänster Hör av er →

Granska antagandena bakom ordet

Använd förklaringen och det länkade arbetet för att formulera frågor till projektet. Vi kan undersöka antagandena tillsammans innan ett verktyg väljs. En infrastrukturplan för 14 avdelningar — 14 forskningsavdelningar under en och samma infrastrukturplan

En infrastrukturplan för 14 avdelningar Hör av er →

Varför det spelar roll

Skuld som tas medvetet kan vara ett bra beslut. Cunninghams rapport säger att lite skuld snabbar upp utvecklingen, så länge den snabbt betalas tillbaka med en omskrivning, och Google-artikeln håller med om att det ofta finns goda strategiska skäl att låna. Det båda varnar för är skuld som blir liggande. I rapporten räknas varje minut som läggs på kod som inte riktigt stämmer som ränta, och hela utvecklingsorganisationer kan stanna av under tyngden av en implementation som ingen konsoliderade. Innan dess blir programmet något som ingen enskild person kan överblicka, vilket leder till programmerare som kan var sitt hörn, och till slut till en produkt som inte kan byta riktning.

Problemet är att skuld ofta inte är ett medvetet val. Dagstuhl-rapporten konstaterar att den ofta uppstår oavsiktligt och överraskar utvecklarna, och därför upptäcks den mest under underhåll, långt efter det beslut som skapade den. Orsakerna är vardagliga: tidspress, en nyckelperson som inte är tillgänglig eller för lite kunskap om en teknisk egenskap vid tillfället. Google-artikeln tillägger att dold skuld är den farliga sorten, eftersom den växer i tysthet.

Metaforen ger också utvecklare och chefer ett gemensamt språk. Dagstuhl-rapporten kallar den en stark kommunikationsmekanism, och den vision deltagarna beskriver omfattar att hantera skuld lika noggrant som fel och nya funktioner, med ett sätt att översätta mellan det utvecklare oroar sig för och det chefer beslutar. En studie som presenterades på seminariet visar vad det kan betyda i ett team. Graziela Tonin, Alfredo Goldman och Carolyn Seaman bad studenter i team med sammanlagt 60 personer att identifiera och följa sin skuld

uttryckligen. En synlig lista lät teamen förhandla om tid för refaktorering med sin kund, fungerade som minne för projektets omogna delar och visade om kodkvaliteten blev bättre. De flesta teammedlemmar sa att de kommunicerade mer, tänkte efter noggrannare om en skuld verkligen behövdes och refaktorerade oftare. Eftersom beslut om att ta skuld diskuterades, tillägger författarna, fick en del team färre experter som ingen annan kunde ersätta.

Så ser det ut i praktiken

Skuld blir hanterbar när den skrivs ned där teamet planerar sitt arbete. En post i ett register, skriven för den här artikeln, kan vara så kort som den här:

```
TD-14       Fakturasummor beräknas på tre ställen
Artefakter: billing/web, billing/export, reports/monthly
Orsak:      ett fast lanseringsdatum; det gick snabbare att kopiera
            regeln än att flytta den till en gemensam modul
Kapital:    ungefär 2 dagar att flytta regeln till en modul, med tester
Ränta:      varje ändring av skattereglerna görs och testas tre gånger,
            och en av de senaste fem ändringarna missade en av kopiorna
Nästa:      amortera vid nästa ändring av skattereglerna; exporten av
            e-fakturor som planeras det här kvartalet får inte bli
            kopia nummer fyra
Ägare:      faktureringsteamet
```

Fälten följer modellen i Dagstuhl-rapporten: en post knuten till konkreta artefakter, med en orsak, ett kapitalbelopp och en ränta. Det är räntefältet som gör posten användbar, för det gör en känsla av att faktureringskoden är rörig till ett pris som någon kan väga. Ändras skattereglerna fyra gånger om året och kostar varje ändring en extra dags arbete och testning, är de två dagarnas kapitalbelopp intjänade inom ett halvår. Den planerade exporten är växande ränta i Dagstuhl-mening: varje ny del som byggs på den kopierade regeln gör det dyrare att rätta den senare. Vore koden däremot frusen, och väntade sig ingen att röra den igen, vore samma post ett argument för att låta skulden ligga.

Skuld handlar inte bara om kod. Portföljposten här på webbplatsen om att förnya interna processer och spara 8 000 timmar dokumenterar grundarens tidigare arbete hos Mappitall. Den frågar läsaren om teknisk skuld bromsar dem och beskriver den sort som samlas kring hur arbetet går till: en mjukvaruarkitektur som hade vuxit genom pålagringar snarare än design, system som fungerade men inte effektivt, och schemaläggning som gjorde att folk antingen väntade eller var överbelastade. Inget av det brann i knutarna, vilket är precis varför det

hade lämnats i fred. Att lägga om arkitekturen, systemen och schemaläggningen sparade ungefär 8 000 timmar under arbetet från 2015 till 2022, och portföljposten betonar att förändringarna bäddades in i hur teamet arbetade, eftersom processförbättringar som inte görs permanenta glider tillbaka till det gamla sättet.

Var det går fel

Allt kallas teknisk skuld. När uttrycket väl har blivit populärt får varje irritationsmoment etiketten, från ett gammalt bibliotek och en saknad funktion till en kollegas stil. Dagstuhl-rapporten tar upp den risken direkt och efterfrågar skarpa definitioner som svar. Använd bara begreppet om genvägar vars pris landar på framtida ändringar, registrera fel som fel och önskemål som funktioner, och avvisa en post i registret som inte kan ange sin orsak och sin ränta.

Skulden finns bara i folks huvuden. Skuld som ingen har skrivit ned betalas av den som rör koden nästa gång, oftast utan att veta varför arbetet tar så lång tid. Håll ett register bredvid backloggen och kontrollera det innan en del av systemet ändras, så som studentteamen i Tonin-studien använde sin lista. En post blir synlig för dem som beslutar vad tiden läggs på först när den står där de tittar.

Kapitalet betalas av där ingen ränta löper. Att städa i en modul som ingen ändrar känns produktivt och ger ingenting tillbaka, medan koden som ändras varje vecka fortsätter att kosta. Ränta löper bara där folk arbetar, så sortera registret efter hur ofta varje del ändras och efter vad som snart ska byggas ovanpå den, och amortera på de dyra posterna först.

Amorteringen väntar på en enda stor omskrivning. Skuld som har legat i åratal möts ofta med ett projekt som ska ersätta hela systemet och som tar ny skuld för egen räkning medan det gamla systemet fortsätter att kosta. Cunninghams råd var att betala tillbaka snabbt, och Google-artikeln räknar upp de vanliga sätten att amortera: refaktorering av kod, bättre enhetstester, borttagning av död kod, färre beroenden, stramare API:er och bättre dokumentation. Inget av dem lägger till en funktion; poängen är att göra nästa ändring billigare. Planera in dem som en del av det vanliga arbetet, nära den ändring som behöver dem.

Experiment blir kvar i produktion. Att prova ett alternativ som en villkorad gren i produktionskoden är billigt för varje enskilt experiment, och Google-artikeln beskriver hur sådana grenar samlas på hög tills det blir svårt eller omöjligt att testa alla kombinationer. Den nämner Knight Capital, vars system enligt artikeln

förlorade 465 miljoner dollar på 45 minuter, till synes för att föråldrade experimentella kodvägar betedde sig oväntat. Gå igenom varje experimentell gren och varje flagga enligt en fast plan, ta bort det som inte längre används och belöna borttagning av kod lika mycket som tillägg.

Ett verktygs poäng tas för skulden. Statisk analys kan räkna duplicerade rader och långa funktioner, och en dashboard får siffran att se ut som ett saldo. Google-artikeln säger rakt ut att metaforen inte ger något strikt mått att följa över tid, och att ett team som fortfarande rör sig snabbt inte bevisar att skulden är låg. På Dagstuhl berättade Forrest Shull att ingen enskild uppsättning regler för att hitta skuld hängde ihop konsekvent med dess symtom, som stigande ändringskostnader och sjunkande tempo, över olika projekt. Håll ett verktygs fynd mot er egen historik av ändringar och fel, behåll de regler som förutsäger kostnader och släpp resten. Artikelns egna frågor är ett bättre prov, bland annat hur snabbt en ny medlem i teamet kommer in i arbetet.

Läs mer

- The WyCash Portfolio Management System — Ward Cunninghams erfarenhetsrapport från OOPSLA ‘92, där skuldanalogin dyker upp första gången.
- Managing Technical Debt in Software Engineering — rapporten från Dagstuhl-seminariet 16162 med arbetsdefinitionen, kapitalbelopp och ränta och studien av uttryckliga skuldlistor.
- Hidden Technical Debt in Machine Learning Systems — artikeln från 2015 av Sculley och kollegor på Google om hur skuld samlas på hög, hur den amorteras och varför den är svår att mäta.
- Refactoring — Martin Fowlers webbplats om de små, beteendebevarande ändringar som Google-artikeln nämner först bland sätten att amortera skuld.

Ordlista

Välj utifrån era faktiska förutsättningar

Beskriv arbetsflödet, befintliga gränssnitt och användarna som behöver förändringen. Vi kan definiera en testbar leverans och en överlämning som gör att teamet kan underhålla lösningen.

Se vad det kostar Hör av er →

Gör begreppen praktiska

Fråga hur ”Vad är teknisk skuld?” gäller i er organisation. Ett konkret exempel från ert nuvarande system räcker för att börja samtalet.

Se våra tjänster

welcome@engineer.company +45 50 10 61 84 Köpenhamn, Danmark

Ta med dig den här sidan

Visitkort

Omsorgsfullt skapad med kärlek av D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Upphovsrätt © 2025 – nu · Engineer ApS · Alla rättigheter förbehållna.

Utskrivet från https://engineer.company/sv/glossary/technical-debt/