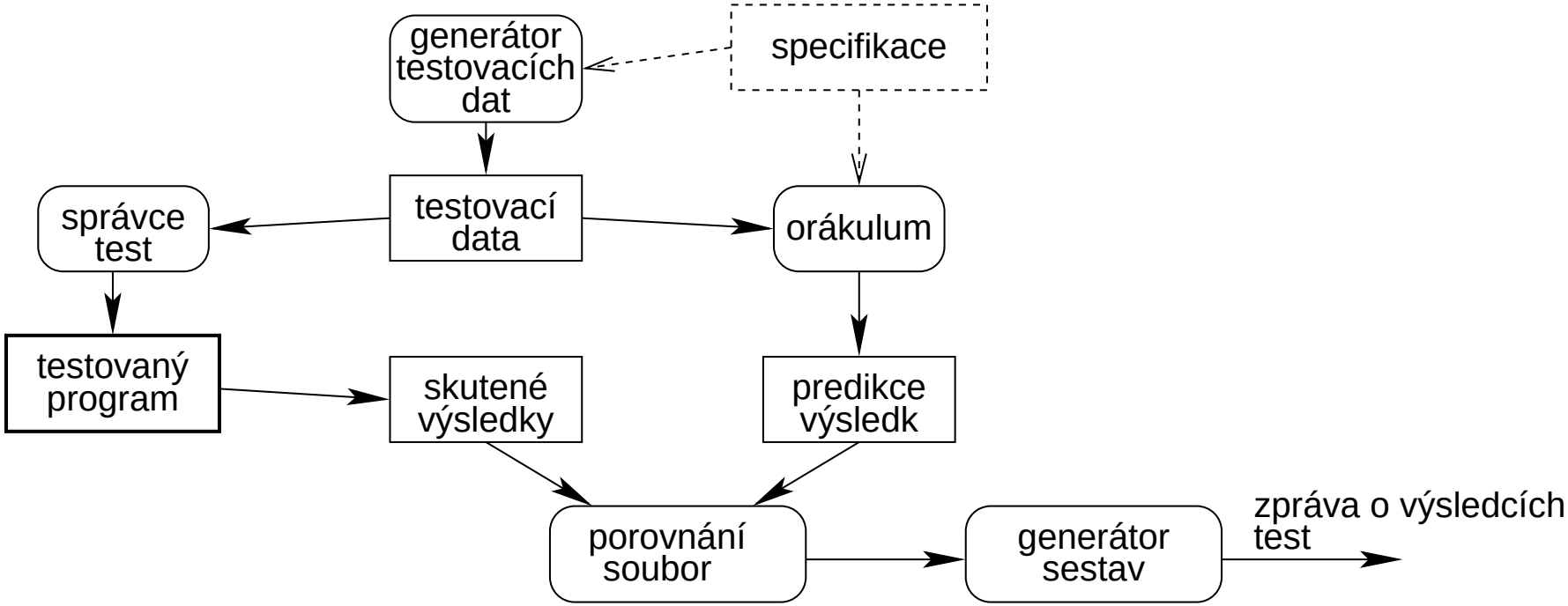
Nástroje pro testování SW

- rozsáhlé systémy - cena testování až 50% ceny vývoje, mohou existovat stovky až tisíce testovacích případů -> automatizace testů

- automatizace testů snižuje cenu změn - programátoři se nemusejí tolik obávat, že změnou zanesou do kódu defekt, protože testy by defekt (s určitou pravděpodobností) odhalily

- často následující uspořádání:



- správce testů (test manager) - řídí běh testů
- generátor testovacích dat (test data generator) - generuje testovací vstupní data pro testovaný SW
- orákulum (oracle) - generuje předpokládané výstupní hodnoty (nový program, prototyp SW, předchozí verze SW – tzv. regresivní testování, SW vytvořený konkurencí apod.)

- program pro porovnání souborů (file comparator) - porovná výsledky skutečného běhu s předpokládanými hodnotami vygenerovanými orákulem; často lze použít univerzální programy
- generátor zpráv (report generator) - umožňuje definovat a generovat zprávy o výsledcích testů

- progresivní fáze testování přidává a testuje nové fce (nově přidané nebo modifikované moduly a jejich rozhraní s již integrovanými moduly)

- regresivní fáze testuje důsledky změn v již integrovaných částech.

Nástroje pro zachycení a pozdější přehrání testů (capture-replay tool)

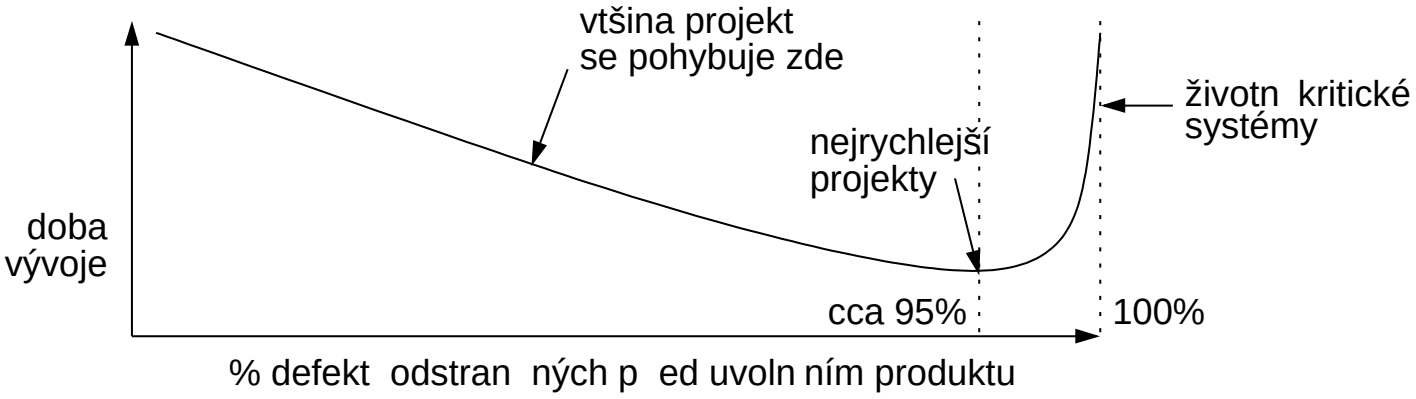
- informace protékají SW systémem - na vhodné místo toku dat aplikací vložíme nástroj, který tekoucí data zaznamená
- tester spustí/vykoná a zaznamená testovací případy
- později může zaznamenané testovací případy spustit znovu (regresivní testování)

- komerční capture-replay nástroje pro zachycení/přehrání stisků kláves a pohybů myši, obsahující i nástroje pro zachycení a porovnání výstupů na obrazovku - používané pro GUI aplikace
- pro většinu ostatních účelů (testování zapouzdřených systémů apod.) je většinou nutné vytvořit si vlastní SW pro podporu testování

Defekty v číslech

- podle kvality vývoje asi 10 až 50 defektů na 1000 řádek kódu před testováním
- testy najdou obvykle méně než 60% defektů, proto se kombinují s inspekcemi
- „paradox“: defekty jsou častěji v testovacích případech než v testovaném kódu (McConnell 1993)
- pro kritické systémy - kombinace formálních metod vývoje, inspekcí a testování

Doba vývoje vs. počet defektů



Ladění

- testováním nalezneme defekty, následuje proces ladění (angl. debugging)
- ladění = identifikace příčiny defektu (90% času) a její oprava (10% času)
-
- v 5 krocích –
- stabilizace symptomu,
- nalezení příčiny,
- oprava,
- otestování opravy defektu,
- vyhledání obdobných defektů

- stabilizace symptomu
    - potřebujeme, aby se defekt projevoval spolehlivě, hledáme testovací případ, který symptom reprodukuje
    - testovací případ co nejvíce zjednodušíme, aby se defekt ještě projevoval (často již můžeme vytvářet hypotézu, proč defekt nastává)
    - defekty, které se neprojevují spolehlivě, (např. neinicializované proměnné, neplatný ukazatel, chyby časového souběhu), se snažíme zviditelnit (např. před spuštěním programu paměť zaplníme náhodnými hodnotami apod.)

- nalezení příčiny symptomu
    - zúžení podezřelé části kódu
    - použití ladícího programu nebo ladících výpisů - sledujeme, kde nastane symptom
    - dobře se vyspat

- oprava defektu
    - u nalezeného defektu bývá oprava poměrně jednoduchá,
    - vysoké nebezpečí zanesení dalšího defektu (podle některých studií více než 50%)
    - větší šanci provést opravu správně mají programátoři s celkovou znalostí programu, je třeba rozumět jak problému, tak opravovanému programu

- otestování opravy defektu
    - defekty je nutné opravovat po jednom, opravy po jedné otestovat
    - poté program otestovat jako celek (regresivní testy), ukáží se případné vedlejší efekty opravy
    - pro případ defektu v opravě je vhodné mít uchovánu předchozí verzi

- hledání obdobných defektů

Defenzivní programování

- zabezpečení definovaného chování při chybných vstupech, zamezení propagace chyb z podprogramu ven

- nástroje: kontroly vstupních parametrů
    - makro assert() v C, aserce v Javě), (programming by contract)
    - výjimky v Javě, C++, Delphi, Pythonu apod. - konstrukce typu try-catch a try-catch-finally:

```
try {
   foo();                                  // zde může nastat výjimka
} catch (SomethingWentWrongException e) {  // pokud nastane výjimka,
   System.out.println("some error");      //  provede se toto
} finally {                                // nakonec se vždy provede
   dispose();                              //  toto
}
```

- reakce na chybu
  - fce vrátí speciální návratový kód
  - programová výjimka, způsob propagace - podle stanovených konvencí
  - nastavení defaultní hodnoty vstupu nebo defaultního stavu

- o způsobu reakce na chybu se by mělo rozhodnout na úrovni architektury, aplikace by se měla ve všech svých částech chovat konzistentně

- během vývoje potřebujeme, aby defekty byly co nejlépe viditelné, ve výsledném produktu se naopak snažíme, aby defekty co nejméně rušily, a proto:

- ponecháváme kód, který kontroluje významné defekty; pokud aplikace hlásí interní chybu, měla by také oznámit, jakým způsobem jí uživatel může ohlásit
- „zrušíme“ kód testující nepodstatné defekty,
- „zrušíme“ kód, který způsobuje havárie,
- ponecháme kód, který umožní přijatelné ukončení aplikace při chybě (např. s uložením dat)

- rušení kódu neprovádíme fyzicky (při ladění ho budeme opět potřebovat), ale např. pomocí preprocesoru vynecháme tělo procedury Assert, využijeme verzování apod.