Test driven development

Test-Driven Development (TDD) je vývojový přístup, který se zaměřuje na psaní testů před implementací kódu. Tyto testy jsou typicky napsány pomocí testovací frameworků jako JUnit pro Java, pytest pro Python nebo NUnit pro C#. Tím je zaručována jeho správnost a funkčnost. Již napsaný kód je v průběhu vývoje znovu testován pro zaručení kompatibility s novým kódem.

Vývoj tímto způsobem se dělí na jednotlivé fáze:

1. **Založení seznamu testů pro novou funkci:**
   * Prvním krokem v TDD je vytvoření seznamu očekávaných variant nového chování. To zahrnuje základní případ a další scénáře, například situaci, kdy dojde k timeoutu služby nebo když klíč ještě není v databázi. Testy by měli být malé a zaměřovat se pouze na jednu část. Specifikace lze zjistit prostřednictvím uživatelských příběhů a uživatelských případů.
   * Příkladem v maturitní práci Unity 2D Action-Adventure hra s prvky stealth, strategie, puzzle je návrh nepřátel. Před jejich implementací byly definovány stavy ve kterých se může nepřítel nacházet a chování v nich. Například při stavu Idle by se měl nepřítel pohybovast v předem známém okruhu uvnitř mapy, neměl by procházet zdmi.
2. **Přidání jednoho testu ze seznamu:**
   * Napsání automatizovaného testu, který projde, pokud nové chování splňuje očekávání.
   * Příkladem je spuštění hry s mnoha nepřáteli, což odhalí možnost provést nežádoucí chování procházení zdmi.
3. **Spuštění všech testů:**
   * Nový test by měl selhat z očekávaných důvodů, což ukazuje, že je třeba nový kód k implementaci požadované funkce. To ověřuje, že testovací sada funguje správně a vylučuje možnost, že nový test je chybný a vždy projde.
   * Příkladem je znovu spuštění hry, kde nepřátelé občasně prochází zdmi díky nedostatečně robustní implementaci NavMeshAgent ukazatele.
4. **Napsání nejjednoduššího kódu, který projde nový test:**
   * I neelegantní nebo obtížný kód je akceptovatelný, pokud projde testem. Kód bude později vylepšen v kroku 6. Nepřidávejte žádný kód mimo testovanou funkcionalitu.
   * Příkladem je přidání kolize pro nepřátele, takže již fyzicky nemohou projít zdí a zastaví se před ní.
5. **Všechny testy by nyní měly projít:**
   * Pokud některý selže, nový kód musí být upraven, dokud všechny testy neprojdou. To zajišťuje, že nový kód splňuje požadavky testů a neporušuje existující funkce.
   * Příkladem je běh hry po přidání kolize. Je nutné kolizní oblasti přidat označení, které umožní ji ignorovat v jiných částech herního kódu, např. echolokaci.
6. **Refaktorování podle potřeby, s použitím testů po každém refaktoru k zajištění zachování funkcionality:**
   * Kód je upravován pro čitelnost a udržitelnost. Výslovně se snažte odstranit tvrdá kódovaná testovací data. Refaktorování je nutnou součástí TDD. Spuštění testovací sady po každém refaktoru pomáhá zajistit, že žádná existující funkcionalita není narušena.
   * Příkladem je přepsání kódu, aby se nepřátelé nesnažili jít dovnitř zdí. Tím pádem lze odstranit redundantní kolizní kód
7. **Přidání dalšího testu ze seznamu:**
   * Opakování procesu s každým testem na seznamu, dokud nejsou implementovány a všechny testy projdou.
8. **Opakování:**
   * Cyklus výše se opakuje pro každý nový kousek funkcionality. Testy by měly být malé a postupné, a commitovány by měly být často. To umožní, že pokud nový kód selže v některých testech, programátor může jednoduše vrátit zpět nebo revertovat, místo aby se musel nadměrně zabývat laděním. Při používání externích knihoven je důležité nepisovat testy, které jsou tak malé, že efektivně testují pouze samotnou knihovnu, pokud není nějaký důvod věřit, že knihovna je chybná nebo nedostačující pro potřeby vyvíjeného softwaru.

# Benefity TDD

1. **Lepší kvalita kódu:**
   * TDD podporuje vývojáře v psaní čistšího a modulárnějšího kódu, protože musí předem napsat testy pro svůj kód.
2. **Regresní testování:**
   * Testy slouží jako záchranná síť a umožňují vývojářům provádět změny v kódové základně s jistotou, že stávající funkce nebudou neúmyslně rozbity.
3. **Zlepšení návrhu:**
   * TDD často vede k lepšímu designu softwaru, protože podněcuje vývojáře, aby přemýšleli o návrhu svého kódu ještě před jeho implementací.
4. **Rychlejší ladění:**
   * Protože jsou chyby zachyceny v rané fázi vývoje, je obvykle snazší a rychlejší je opravit.