XIX

Jak se pozná, že to funguje?

Po této kapitole přestaneš věřit větě „funguje to“ a začneš žádat důkaz — jeden test, který by spadl, kdyby byl kód špatně. Naučíš se stavět testy dřív, než pustíš model generovat; obrátíš role tak, že úsudek zůstane tobě; a uvidíš, proč je zpětná vazba, která nespí a negratuluje, ten nejlepší přítel, jakého při práci s AI máš.

Testování ukáže přítomnost chyb, nikdy ne jejich nepřítomnost.

— Edsger W. Dijkstra, *„Testing shows the presence, not the absence of bugs“ · konference NATO, Řím, 27.–31. 10. 1969; in J. N. Buxton, B. Randell (eds.),* Software Engineering Techniques*, 1970, s. 16*

Stroj, který zabíjel s jistotou

Therac-25 byl zázrak své doby: počítačem řízený ozařovač firmy AECL, který jedním přístrojem uměl to, co dřív dělaly dva. V nižším režimu posílal do nádoru slabý svazek elektronů; ve vyšším roztočil stokrát silnější rentgenový paprsek a mezi svazek a pacienta zasunul kovový terč a clonu, které tu smrtící sílu ztlumily na léčebnou dávku. Celé to hlídal software. A právě tím se lišil od svých předchůdců, Theraku-6 a Theraku-20: ty měly *hardwarové pojistky* — fyzické západky, které přístroj nepustily vystřelit plnou silou, dokud nebyl terč na místě. U pětadvacítky je konstruktéři vypustili. Software je přece ověřený; proč zdvojovat drátem, co hlídá program.

Mezi červnem 1985 a lednem 1987 předávkoval Therac-25 nejméně šest pacientů — pokaždé mnohonásobkem léčebné dávky. Nejméně tři na následky zemřeli. V Tyleru v Texasu roku 1986 se ukázalo, co se dělo. Zkušená operátorka, která zadávání znala nazpaměť, opravila na obrazovce režim tak rychle, že program nestihl dojet přenastavení mechaniky. Vznikla *souběžná chyba* (race condition): dvě části programu se míjely v čase a jedna věřila, že terč je zasunutý, zatímco druhá ho zrovna odsunula. Stroj vystřelil plný rentgenový svazek bez clony přímo do pacienta. Na obrazov-

ce blikalo lakonické *Malfunction 54* — hláška, která operátorce neřekla, jestli šlo o maličkost, nebo o smrtelnou dávku; a protože se přístroj tak často mýlil v drobnostech, zmáčkla „pokračovat“.

Byla to jen jedna ze dvou závad. Tu druhou odhalila nehoda v Yakimě v lednu 1987: jednobajtový čítač v přípravné rutině přetékal. Bajt drží čísla nula až dvě stě padesát pět; při dvěstěpadesátéšesté kontrole se přehoupl zpátky na nulu — a přesně v ten okamih software přeskočil ověření, že je clona na místě, protože nulu četl jako „vše připraveno“. Kdo obsluhu zrovna potvrdil ve chvíli přetečení, poslal do pacienta holý svazek. Vyšetřovatelky Nancy Levesonová a Clark Turner to o šest let později rozebraly do posledního detailu a jejich zpráva se stala povinnou četbou každého softwarového inženýra. Napsaly větu, kterou si přepiš nad monitor: nikdo ten software nikdy pořádně neotestoval na kombinacích, které se v praxi děly. Věřilo se mu. Nebyl důvod — byla jen důvěra.

Prameny: N. G. Leveson, C. S. Turner, An Investigation of the Therac-25 Accidents, *IEEE* Computer *26(7), červenec 1993. AECL Therac-25; ≥ 6 masivních předávkování 6/1985–1/1987, ≥ 3 úmrtí. Souběžná chyba (Tyler, Texas, 1986) spuštěná rychlým zadáváním zkušené operátorky; přetečení jednobajtového čítače (Yakima, 1/1987) jako druhá, nezávislá závada.*

„Software je přece ověřený“ *je parafráze postoje, ne doslovný citát: Therac-6 a Therac-20 měly hardwarové pojistky, které Therac-25 postrádal — spolehlo se na software, jenž se pod nezávislou kontrolu nikdy nedostal.*

Nečti Therac-25 jako historku o zlé firmě nebo hloupých programátorech. Ti lidé nebyli hloupí a stroj většinu času léčil, jak měl. To je právě ta past: *většinu času fungoval*. Kdo se díval, viděl přístroj, který dělá svou práci — a z toho pozorování si každý odvodil, že je v pořádku. Jenže „viděl jsem ho fungovat“ a „vím, že funguje“ jsou dvě různé věty a mezi nimi leží celá tahle kapitola. To první je dojem. To druhé je důkaz. A důkaz, že program dělá, co má, se jmenuje *test*.

„Funguje to“ je nejnebezpečnější věta v řemesle

Vrátím se teď k jizvě, kterou jsem otevřel v kapitole čtvrté a nechal ležet. Slíbil jsem, že se k ní vrátíme, až budeme mít nástroj — a ten nástroj je tahle kapitola.

> Dva roky. Dva roky jsme s kolegou tlačili projekt, na kterém stála pořádná část práce celého týmu, a celou tu dobu jsem měl skvělý pocit. Seděl jsem nad tím po nocích, řešil jednu záludnost za druhou, a čím víc mě to bolelo, tím jistější jsem si byl, že děláme něco pořádného — vždyť to dá takovou práci. Pocit rostl s dřinou. Nerostl se správností. Uvnitř totiž seděla chyba tak hloupá, že se stydím ji popsat: jedna hodnota, která nikdy neměla být záporná, občas záporná byla, a všechno ostatní se od ní ohýbalo. Dva roky jsme obcházeli důsledky místo příčiny. Přišli jsme na to náhodou, ne metodou — a když jsem tu příčinu konečně viděl, došlo mi to horší: *jeden jediný test*, jedna věta „tahle hodnota nesmí být záporná“, spuštěná v prvním týdnu, by nás zastavila na místě. Nemuseli jsme na to přijít my. Stačilo napsat stroji,

aby to hlídal za nás. Pocit dřiny nás nesl dva roky. Test by nás zradil hned první den — a v tom je jeho celý dar.

Zastav se u toho slova: *zradil*. Zní jako urážka a je to poklona. Pocit z dobře odvedené dřiny je milý přítel, který ti tleská; problém je, že tleská i tehdy, když děláš nesmysl — protože nekóduje správnost, kóduje námahu. Test je opak. Test je zhmotněná zpětná vazba, která *nespí, nepodléhá náladě a negratuluje ti*. Je ti jedno, jak těžké to bylo napsat; ptá se jen na jednu věc — platí, co má platit, nebo ne? — a když neplatí, řekne to bez ohledu na to, jak moc jsi do toho vložil. Semmelweisova tabulka z kapitoly devět byla přesně tohle: přístroj, který mu byl ochoten kdykoli sdělit, že se plete. Test je ta tabulka přeložená do kódu a spuštěná automaticky, pokaždé.

A tady je věta, kterou nes celou kapitolou jako nit: *„funguje to“ je nejnebezpečnější věta v řemesle.* Ne proto, že by lhala vždycky — často je pravdivá. Nebezpečná je proto, že zní stejně, ať je pravdivá, nebo ne. „Funguje to“ z úst někoho, kdo to viděl jednou proběhnout, a „funguje to“ z úst někoho, kdo na to má sto testů, jsou nerozlišitelné zvuky — a přesně proto potřebuješ něco, co ten rozdíl slyší za tebe.

Dobrá zpráva, kvůli které to celé stojí za práci: co Therac stálo životy a mě dva roky, dnes napíšeš za odpoledne. Napsat test býval otravný obřad; dneska řekneš asistentovi „napiš mi test, který ověří, že faktura nikdy nevyjde záporná“, a máš ho za pár vteřin. Cena zpětné vazby spadla skoro na nulu — a to znamená, že poprvé v dějinách si *každý* může postavit tu neúnavnou tabulku, na kterou Semmelweis potřeboval kliniku a roky. Zbývá jediné: chtít ji.

→ kap. 4: pocit dřiny nic nedokazuje — čím víc práce, tím větší jistota, a ta jistota je klam (effort heuristic). Test měří správnost, ne námahu; proto tě zradí, když si nejvíc věříš.

→ kap. 9: refrén „jak bych poznal, že se pletu?“ *Test je odpověď zapsaná do kódu předem:* tenhle výsledek by musel nastat, aby se ukázalo, že je to špatně.

Řemeslo: pyramida, smyčka a pomník

TEENAGER · MECHANISMUS

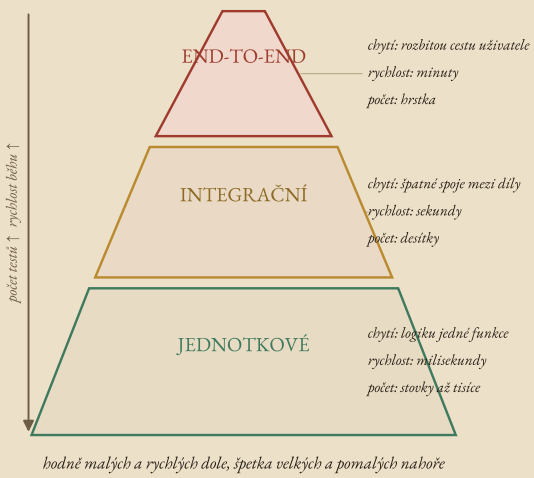
Testy nejsou jeden druh nářadí; jsou tři vrstvy, každá chytá jiný druh chyby, a dohromady tvoří *pyramidu*. Dole hodně malých a rychlých, nahoře špetka velkých a pomalých — a když ten poměr obrátíš, testy tě začnou brzdit místo chránit.

```
# JEDNOTKOVÝ TEST — jedna funkce, jeden
invariant, milisekundy
test_že(faktura(množství: 3, cena: 100) == 300)
test_že(faktura(množství: 0, cena: 100) == 0)
test_že(faktura_není_záporná(sleva: 120, cena:
100))   # ← ten, co mi chyběl

# INTEGRAČNÍ TEST — spolupráce dílů: sedí spoj
mezi nimi?
objednávka = vytvoř_objednávku(zboží, sklad)
test_že(sklad.zbylo == původní -
objednávka.množství)

# END-TO-END TEST — celá cesta uživatele od
kliknutí po výsledek
otevři_stránku(); přidej_do_košíku("kniha");
zaplať()
test_že(vidím("Děkujeme za nákup"))
```

Jednotkových testů měj stovky: jsou levné, běží v mžiku a přesně ukážou, *kde* to prasklo. End-to-end testů měj hrstku: dají jistotu, že to celé drží pohromadě, ale jsou pomalé a když spadnou, hledáš příčinu dlouho.



hodně malých a rychlých dole, špetka velkých a pomalých nahoře

Hrdinský graf kapitoly. Zdola nahoru: široká základna jednotkových testů (chytí logiku jedné funkce, běží v milisekundách, je jich tisíce), užší pásmo integračních (spoje mezi díly, sekundy, desítky), špička end-to-end (celá cesta uživatele, minuty, hrstka). Šipka vlevo: čím níž, tím víc testů a tím rychleji běží. Obrácená pyramida — pár pomalých testů nahoře, nic dole — je past: pomalá a slepá k tomu, kde *chyba je.*

end-to-end (E2E) — „celá cesta uživatele“: test, který projde aplikaci tak, jak by ji proklikal člověk, od začátku do konce. Nepřekládá se; glosa jednou a dál E2E.

Když máš pyramidu, přijde otázka, kdy testy *psát*. Nejsilnější odpověď obrací pořadí, na které je většina lidí zvyklá: test *napřed*, kód potom. Jmenuje se to vývoj řízený testy a drží se ve třech krocích, které se opakují pořád dokola.

TEENAGER · MECHANISMUS

Smyčka *červená → zelená → refaktor* — tři kroky, které nikdy neměníš pořadí:

```
1. ČERVENÁ    Napiš test na chování, které kód
ještě NEUMÍ.
              Spusť ho. MUSÍ spadnout (červená) —
tím ověříš,
              že test vůbec něco měří, a ne že
prošel omylem.

2. ZELENÁ     Napiš NEJMENŠÍ kus kódu, po kterém
test projde.
              Ne elegantní, ne obecný — jen
zelený. Teď máš
              jednu ověřenou pravdu, na které
stojíš.

3. REFAKTOR   Teď kód ukliď, zkrášli, zjednoduš —
a testy ti
              hlídají záda: dokud svítí zeleně,
nic jsi nerozbil.
```

Krok jedna je ten, který lidé vynechávají, a je nejdůležitější. *Test, který jsi neviděl spadnout, není test* — je to modlitba. Musíš ho vidět červený na chování, které kód neumí, jinak nevíš, jestli něco měří, nebo jen vždycky svítí zeleně bez ohledu na svět.

Krok „červená“ je Popper z kap. 9 v provozní podobě: dobrý test musí umět spadnout. *Tvrzení, které nedokáže selhat, ti nic neslibuje — a test, který nikdy nezčervená, nic nehlídá.*

Tři kroky jsou příbuzné třem stromům gitu (kap. 18): working tree „píšu“, index „chystám k důkazu“, commit „zapečetěno“ — táž disciplína malých ověřených kroků.

A když už bug jednou vylezl a tys ho opravil? Napíšeš na něj test — a ten test se jmenuje *regresní*. Je to **pomník bugu**: hrob, na který se chodíš přesvědčit, že mrtvý je pořád mrtvý. Každá vada, kterou jsi kdy lovil, si zaslouží jeden řádek, který hlídá, aby se nevrátila zadními vrátky při příští úpravě. Kdyby měl Therac na svou souběžnou chybu regresní test, druhá nehoda po první nikdy nepřijde. Sbírka regresních testů je zhmotněná paměť tvých omylů — a čím je delší, tím míň chyb potkáváš podruhé.

regresní test — test, který hlídá, aby se jednou pohřbený bug nevrátil (pomník bugu). Pozor na kolizi se statistickou regresí *z kap. 10: společný předek je „návrat“ — tady návrat chyby, tam návrat k průměru.*

→ kap. 21: tuhle sbírku testů pak spouští linka (CI) *za tebe po každé změně — i v noci, i když spíš. Zpětná vazba, která nikdy nemá volno.*

Obrácení rolí: testy jako zadání pro stroj

Teď to nejdůležitější celé kapitoly pro toho, kdo programuje s asistentem. Dosud jsi test bral jako soud *po* napsání kódu. Obrať to. *Napiš testy sám, dřív než pustíš model generovat* — a nech ho psát kód tak dlouho, dokud tvoje testy nesvítí zeleně.

Proč je to tak silné? Protože tím děláš dvě věci najednou. Za prvé: testy jsou *specifikace*. Ve chvíli, kdy dokážeš napsat test „faktura nikdy nevyjde záporná“, jsi přesně, měřitelně a bez mlžení definoval, co „správně“ znamená — a to je zadání, kterému model rozumí líp než odstavci přání. Za druhé, a podstatněji: úsudek zůstává *tobě*. Model umí generovat plynule i nesmysl (kapitola šestnáct); když necháš soud na něm, necháš lišku hlídat kurník. Ale když soud napíšeš ty a on jen plní, obrátil jsi role tak, že stroj dělá tu snadnou půlku (psaní kódu) a ty tu těžkou a nezcizitelnou (co má vlastně platit). Testy jsou to místo, kde se do řetězu vrací tvůj úsudek.

Je tu ovšem jeden háček a je tvrdý: *tenhle pakt platí, jen když test píšeš ty, ne když si ho necháš vygenerovat od téhož modelu*. Kdo požádá stroj „napiš kód i testy k němu“, dostane kód a k němu testy, které tenhle kód pochválí — papoušek, který zkouší sám sebe. Model si k vlastnímu dílu napíše zkoušku, kterou vlastní dílo projde; to není soud, to je autogratulace v hávu inženýrství. Test má cenu přesně tolik, kolik nezávislého úsudku jsi do něj vložil ty. Nech stroj psát kód. Soud napiš sám.

→ kap. 8: napsat testy před *generováním je pakt — smlouva s budoucím já uzavřená za střízliva, sourozenec „predikce zapsané před simulací“ (kap. 14) a „commitu před přepsáním“ (kap. 18). Napřed řekni, co je správně; teprve pak se ptej stroje.*

DEMO

`lesk-a-bida-vibecodingu.gaussalgo.com/d/napis-test-chyt-bug`

Napiš test, chyť bug: schoval jsem do funkce jednu chybu — takovou, jaká mě stála dva roky. Piš asserty a sleduj čas do odhalení; jakmile napíšeš ten jeden, který na chybu sáhne, spadne červeně a ty víš přesně kde.

Věž: co musí platit vždy — a proč to nikdy nedokážeš celé

EINSTEIN · MATEMATIKA — přeskočitelné

Od příkladů k invariantům. Jednotkový test kontroluje *jeden* příklad: „faktura(3, 100) je 300“. Ale příkladů je nekonečně a ty jich napíšeš pět. Silnější je zeptat se, co musí platit *pro každý vstup, ať přijde jakýkoli* — tomu se říká **invariant**, a testování, které ho prověřuje, se jmenuje *testování vlastností* (property-based testing). Místo abys sám vymýšlel vstupy, popíšeš pravidlo a stroj vrhne proti kódu stovky náhodných případů:

$$\forall x \in \text{vstupy}: \quad \text{platí}(\text{invariant}(x)).$$

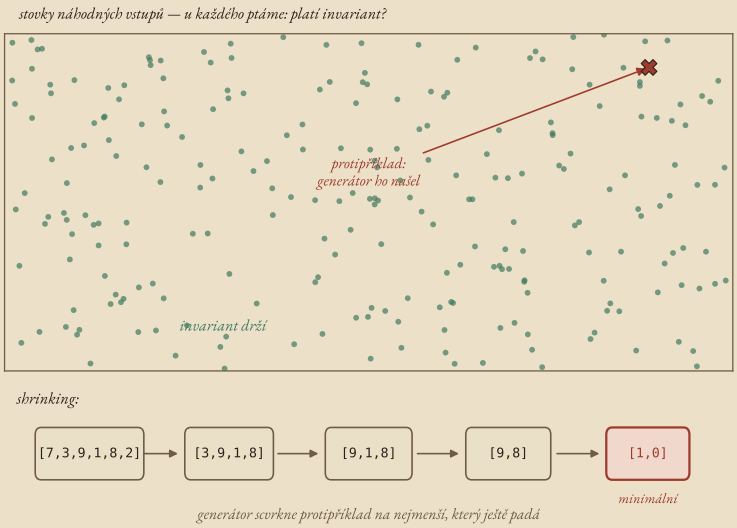
Příklad z mé jizvy: místo pěti konkrétních faktur napíšeš jediné pravidlo — *pro každé množství a každou cenu je výsledná částka nezáporná* — a generátor ti do něj nasype tisíc kombinací, na které bys sám nikdy nepomyslel, včetně té jedné se slevou vyšší než cena, co mě stála dva roky.

invariant — co musí platit vždy, *ať přijde na vstup cokoli. Ne „na tomhle příkladu to vyšlo“, ale „na žádném příkladu to nesmí selhat“. Testování vlastností (property-based) hledá protipříklad za tebe.*

V praxi: knihovny jako Hypothesis (Python) nebo QuickCheck (Haskell) — generátor a shrinking dostaneš hotové.

EINSTEIN · MATEMATIKA — přeskočitelné

Generátory a shrinking. Dvě součástky dělají z náhodného sypání použitelný nástroj. *Generátor* umí vyrobit náhodný vstup správného tvaru: náhodné číslo, náhodný seznam, náhodnou fakturu. *Shrinking* (scvrkávání) je ta chytřejší půlka: jakmile generátor najde vstup, který invariant poruší, málokdy je to hezký protipříklad — bývá to obří, náhodný nepořádek. Shrinking ho pak systematicky *zmenšuje* — ubírá prvky, snižuje čísla —, dokud nezbude *nejmenší* vstup, který ještě padá. Z nálezu „seznam dvou set čísel selhává“ se stane „`[1, 0]` selhává“ — a to už ti řekne příčinu skoro samo.



Nahoře stovky náhodných vstupů proti invariantu: skoro všechny drží (zeleně), generátor ale najde jeden, který padá (sanguine křížek). Dole shrinking*: z velkého náhodného protipříkladu se řetězem odvozuje pořád menší, který stále selhává, až zbude minimální* `[1, 0]`*. Nejmenší padající případ je skoro hotová diagnóza — na rozdíl od původního nepořádku ho přečteš okem.*

EINSTEIN · MATEMATIKA — přeskočitelné

Mutation testing: kdo hlídá hlídače? Testy hlídají kód. Co ale hlídá testy? Odpověď je nádherně jednoduchá: schválně *rozbij kód* a koukni, jestli si toho testy všimnou. *Mutation testing* vezme tvůj program a nadělá do něj drobné „mutace“ — změní `<` na `<=`, `+` na `-`, smaže řádek — a pak spustí testy. Když se test na každou takovou úmyslnou chybu *rozzáří červeně*, testy hlídají. Když mutace přežije, aniž by kterýkoli test protestoval, máš díru: kus kódu, který nikdo neměří. Je to červená–zelená smyčka obrácená naruby — tady chceš, aby test spadl, protožes ho o to schválně požádal.

Metrika pokrytí *(coverage) říká jen, které řádky se při testech* spustily *— ne že se* ověřily. *Řádek proběhne i bez jediného assertu. Mutation testing měří, co testy doopravdy hlídají; proto je poctivější než pokrytí.*

EINSTEIN · MATEMATIKA — přeskočitelné

Meze: pokrytí není korektnost — a Riceova věta říká, proč nikdy nebude. Ať napíšeš testů kolik chceš, dokazují jen přítomnost chyb tam, kde spadly — ne jejich nepřítomnost jinde. To je Dijkstra z epigrafu. A není to lenost ani nedostatek nástrojů; je to *matematická nemožnost*. Roku 1953 dokázal Henry Gordon Rice větu, která říká: každá netriviální otázka o tom, *co program dělá* (ne jak vypadá, ale jak se chová), je obecně nerozhodnutelná — neexistuje a nemůže existovat algoritmus, který by pro libovolný program spolehlivě řekl „tenhle je správně".

$$\text{žádný } A: \quad A(\text{program}) = \begin{cases} \text{„správně“} & \text{pro každý korektní program} \\ \text{„špatně“} & \text{pro každý vadný} \end{cases}$$

Proto testy nikdy nedají *jistotu* korektnosti — dávají jen rostoucí důvěru úměrnou tomu, kolik způsobů, jak se to mohlo pokazit, jsi vyloučil. To není slabina metody; je to poctivé přiznání jejích hranic. Kdo slibuje „otestováno, tedy bezchybné", prodává ti Therac.

Riceova věta (H. G. Rice, 1953) — jediná glosa pokory: žádný obecný algoritmus nerozhodne netriviální vlastnost chování programu. Zobecnění problému zastavení. Důsledek pro tebe: nástroj, který slíbí prověřit libovolný *program na správnost, je nemožný — a kdo ho nabízí, klame.*

EINSTEIN · MATEMATIKA — přeskočitelné

Souběžné chyby a proč je běžný test nechytí. Therakova race condition neležela v žádném řádku — ležela v *čase* mezi dvěma řádky. Běžný test spustí kód v jednom pořadí a změří výsledek; jenže souběžná chyba se projeví jen při *jednom konkrétním prokládání* dvou souběžných dějů z miliardy možných, a ten jeden test skoro nikdy netrefí. Proto Therac procházel: v laboratoři operátor nikdy nezadával tak rychle jako zocelená profesionálka v provozu. Na tohle existuje těžší kalibr — *model checking*: nástroj, který systematicky proiteruje *všechna* možná prokládání a hledá to jedno vražedné. Je to okno do formálních metod, kam běžné testování nedosáhne — a připomínka, že u systémů, kde chyba stojí život, je pyramida testů začátek, ne konec.

→ kap. 20: co test nechytí, musí ustát robustní design *— proto Therac chyběly hardwarové pojistky. Testy jsou první vrstva obrany, ne jediná; když jedna selže, drží další. Omyly nikdy nezmizí, proto systémy stavíme tak, aby jeden omyl nebyl konec.*

Co si odnést z testů

Poskládej to dohromady. Therac-25 nebyl příběh zlých lidí; byl to příběh věty „funguje to", vyslovené o stroji, který většinu času opravdu fungoval — dokud nezabil. Rozdíl mezi „viděl jsem to fungovat" a „vím, že to funguje" je test: zpětná vazba, která nespí, nepodléhá náladě a negratuluje. Mě ta chybějící věta stála dva roky; kdybych ji napsal první týden, zradila by mě hned — a to je její dar, ne urážka.

A tři věci si odnes do ruky. *Pyramida:* hodně malých rychlých testů dole, špetka velkých pomalých nahoře — nikdy obráceně. *Obrácení rolí:* u kódu, který píše stroj, napiš soud sám a nech ho plnit, protože

kdo si nechá napsat i zkoušku, nechává papouška opravovat papouška. A *pokora:* testy dokazují přítomnost chyb, ne jejich nepřítomnost — Riceova věta zaručuje, že jistotu korektnosti nedostaneš nikdy, jen rostoucí důvěru. Proto je omyl věčný a proto příští kapitola nestaví systémy bezchybné, ale robustní — takové, které jeden omyl přežijí a vyjdou z něj silnější. Chyby jsou základ poznání (kapitola devět); zpětná vazba je lék; ale protože chyby nikdy nezmizí, potřebuješ i stroj, který ránu ustojí. O tom je most, po kterém teď přejdeme dál.

„Jak bych poznal, že se pletu?“

Konkrétní test k této kapitole máš v ruce pokaždé, když ti kód napsala AI: *napiš jeden jediný test, který by spadl, kdyby byl ten kód špatně.* Jednu větu, jeden assert — „tahle faktura nesmí vyjít záporná“, „tenhle seznam musí zůstat seřazený“, „tenhle součet se musí rovnat vstupu“. Když ho umíš napsat a spustit a on svítí zeleně, poprvé nemáš dojem, ale důkaz. A když ho napsat *neumíš* — když nedokážeš vyslovit jediný měřitelný výsledek, který by odlišil správný kód od špatného —, kapitola tvrdí něco nepříjemného: pak totiž nevíš, co „správně“ vlastně znamená, a všechno tvé „funguje to“ je jen ozvěna Theraku. Kapitola se plete jedině tehdy, jestli mi ukážeš kód, o němž s jistotou víš, že je správně, a přitom *žádný* takový test napsat nejde — a v tom případě chci ten kód vidět, protože buď jsi našel hranici metody, nebo jsi jen znovu spletl dojem s důkazem.