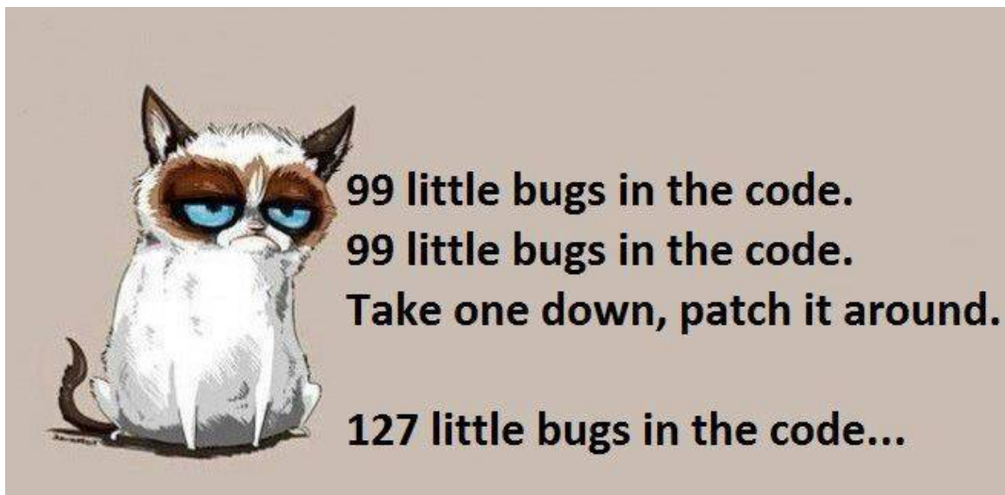
TESTOVÁNÍ SOFTWARE

Jaroslav Žáček
jaroslav.zacek@osu.cz
http://www1.osu.cz/~zacek/



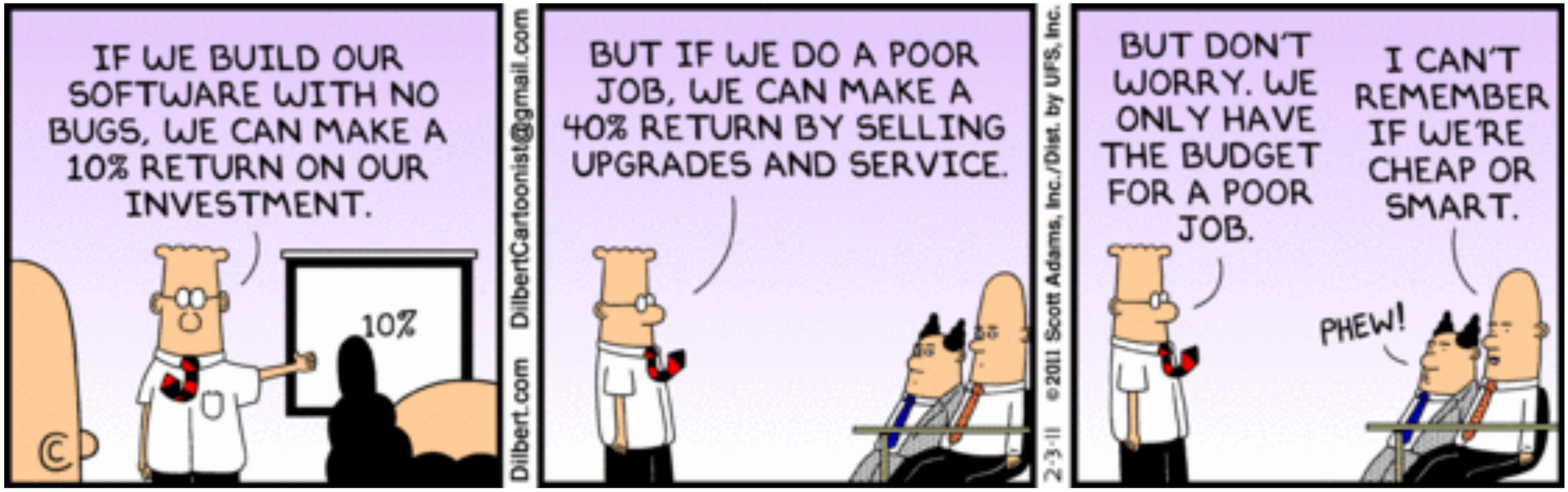
TESTOVÁNÍ

- Obsáhlá disciplína, existuje spoustu pohledů
- Problém při nastavení míry kvality

Kvalita: Schopnost objektu být použit.

Použit jak? Použít kým?

Chyba: Vlastnost objektu, která snižuje jeho kvalitu.


IF WE BUILD OUR SOFTWARE WITH NO BUGS, WE CAN MAKE A 10% RETURN ON OUR INVESTMENT.
10%
BUT IF WE DO A POOR JOB, WE CAN MAKE A 40% RETURN BY SELLING UPGRADES AND SERVICE.
BUT DON'T WORRY. WE ONLY HAVE THE BUDGET FOR A POOR JOB.
I CAN'T REMEMBER IF WE'RE CHEAP OR SMART.
PHEW!
DilbertCartoonist@gmail.com
Dilbert.com
2-3-11 ©2011 Scott Adams, Inc./Dist. by UFS, Inc.

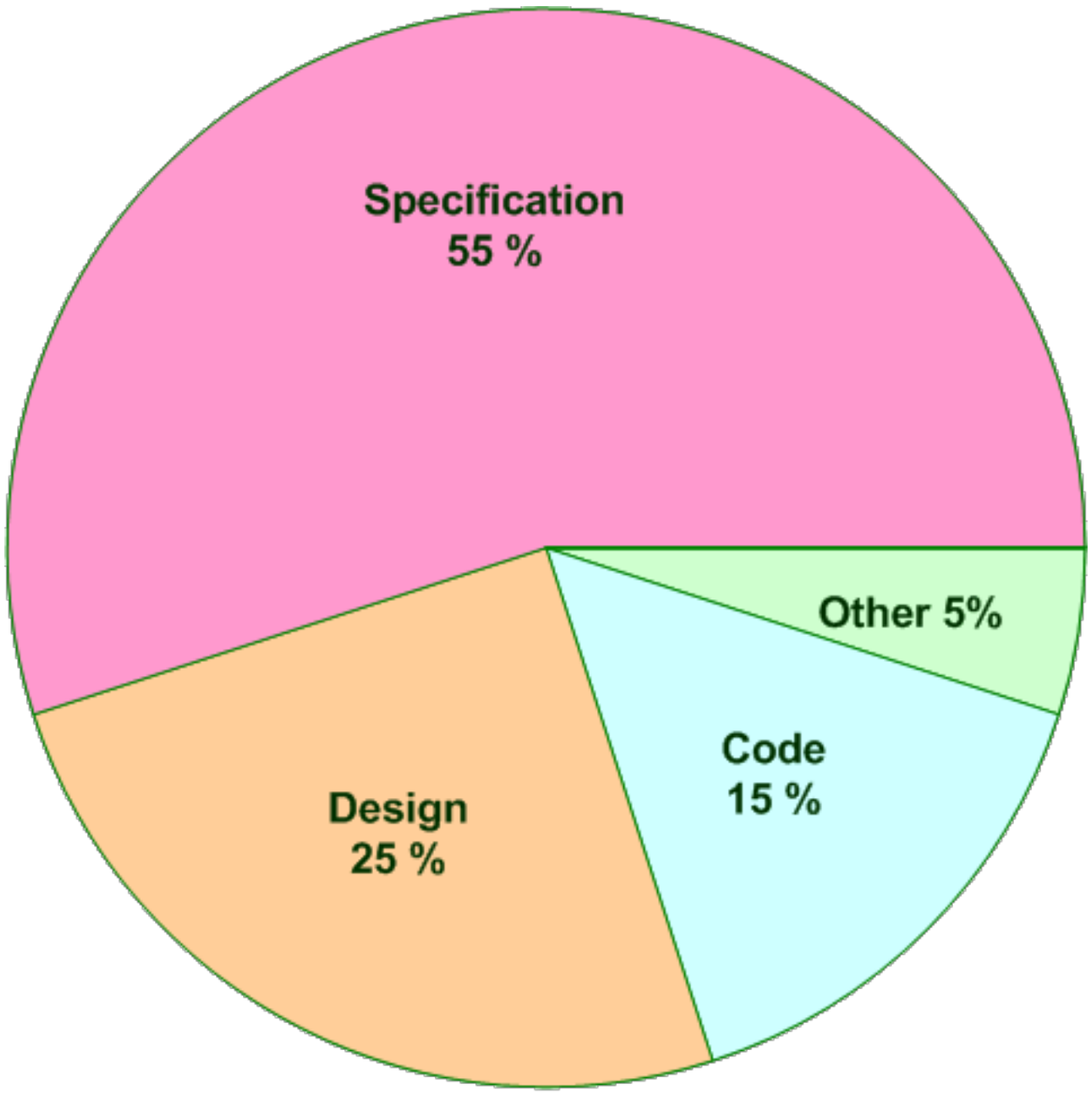
TESTOVÁNÍ

R. Patton: Cílem softwarového testera je vyhledávat chyby, vyhledat je co nejdříve a zajistit jejich nápravu.

Testování: Disciplína, která ověřuje kvalitu objektu tak, že hledá chyby.

KDE VZNIKAJÍ CHYBY V SOFTWARE?

KDE VZNIKAJÍ CHYBY V SOFTWARE?



Causes of Bugs

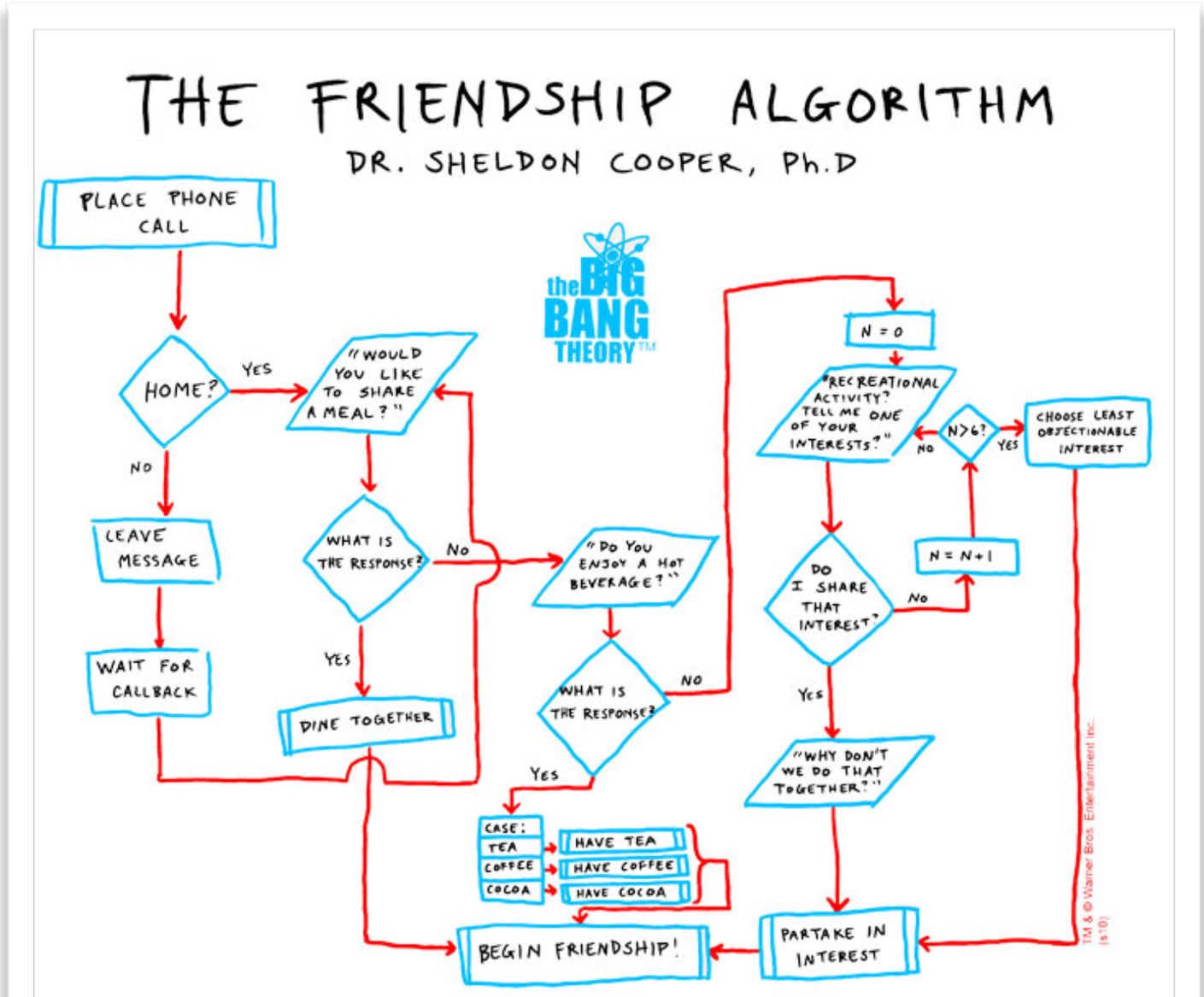
ZÁKLADNÍ ČLENĚNÍ TESTOVÁNÍ

- Black box - nevím a ani nechci vědět, co je uvnitř
- White box - mám povědomí o implementaci

- Statické - testujeme převážně dokumentaci
- Dynamické - testujeme software

ZÁPIS

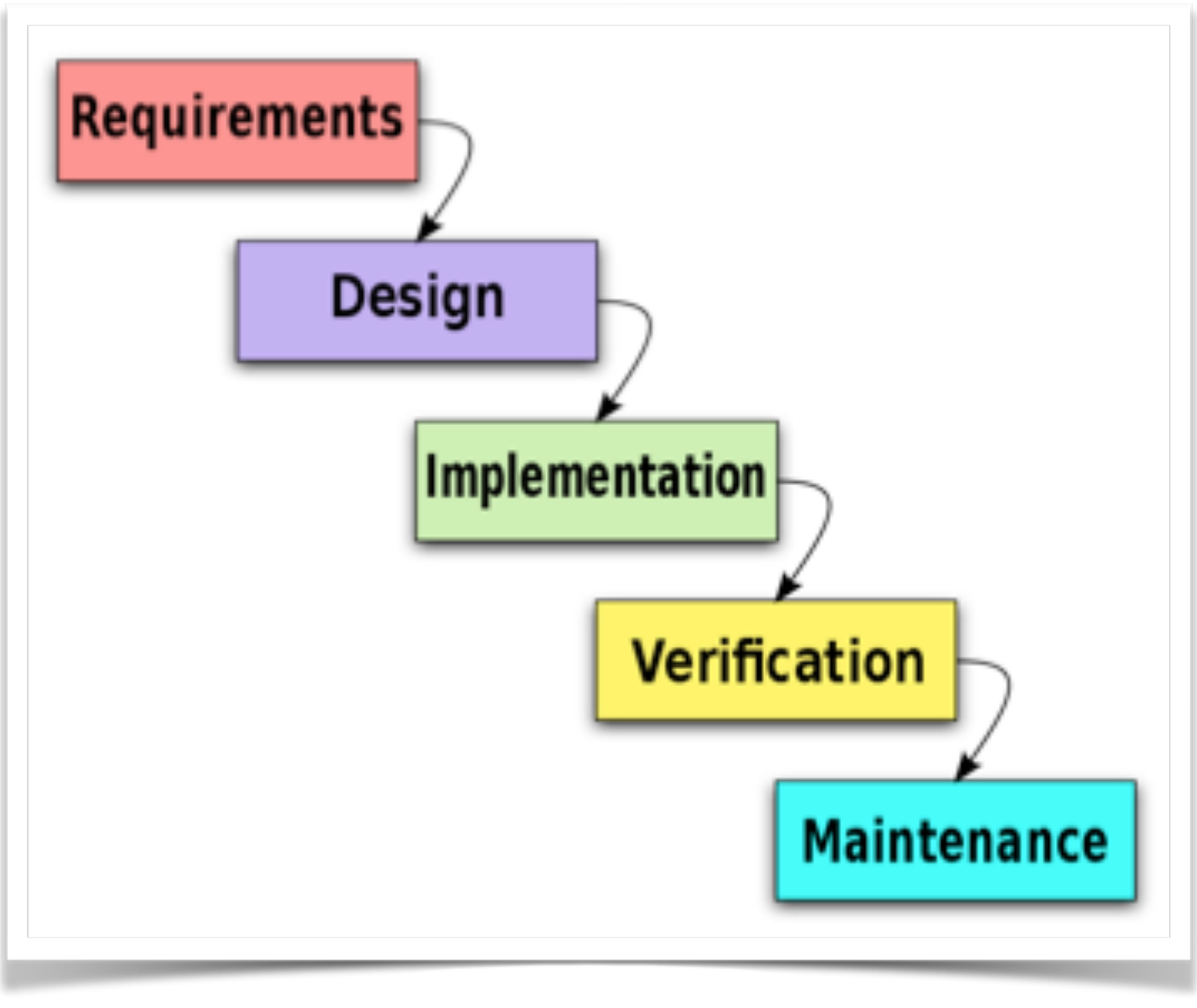
- IF **A** THEN **B** ELSE **C**
- Stačí nám zde dvouhodnotová logika?



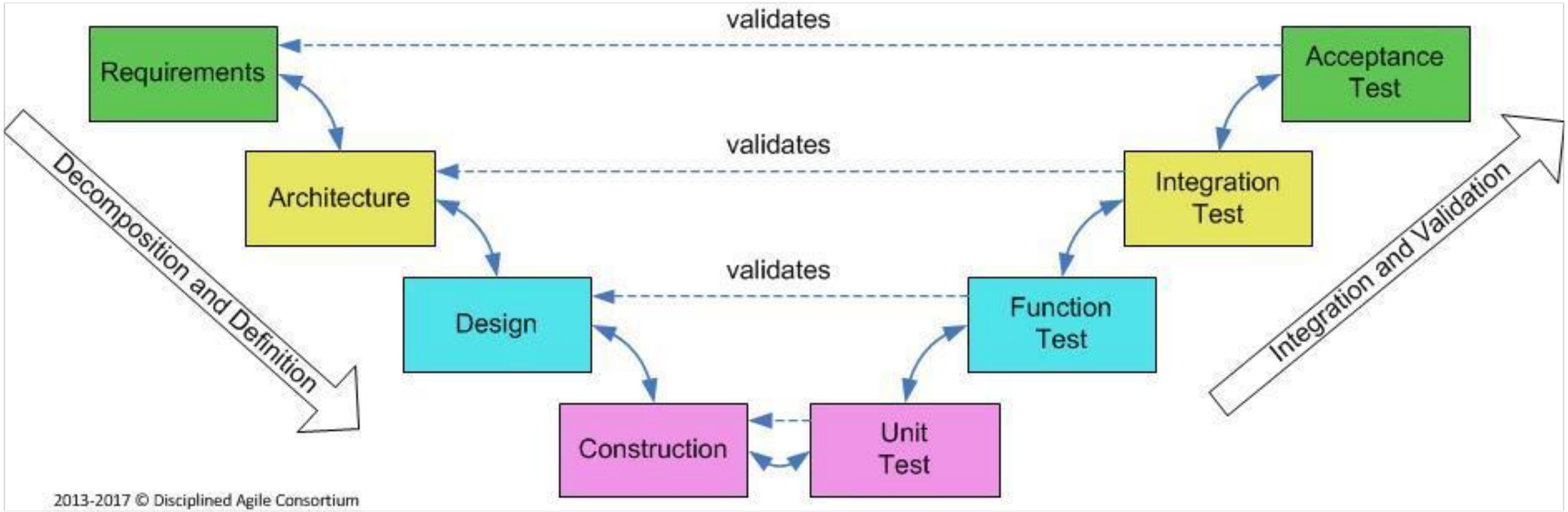
KDY JE VHODNÉ TESTOVAT?

- Ve všech fázích vývojového procesu?
- Všechno, nebo jen určité části aplikace?
- Testuje se i po dokončení vývoje?
- Testujeme již dříve otestované?
- Když se změní požadavky, co se stane se stávajícími testy?

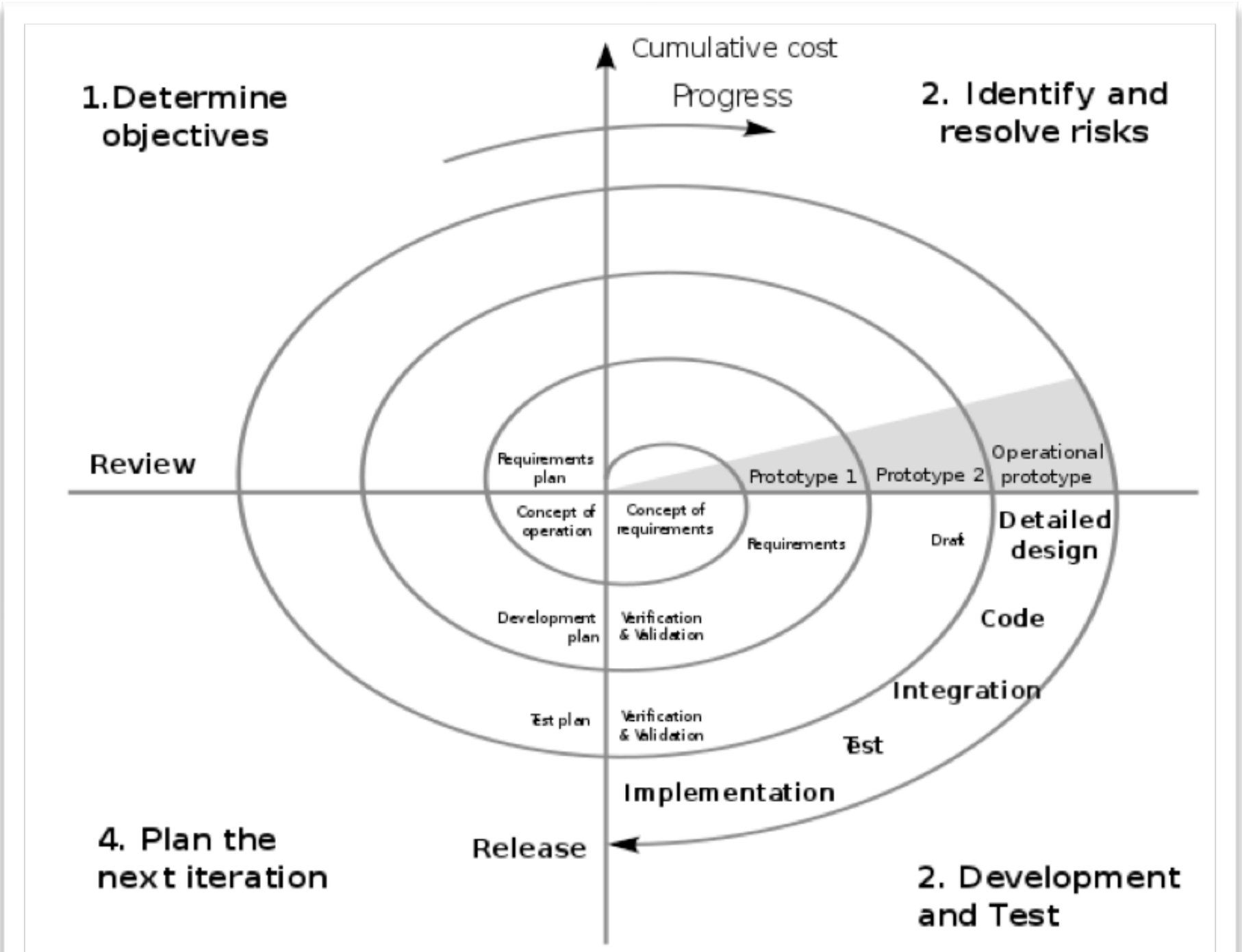
METODIKY VÝVOJE A TESTOVÁNÍ



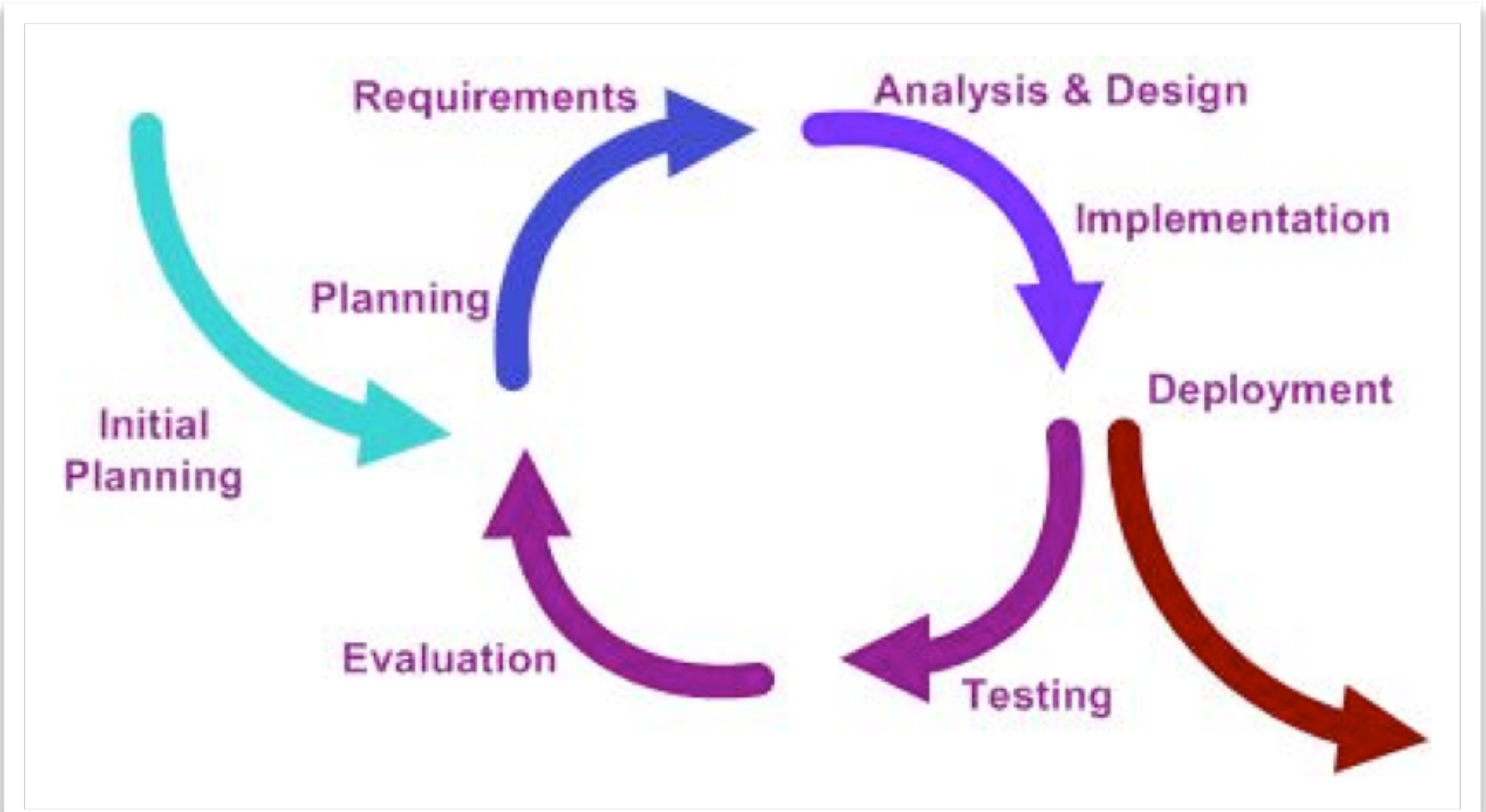
METODIKY VÝVOJE A TESTOVÁNÍ



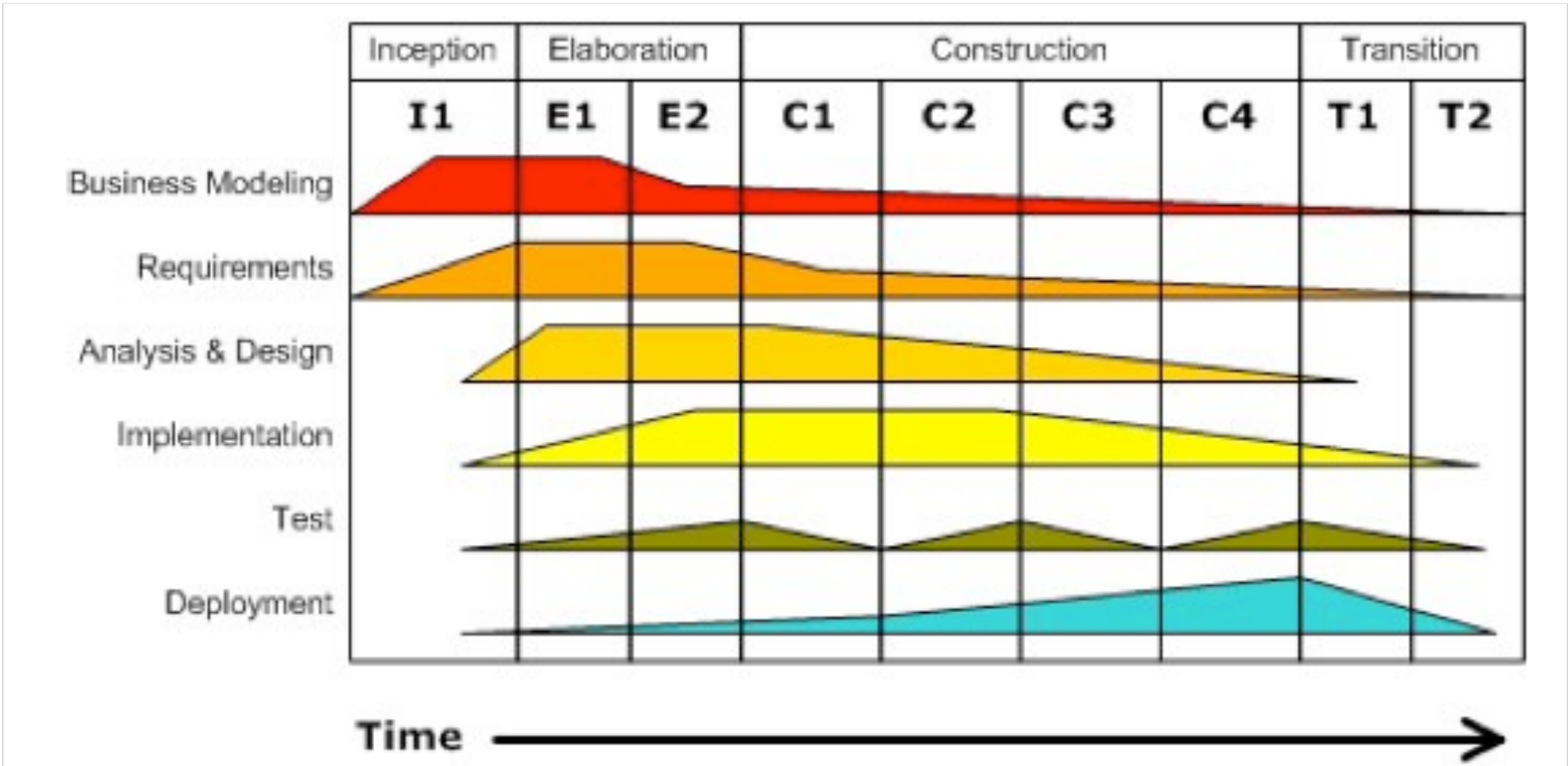
METODIKY VÝVOJE A TESTOVÁNÍ



METODIKY VÝVOJE A TESTOVÁNÍ



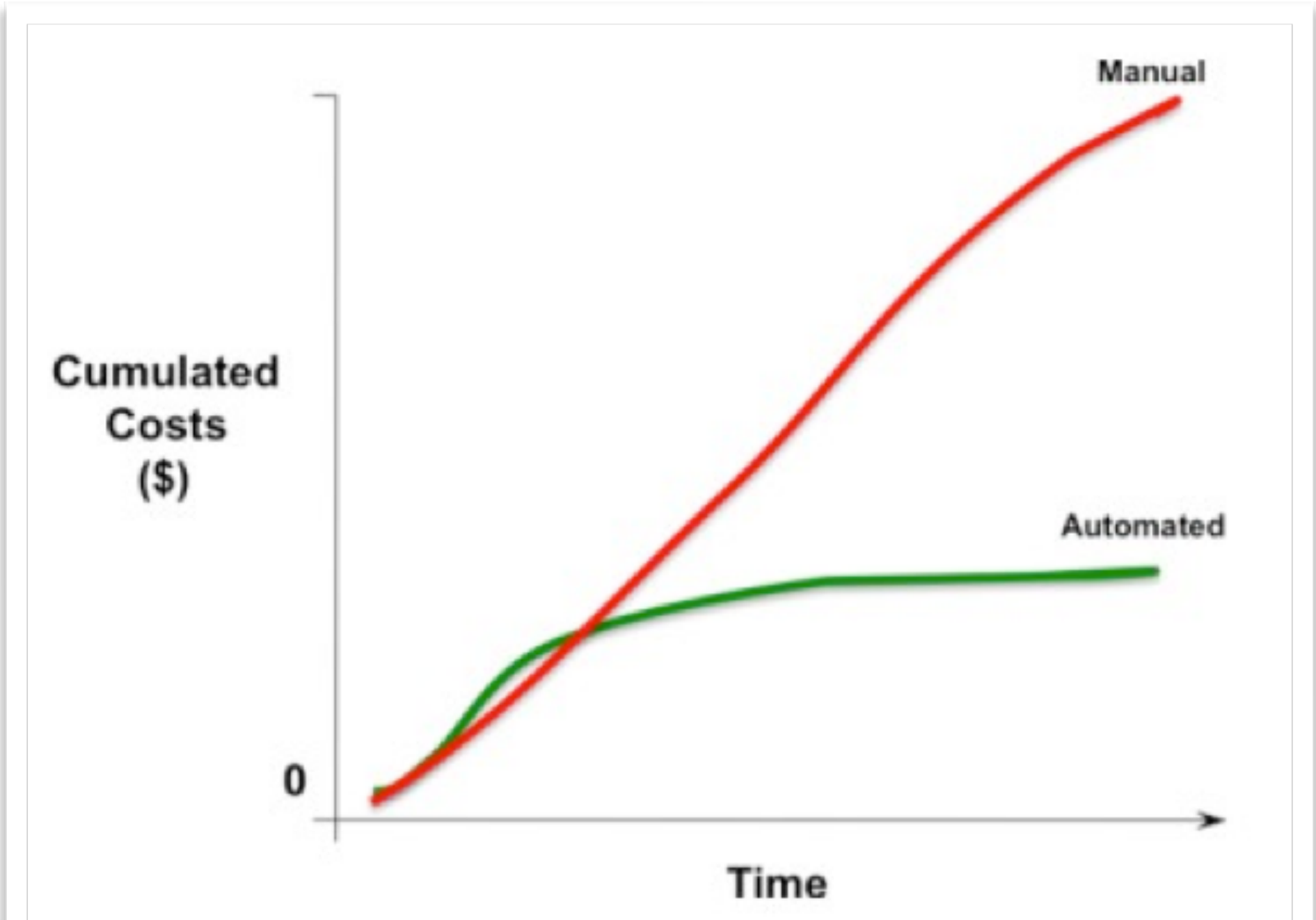
TESTOVÁNÍ V PRŮBĚHU VÝVOJE



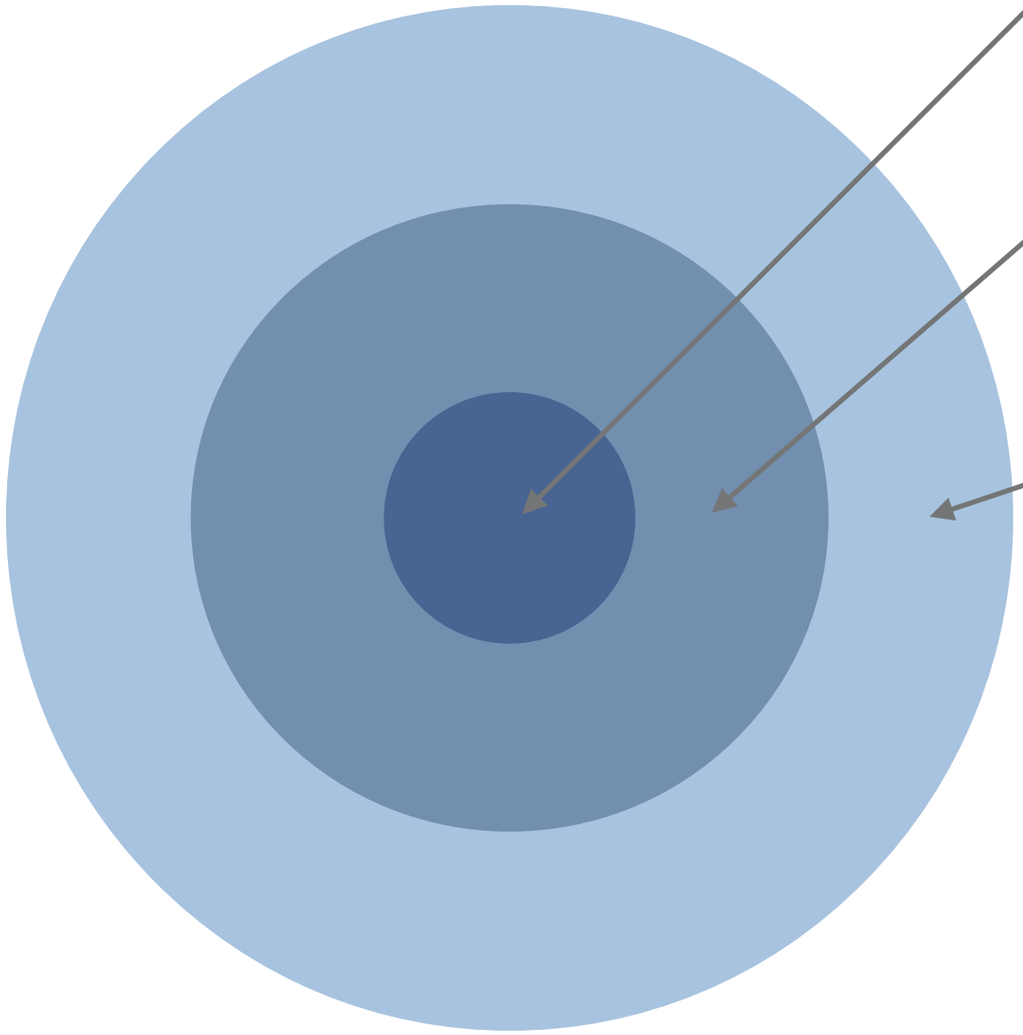
TYPY TESTŮ – AUTOMATIZACE

- **Manuální testy**
  - prováděny lidmi (interakce se systémem/GUI)
  - vizuální kontrola
  - kontrola lokalizace
    - typicky testy použitelnosti
- **Automatizované testy**
  - množina programového kódu a dat
  - ověřují získaná data oproti očekávaným hodnotám

TYPY TESTŮ – AUTOMATIZACE



TYPY TESTŮ



Unitové *testy - funguje mi ten algoritmus správně?*

Integrační *testy - fungují jednotlivé třídy a komponenty jako celek?*

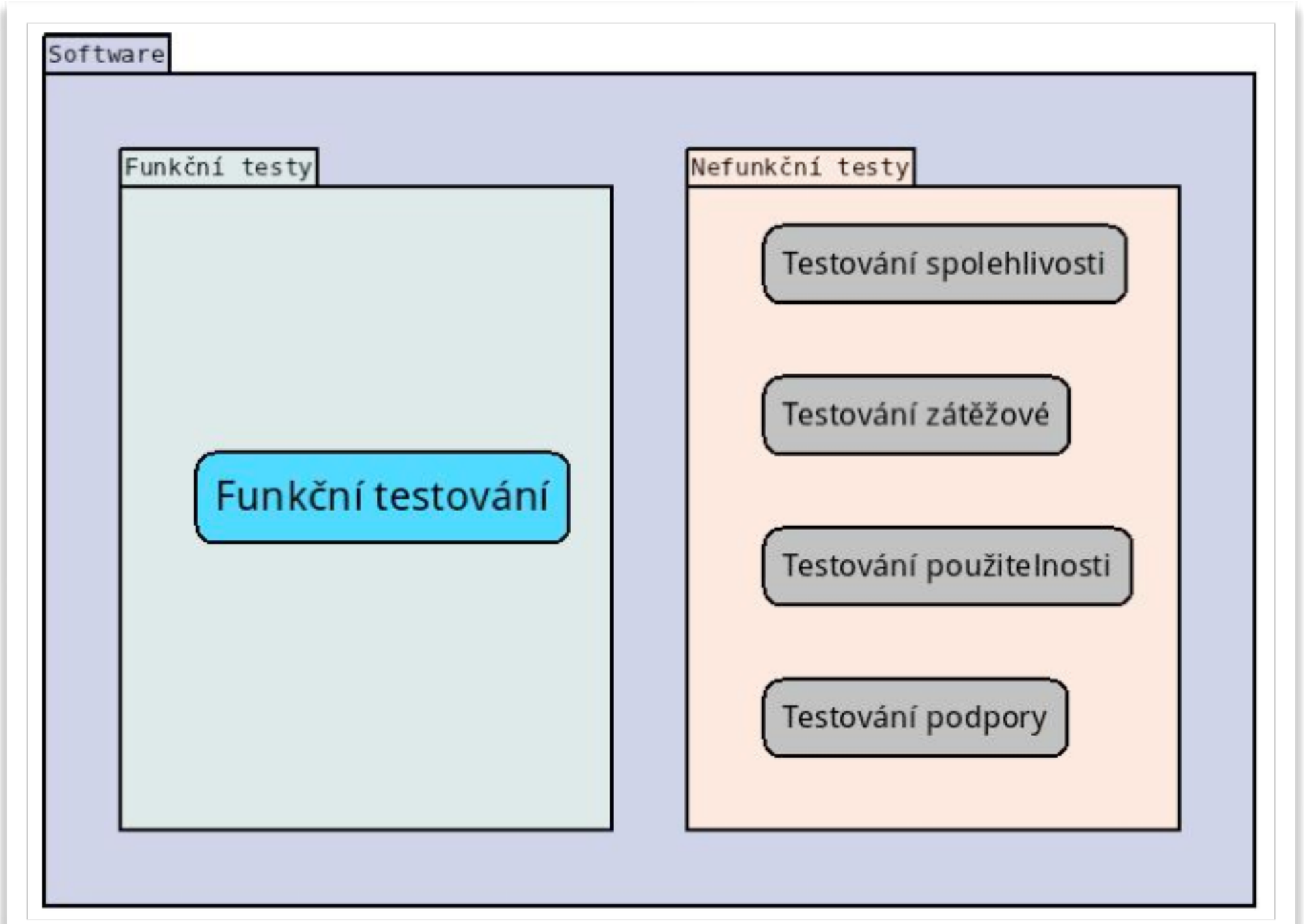
Akceptační *testy - funguje aplikace tak, jak to chce uživatel?*

UNITOVÉ VS INTEGRAČNÍ TESTY

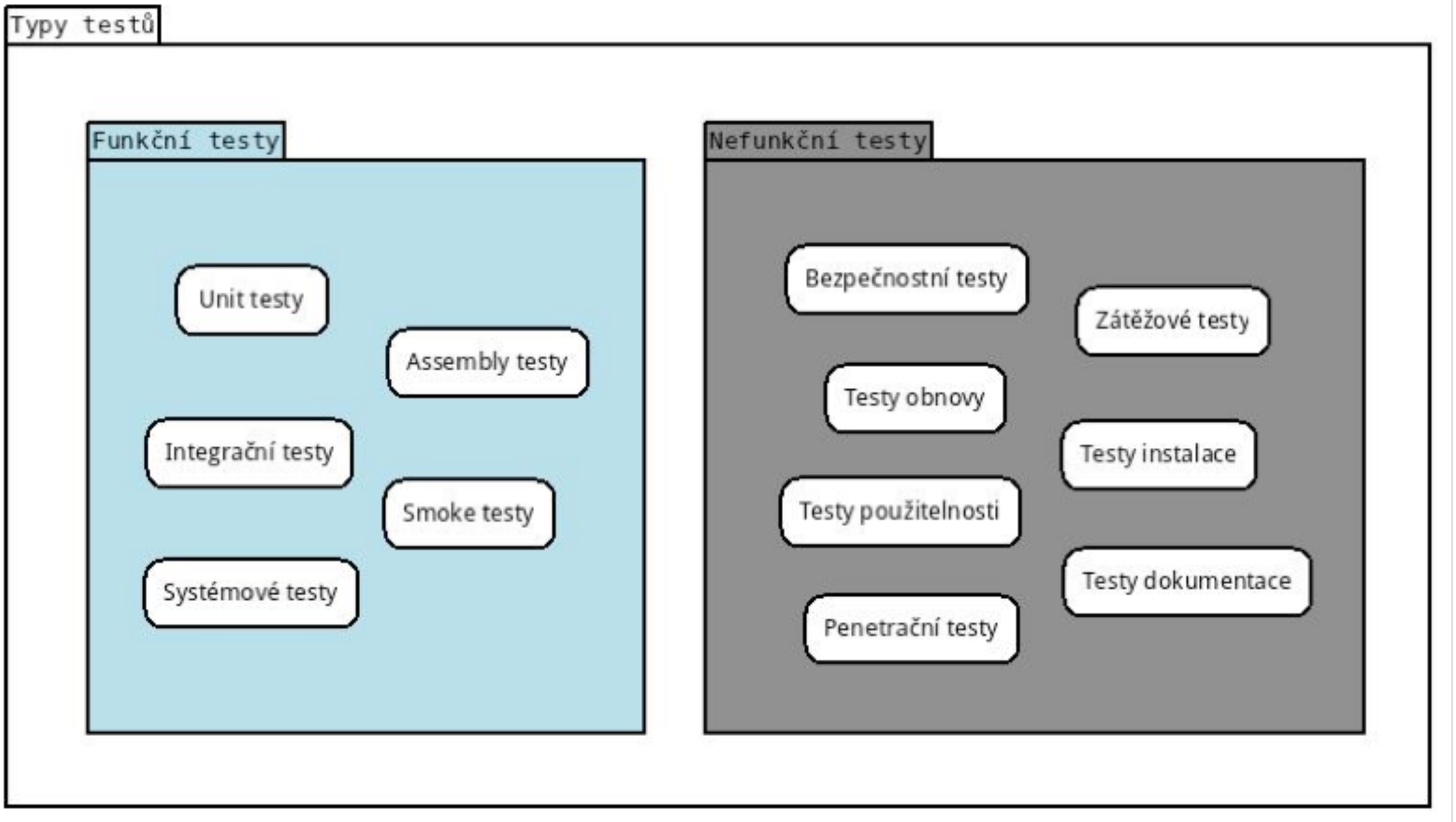
Unit test testuje pouze jednu vrstvu

Integrační testování testuje souhru více vrstev

TESTOVANÉ OBLASTI



TYPY TESTŮ


Typy testů
Funkční testy
Unit testy
Assembly testy
Integrační testy
Smoke testy
Systémové testy
Nefunkční testy
Bezpečnostní testy
Zátěžové testy
Testy obnovy
Testy instalace
Testy použitelnosti
Testy dokumentace
Penetrační testy

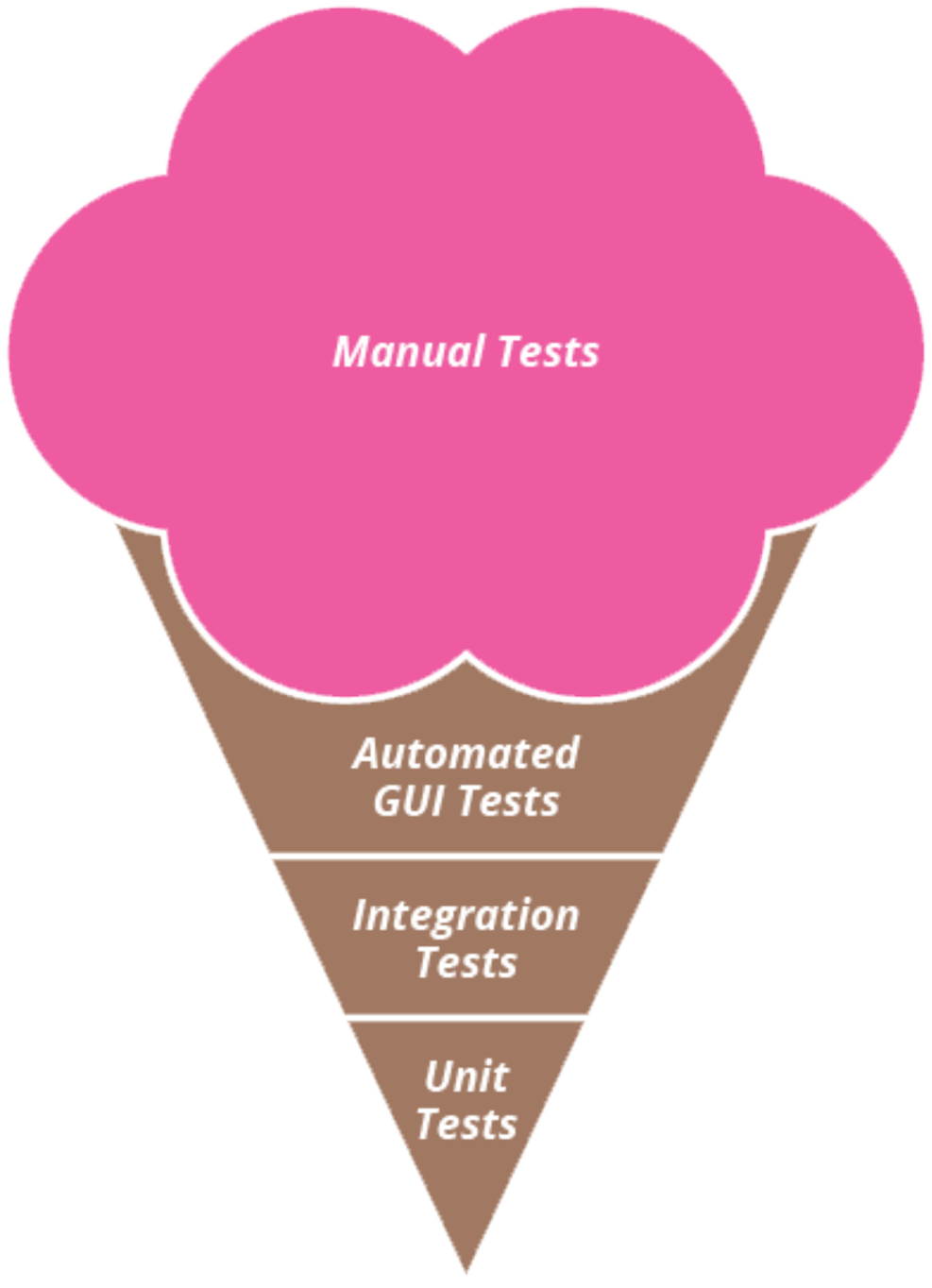
TESTOVÁNÍ A FUNKČNÍ POŽADAVKY

- Dle FURPS
  - Funkcionality
  - Usability
  - Reliability
  - Performace
  - Supportability

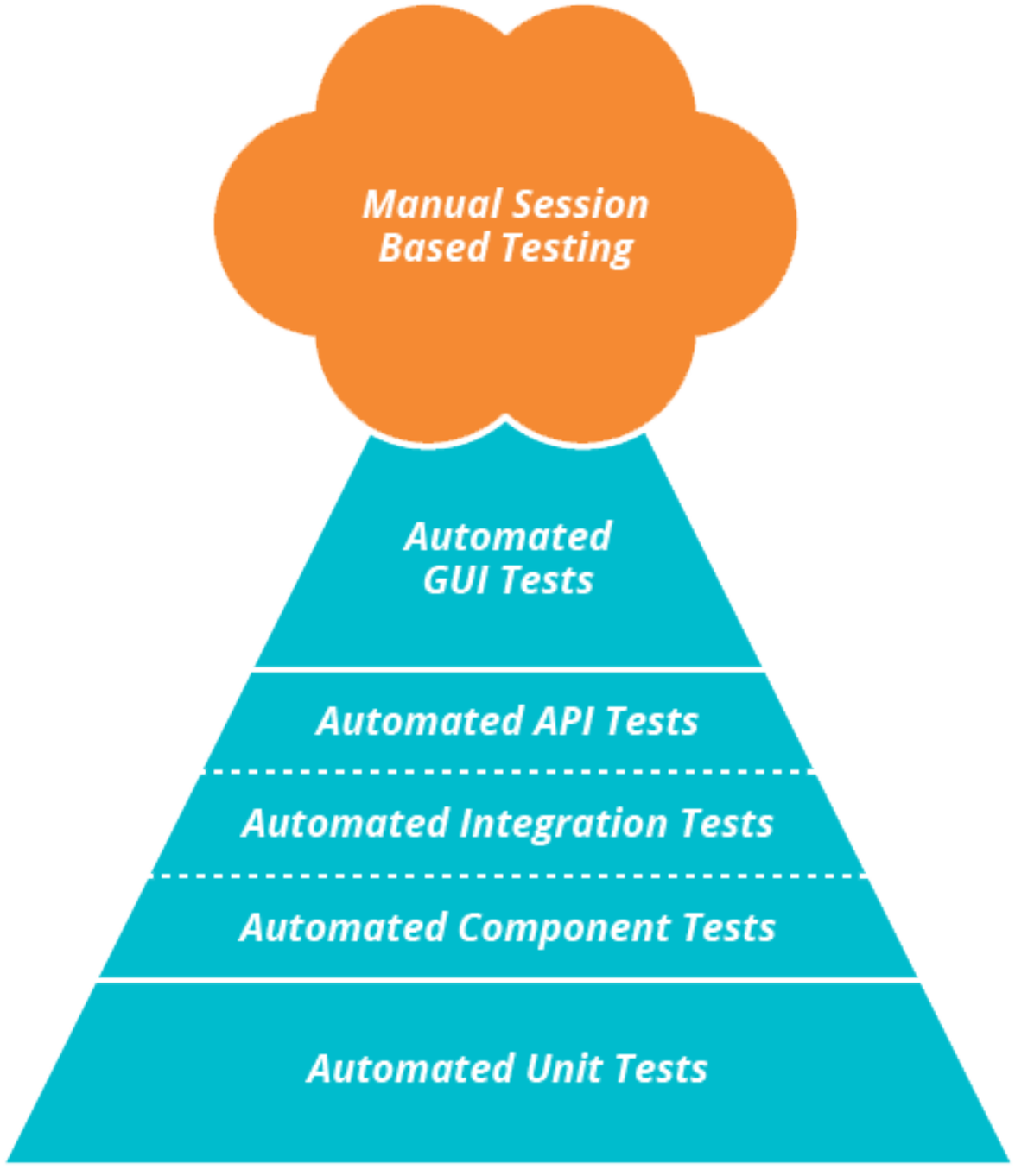
- ISO/IEC 12207
  - Hardwarové požadavky
  - Softwarové požadavky
  - Ostatní požadavky

Každé dělení má svůj účel, který slouží jiným typům testování

BAD PRACTICE



GOOD PRACTICE



STANDARD PRO TESTOVÁNÍ?

- ISO/IEC 29119
  - Concepts and Definitions (2013)
  - Test processes (2013)
  - Test documentation (dříve IEEE 829) (2013)
  - Test techniques (2015)
  - Keyword driven developmen (2016)
- http://softwaretestingstandard.org/

FORMALIZMUS ZÁPISU TESTU

- **Testovací případ** (Test Case)
  - Množina podmínek a hodnot, které určují, zda se software chová správně.
  - základní jednotka testování
  - vychází z Use Case, nejlépe ze scénáře
  - je definován atributy (specifikace vstupů a výstupů, požadavky na prostředí, jednoznačný identifikátor, popis testované oblasti)

Testovací scénář je množina testovacích případů slučujících je do logického sledu tak, aby bylo možné je provést a dávaly takto smysl.

TEST CASE

	Step	Description
Main Success Scenario **A: Actor** **S: System**	1	A: Inserts card
	2	S: Validates card and asks for PIN
	3	A: Enters PIN
	4	S: Validates PIN
	5	S: Allows access to account
Extensions	2a	Card not valid S: Display message and reject card
	4a	PIN not valid S: Display message and ask for re-try (twice)
	4b	PIN invalid 3 times S: Eat card and exit

ZÁKLADNÍ VLASTNOSTI TC

- **Atomicita** - má pouze jeden možný výsledek (prošel/neprošel).
- **Srozumitelnost** - je možné jej testovat bez dalších nezbytných vysvětlení (v případě manuálních testů)
- **Jednoznačnost** - detailně a jednoznačně popisuje prováděný test.
- **Jedinečnost** - neexistují duplicitní testovací případy

DÍLČÍ ČÁSTI

- **Testovací skript**
  - instrukce pro vykonávání testů
  - vykonávány automatizovaně nebo manuálně
- **Plán testů**
  - identifikuje strategie a zdroje, které budou při testování využity
- **Vyhodnocení testů**
  - souhrnná analýza výsledků, vytvořena alespoň jednou v rámci každé iterace

TEST SCRIPT

Test Script: Make Text Bold in Microsoft Word				
Step #	**Instruction**	**Expected Result**	**Actual Result**	**Comments**
1	Open up the Microsoft Word application.	Microsoft Word opens.	Pass	For this test, we are using MS Word 2003.
2	Create a new blank document.	A new blank document is displayed.	Pass	
3	Type the following sentence into the document: "This is what bold text looks like."	The sentence is displayed in the document.	Pass	
4	Highlight the word "bold".	The word "bold" is highlighted.	Pass	
5	Click the bold "B" button on the top toolbar.	The word "bold" is bolded.	Pass	

NEJČASTĚJI POUŽÍVANÉ TESTY

- **Testování jednotek** (Unit Testing)
  - nejmenší část systému
  - výsledek činnosti jednoho programátora
  - V C++ nebo v Javě je to třída, v C funkce
- **Integrační testy**
  - ověřuje vzájemnou spolupráci jednotek v rámci většího celku (subsystému, komponenty)

NEJČASTĚJI POUŽÍVANÉ TESTY

- **Testy systému**
  - projevují se na nejvyšší úrovni integrace
  - testují bezpečnost, spolehlivost, dostupnost, záloha, obnova ze zálohy
- **Akceptační testy**
  - stanovuje zákazník
  - posouzení, zda systém odpovídá potřebám uživatelů (pilotní testování, alfa verze, beta verze)

UNITOVÉ TESTOVÁNÍ

- Píší ho programátoři pro programátory.
- Testuje malou specifickou oblast zdrojového kódu (metodu, atribut - get/set metody, objekt).
- Výhody použití:
  - přehlednější kód, lepší architektura
  - důvěra v kód (dělá to co to má dělat)
  - usnadnění zavádění změn do aplikace

ŽIVOTNÍ CYKLUS UNIT TESTU

1. Nastavení (setup metoda)
2. Příprava vstupů
3. Volání metod (metod)
4. Kontrola výstupu
5. “Uklizení po sobě” - tear down

NÁSTROJE

- xUnit
  - Vytvořil Kent Beck (původně SUnit)
  - JUnit - testovací framework pro jazyk Java
  - NUnit - klon JUnit pro platformu .NET, funkčně identický
  - PHPUnit - napsán v PHP, inspirován SUnit

Kompletní list na

http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks

METODY PRO REALIZACI TESTOVÁNÍ

- fail(String Zpráva) - kontrola dosažitelnosti určité části kódu
- assertTrue(boolean)
- assertEquals(expected, actual)
- assertNull(object) - testuje null (objekt, atribut)
- assertNotNull(object)
- assertSame (expected, actual) - kontrola reference na objekt
- assertNotSame(expected, actual)

PŘÍKLAD – KALKULAČKA

```java
public class Calc {
 public long add(int a, int b) {
  return a+b;
 }
}
```

```java
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class CalcTest {
 @Test
 public void testAdd() {
  assertEquals(5, new Calc().add(2, 3));
 }
}
```

PŘÍKLAD – ÚČET

```
namespace Bank
{
  using NUnit.Framework;

  [TestFixture]
  public class AccountTest
  {
    [Test]
    public void TransferFunds()
    {
      Account source = new Account();
      source.Deposit(200m);

      Account destination = new Account();
      destination.Deposit(150m);

      source.TransferFunds(destination, 100m);

      Assert.AreEqual(250m, destination.Balance);
      Assert.AreEqual(100m, source.Balance);
    }
  }
}
```

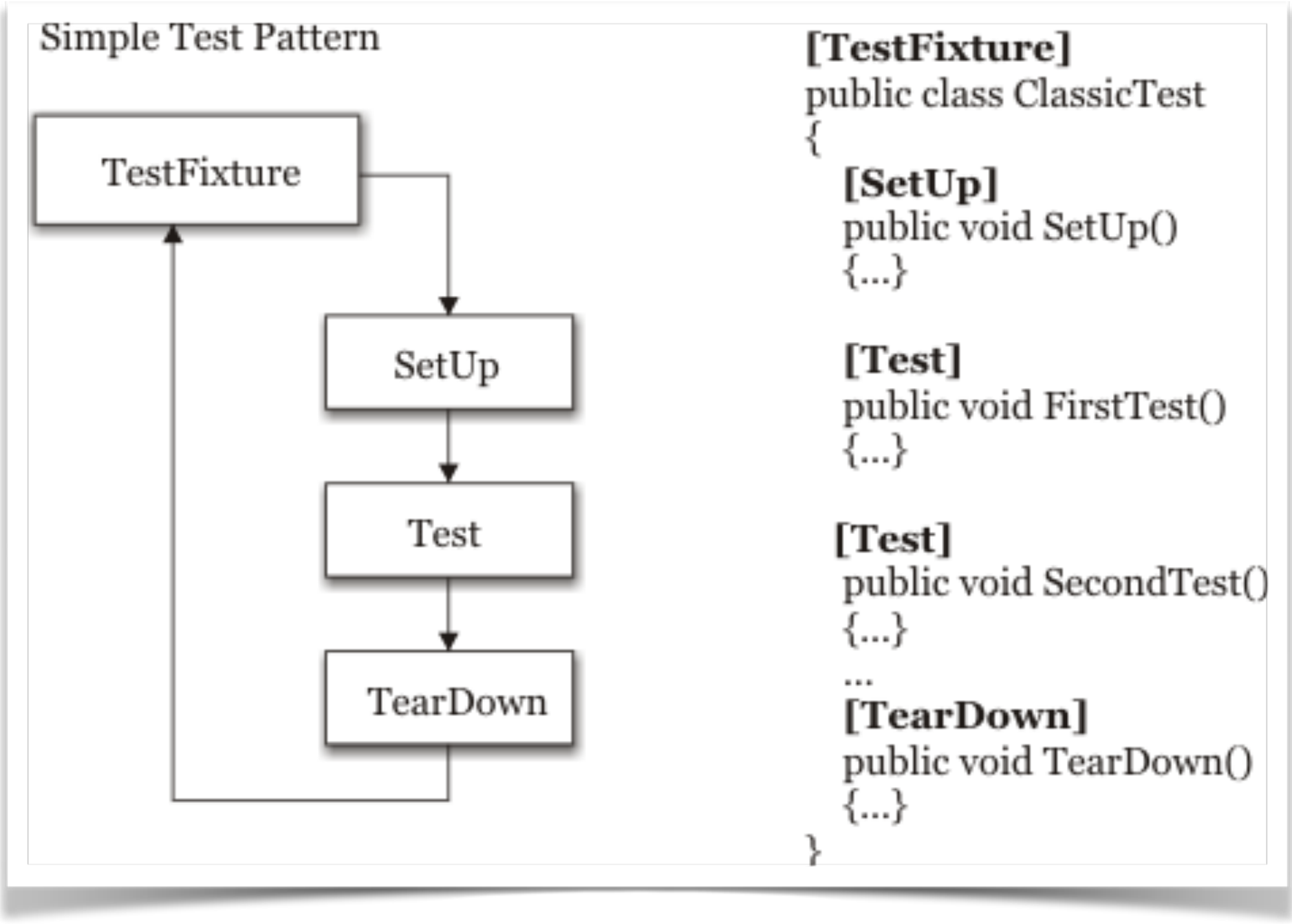
PŘÍKLAD – PRÁZDNÉ POLE

```php
<?php
require_once 'PHPUnit/Framework.php';

class ArrayTest extends PHPUnit_Framework_TestCase
{
    public function testNewArrayIsEmpty()
    {
        // Create the Array fixture.
        $fixture = array();

        // Assert that the size of the Array fixture is 0.
        $this->assertEquals(0, sizeof($fixture));
    }

    public function testArrayContainsAnElement()
    {
        // Create the Array fixture.
        $fixture = array();

        // Add an element to the Array fixture.
        $fixture[] = 'Element';

        // Assert that the size of the Array fixture is 1.
        $this->assertEquals(1, sizeof($fixture));
    }
}
?>
```

TŘI ČÁSTI TESTU

```
@Test
public void shouldCalculateSum() {

 Calculator calculator = new Calculator()    //Arrange

 Integer result = calculator.sum(1, 3);      //Act

 assertEquals(result, 4);                    //Assert

}
```

TEST FIXTURE



MŮŽU TYTO KÓDY OTESTOVAT JUNIT?

```
public Calendar getTomorrow() {
    Thread.sleep(1000*60*60*24);
    return Calendar.getInstance();
}
```

```
A a = new A();
if (a != null) ...
```

```
if(currValue == null){
            file.setProperty(key, value);
        }else{
            file.setProperty(key, value);
        }
```

```
import org.apache.commons.lang.math.RandomUtils;

...
...
...

public static int randomInt(int from, int to) {
        int result;
        do {
            result = RandomUtils.nextInt(to);
        } while (result < from);
        return result;
    }
```

POŽADAVKY NA "DOBRÝ" UNIT TEST

- Konzistence
- Atomicita
- Jednoznačná odpovědnost
- Jednoduchý na pochopení
- Žádné složité rozhodování a cykly

KONZISTENCE

- Vícenásobné spuštění testu by vždy mělo vracet stejný výsledek.
- Testování by nemělo mít vliv na vnitřní stav testovaného systému.

Problémový kód:

```
import java.util.Date;
Date d = new Date();
```

ATOMICITA

- Výsledek testu je vždy **PASS** nebo **FAIL**.
- Neexistují částečně splněné testy.
- Izolovanost jednotlivých testů - spuštění testu **B** nesmí mít vliv na výsledek testu **A**.

JEDNOZNAČNÁ ODPOVĚDNOST

- Jeden test by měl být zodpovědný pouze za jeden scénář.
- Testují se scénáře, nikoli metody:
  - Jedna metoda, více možných scénářů -> více testů
  - Jeden scénář, více metod -> jeden test

JEDNODUCHÝ NA POCHOPENÍ

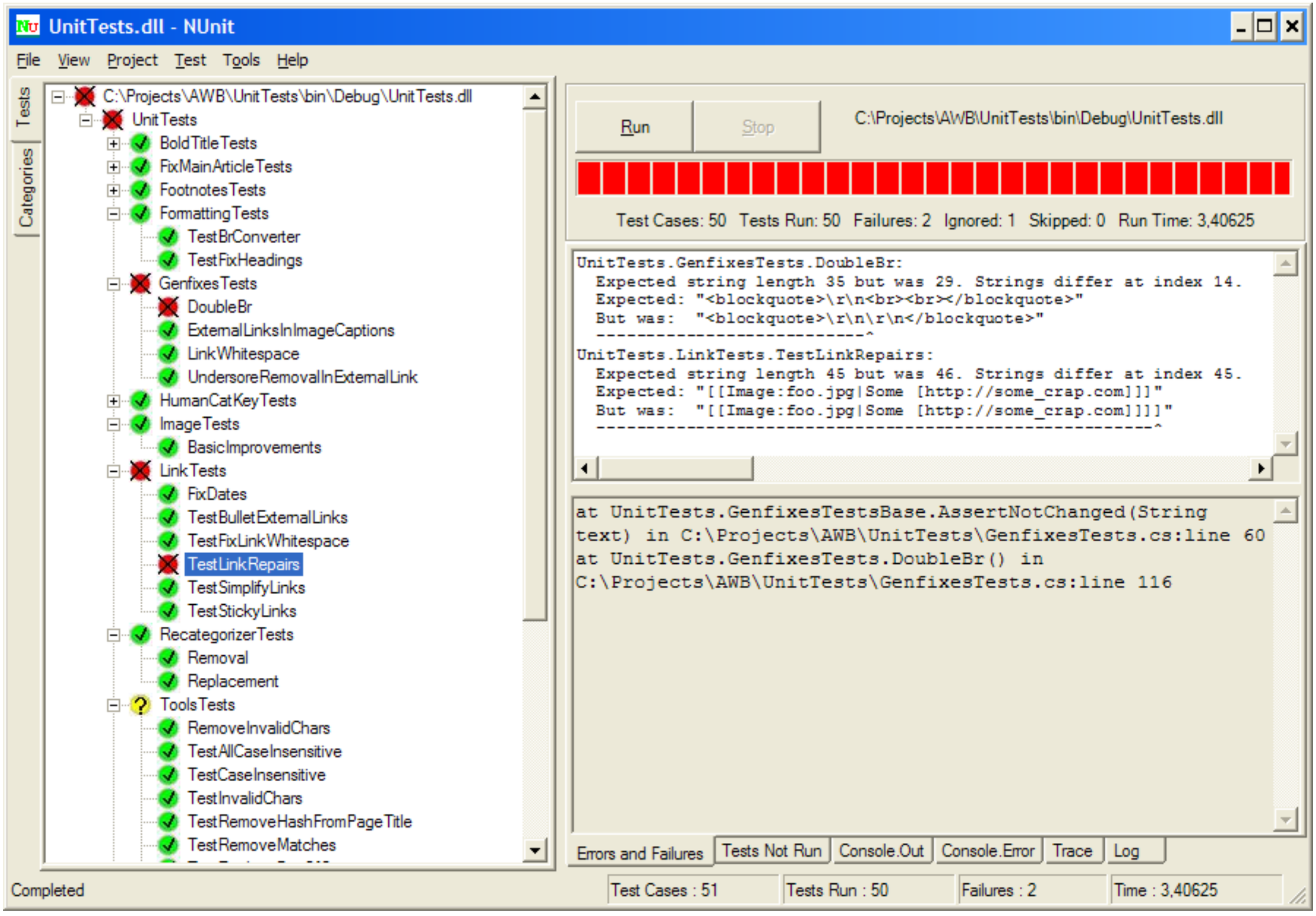
- Stejně jako na zdrojový kód se i na unit testy vztahují konvence kódu
  - jména proměnných (int a, int b, String c…)
  - jména metod
  - jména tříd
  - minimum podmíněné logiky
  - minimum cyklů
- Testy pojmenovávejte podle úspěšných scénářů
  - UzivatelVytvoriRezervaci()
  - VlozCastkuNaUcet()

ROZHODOVÁNÍ A CYKLY

- Testy by neměly obsahovat neurčitosti
  - Měly by být známy a definovány všechny vstupy
  - Chování metod by mělo být předvídatelné
  - Očekávaný výsledek by měl být striktně definován
    - Místo použití IF a CASE raději rozdělit problém do více testů
- Test by neměl obsahovat cykly
  - Pokud je potřeba testovací logiku opakovat, zřejmě je test příliš komplikovaný
  - Místo použití cyklu uvnitř testovací metody je možno test spustit vícekrát

Další tipy: https://examples.javacodegeeks.com/core-java/junit/junit-best-practices/

NUNIT – VÝSTUP



KONVENCE NEBO ANOTACE?

- **Konvence** - jméno třídy a metod musí mít shodný prefix
  - nepřehledné, se vzrůstající komplexností hůře udržitelné
  - každý framework může mít vlastní definované konvence
- **Anotace** - nahrazují jmenné konvence
  - nutná podpora frameworku, správná verze Javy

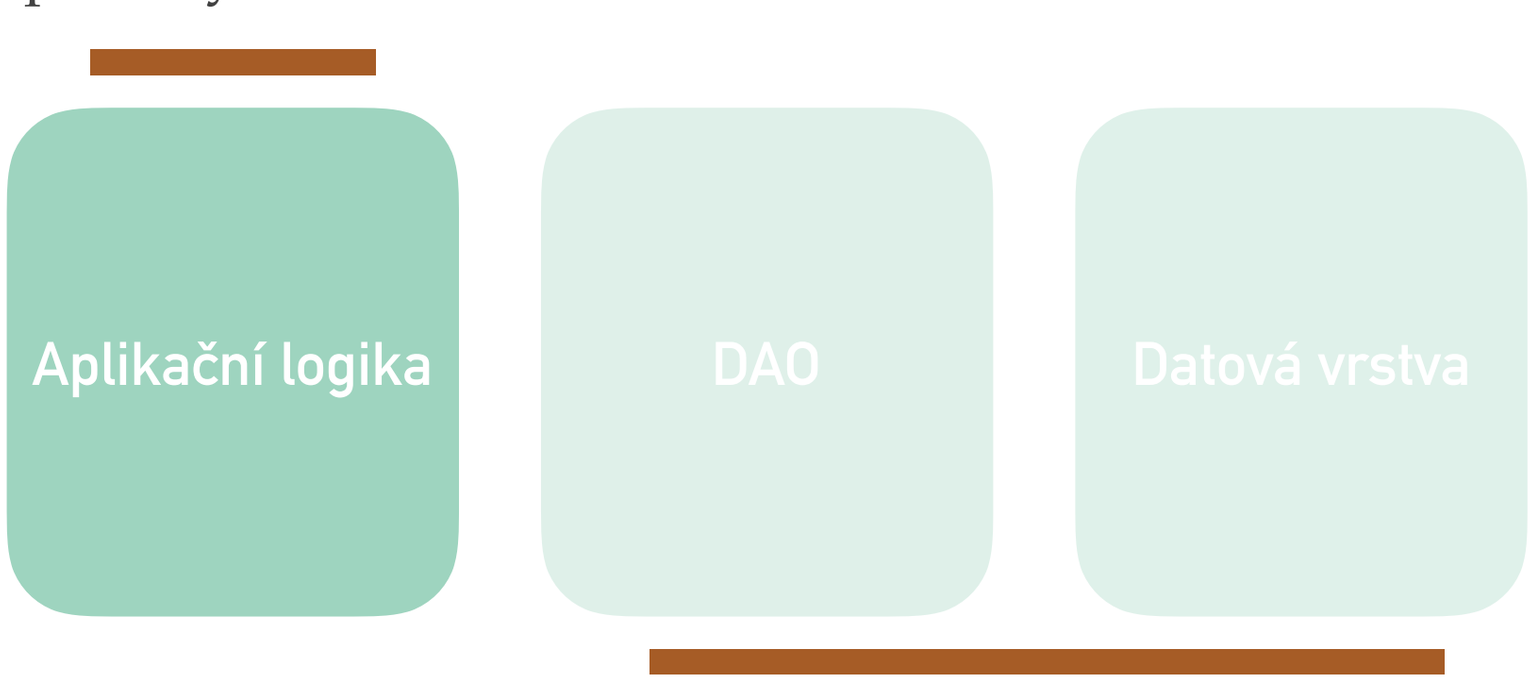
JUNIT ANOTACE

- **@Test** - označuje testovanou metodu
- **@Test(expected=IndexOutOfBoundsException.class)** - testovaná metoda s očekávanou výjimkou
- **@Before** - metoda spuštěná před každým testem (poskytnutí vstupních dat)
- **@After** - metoda spuštěná po skončení každého testu (uvolnění paměti)
- **@BeforeClass** - metoda spuštěná před prvním testem (připojení do DB)
- **@AfterClass** - metoda spuštěná po skončení testování (zavření spojení do DB)

JAKÉ NASTÁVAJÍ PROBLÉMY S TESTOVÁNÍM V OOP PARADIGMATU?

Unitové testování s Mockováním

Jednotlivé objekty jsou na sobě závislé.
Některé metody nemají návratovou hodnotu.



Mockované objekty zajišťují, aby byl unit test izolovaný

CO TO JE MOCK OBJEKT

- Vytvořen mockovacím frameworkem.
- Implementuje stejné rozhraní jako původní objekt.
- Poskytuje testovacímu frameworku veškeré nutné závislosti pro otestování závislého objektu.
- Umožňuje nám ověřit, jestli testovaná metoda používá závislé objekty korektně.

CO JE DOBRÉ POUŽÍT?

- Vytváří MOCK objekty
- Ověří počty volání a předané parametry
- Může sledovat a ovlivňovat existující objekty
- Umožňuje konfiguraci chování mocku

PŘÍKLAD POUŽITÍ

```
public class ShippingServiceTest {

    private InventoryService inventoryService = createNiceMock(InventoryService.class);

    private Logger logger = createNiceMock(Logger.class);

    private PricingService pricingService = createNiceMock(PricingService.class);

    private TrackingRepository trackingRepository = createNiceMock(TrackingRepository.class);
    private ShippingService shippingService;

    @Before
    public void createShippingService() throws Exception {
        shippingService = new ShippingService(pricingService, inventoryService);
        shippingService.setTrackingRepository(trackingRepository);
        shippingService.logger = logger;
    }
}
```

E2E

- Testuje celou aplikaci - chování z pohledu uživatele.
- Často jsou prováděny manuálně.
- Závisí na typu aplikace
  - desktopová
  - webová
  - REST rozhraní

SELENIUM

- Způsob jak automatizovaně testovat webovou aplikaci
- Využitelné pro téměř každou webovou aplikaci.
- Pro napsání testů můžeme použít Javu, C#, Ruby, Python, Javascript.
- …nebo můžeme test case naklikat pomocí pluginu do Firefoxu

SELENIUM WEBDRIVER

- Slouží pro realizaci maker v prostředí prohlížeče (simuluje prohlížeč)
  - Chrome, IE, Opera, Edge, PhantomJS

UKÁZKA KÓDU

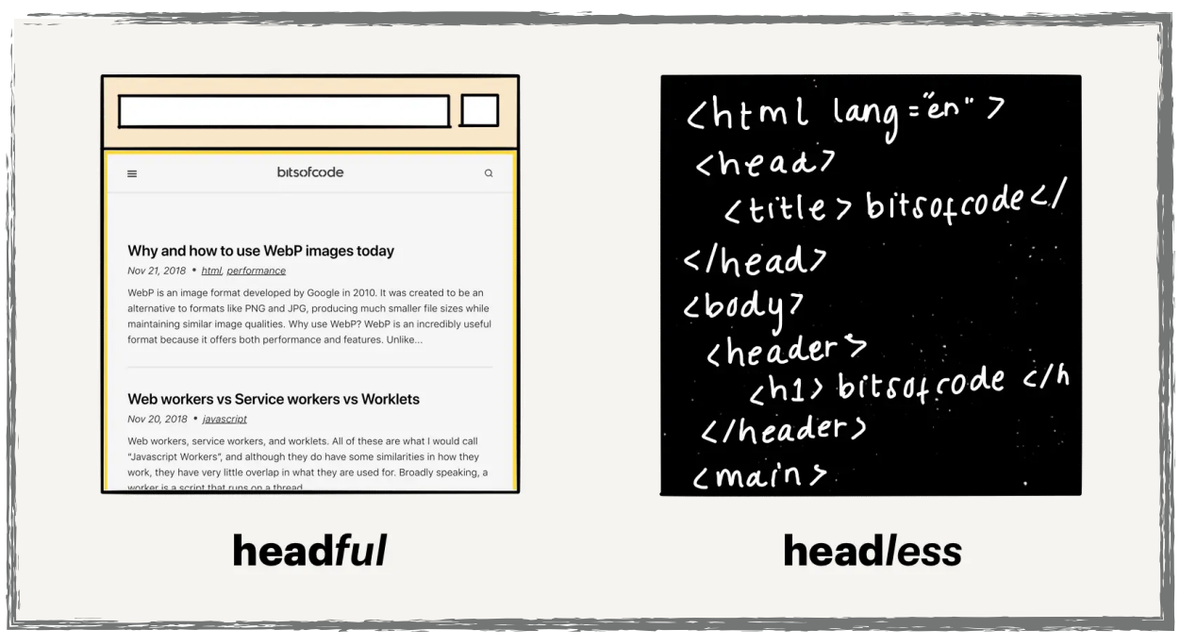
```
// přechod na testované url
driver.get(“http://library.cz/books/create");

// nalezení elementu
WebElement bookName = formElement.findElement(By.id(“book-form"));

// interakce s elementy
bookName.clear();
bookName.sendKeys("Guide to galaxy");
formElement.submit();
WebElement title = driver.findElement(By.cssSelector("div#detail > h2”));

// kontrola výsledku
assertThat(title.getText()).isEqualTo("Guide to galaxy");
```

TESTOVÁNÍ SPA WEBOVÝCH APLIKACÍ

- **Jasmine**
  - obecný framework pro testování JavaScriptu
  - podobný přístup jako Unit
  - použitelný pro všechny JS frameworky (Angular, React, Vue, …)
  - poznáte ho podle klíčových slov “it” a “expect” v kódu
- **Karma**
  - automatizace pro spouštění testů - test runner
  - Nastaví se framework (např. Jasmin), prohlížeč (Chrome, Firefox,…), kde jsou jednotlivé testy a poté Karma vše spustí a vyhodnotí

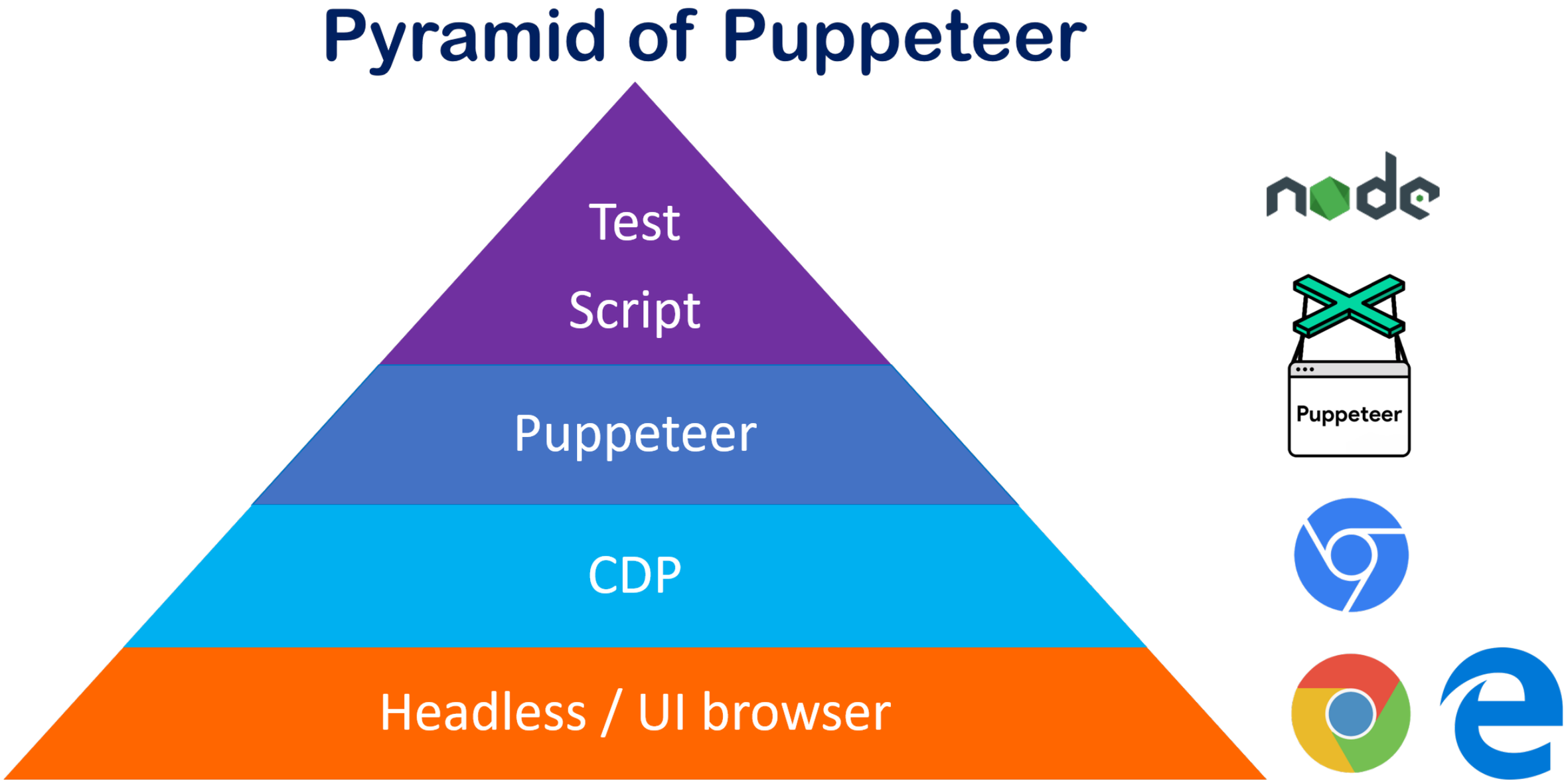
TESTOVÁNÍ BEZ PROHLÍŽEČE – HEADLESS

- Ne vždy musím nutně vidět okno prohlížeče, zajímá mě jen výsledek testu. Bez zobrazení prohlížeče je test rychlejší.
- Framework na pozadí spustí všechny služby prohlížeče až na vlastní UI (nevykresluje DOM, pouze ho sestaví).
- Dá se nastavit, aby při chybě i po ukončení testu framework zachytil screenshot výsledku.

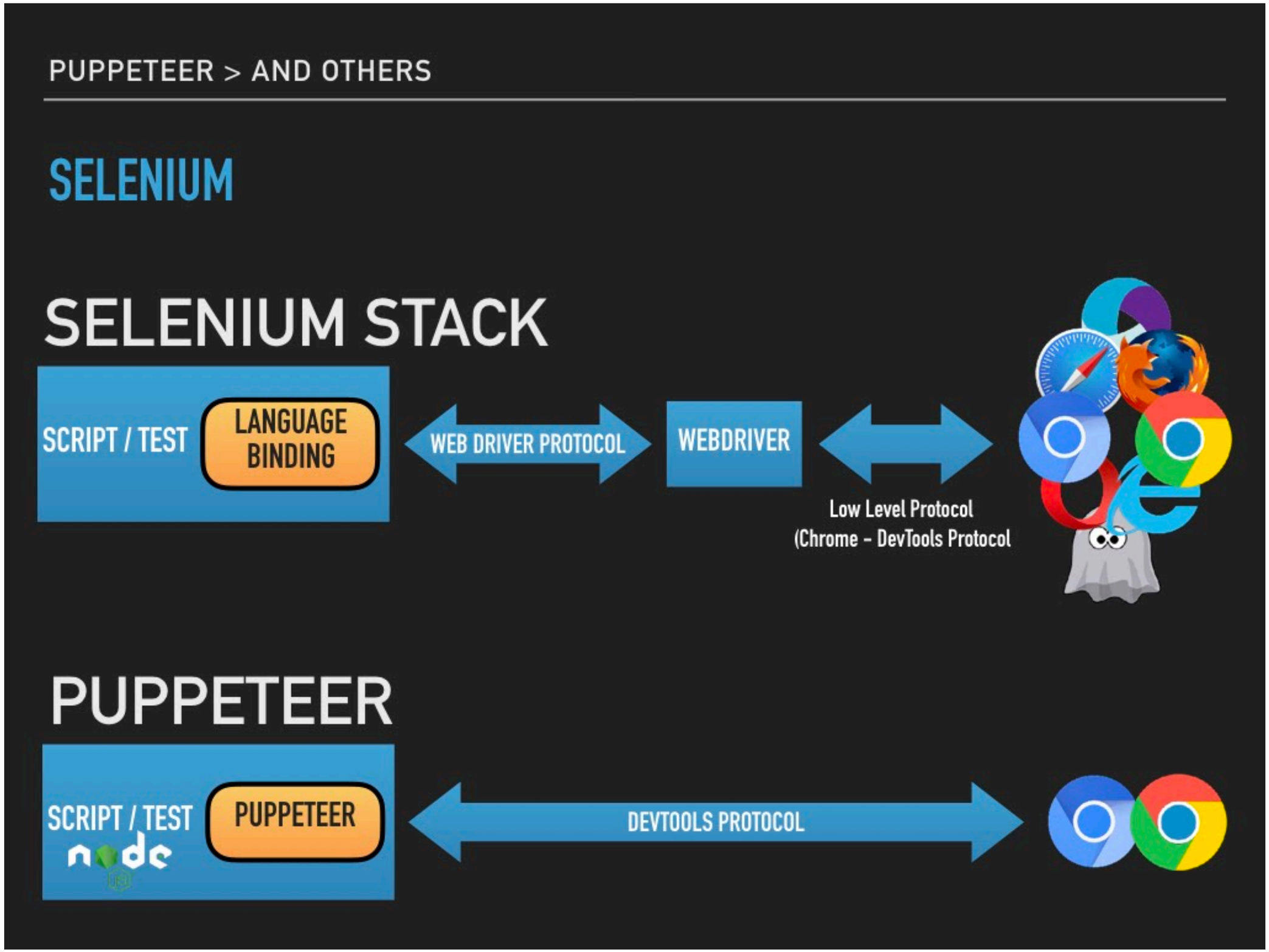


TESTOVÁNÍ BEZ PROHLÍŽEČE – HEADLESS

- Dříve PhantomJS, CasperJS, SlimerJS
- Nyní Puppeteer: https://youtu.be/lhZOFUY1weo

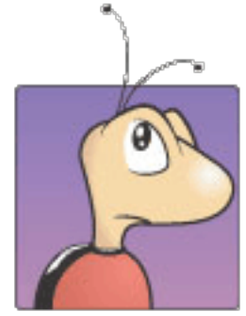


SELENIUM VS. PUPPETEER



SOFTWARE PRO CHYBOVÉ ŘÍZENÍ

- Bugzilla (Perl + MySQL)
- MantisBT (PHP + MySQL)
- Jira (zaměřen hlavně na projektové řízení)
- Trac
- Redmine (zaměřen hlavně na projektové řízení)



mantis BUG TRACKER

trac Integrated SCM & Project Management

REDMINE flexible project management