Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Bakalářská práce

Testování SW komponent SmartCGMS

Pilsen, 2021 David Markov

ZÁPADOČESKÁ UNIVERZITA V PLZNI

Fakulta aplikovaných věd

Akademický rok: 2020/2021

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**David MARKOV**
Osobní číslo:	**A18B0262P**
Studijní program:	**B3902 Inženýrská informatika**
Studijní obor:	**Informatika**
Téma práce:	**Testování SW komponent SmartCGMS**
Zadávající katedra:	**Katedra informatiky a výpočetní techniky**

Zásady pro vypracování

1. Seznamte se s programátorským rozhraním systému SmartCGMS (diabetes.zcu.cz/smartcgms) a nástroji pro tzv. continuous deployment.
2. Vytvořte jednotkové testy na úrovni API SmartCGMS, které ověří funkčnost jednotlivých komponent.
3. Vytvořte 3 regresní testy celého systému, které budou použité při rozšiřování API.
4. Celé řešení integrujte do vybraného continuous deployment nástroje.
5. Zdokumentujte vytvořené testy a zhodnoťte dosažené výsledky.

Rozsah bakalářské práce: **doporuč. 30 s. původního textu**
Rozsah grafických prací: **dle potřeby**
Forma zpracování bakalářské práce: **tištěná**

Seznam doporučené literatury:

Dodá vedoucí bakalářské práce.

Vedoucí bakalářské práce: **Ing. Martin Úbl**
Katedra informatiky a výpočetní techniky

Datum zadání bakalářské práce: **5. října 2020**
Termín odevzdání bakalářské práce: **6. května 2021**

L.S.

Doc. Dr. Ing. Vlasta Radová
děkanka

Doc. Ing. Přemysl Brada, MSc., Ph.D.
vedoucí katedry

V Plzni dne 26. října 2020

Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Pilsen dne 4. května 2021

David Markov

Abstract

The goal of this thesis is to create a framework for SmartCGMS software components testing. Framework is implemented in C++ language in C++17 standard. In theoretical part, the problematic of tested system and used technologies is described. Practical part consists of framework design and implementation and it's integration into the process of continuous integration and deployment. The conclusion contains an evaluation of the results.

Abstrakt

Cílem této práce je vytvoření nástroje pro testování softwarových komponent SmartCGMS. Pro implementaci tohoto nástroje je využit jazyk C++ a jeho standard C++17. V teoretické části je popsána problematika testovaného systému a použité technologie. Praktická část sestává z návrhu a implementace testovacího nástroje a jeho začlenění do cyklu kontinuální integrace a kontinuálního nasazení. Závěr práce se věnuje zhodnocení dosažených výsledků.

Poděkování

Rád bych poděkoval panu Ing. Martinu Úblovi za vedení bakalářské práce a za jeho cenné rady. Dále bych mu rád poděkoval za jeho trpělivost.

Obsah

1 Úvod

Žijeme v době technologického rozvoje. Vzniká nespočet nových systémů a nástrojů pro usnadnění našeho každodenního života. S každým vytvářeným softwarem je ale třeba brát zřetel na jeho důkladné otestování. Každý kus softwaru, který je nasazen do produkčního prostředí, musí být řádně otestován tak, aby při jeho běhu v ostrém provozu nedocházelo k žádným, nebo alespoň k minimálním a neškodným chybám. Toto platí dvojnásobně u softwarových produktů, které jsou určeny k používání v medicínském prostředí.

Cílem této práce je vytvořit jednoduchý a spolehlivý testovací nástroj pro testování softwarových komponent *SmartCGMS*. Tento nástroj musí být přenositelný a musí být napojen na aplikační programovací rozhraní *SmartCGMS*. Dále bude třeba ho začlenit do vybraného nástroje pro kontinuální integraci a kontinuální nasazování, aby bylo možné funkcionalitu testovaného systému ověřovat pravidelně. Práce vychází ze základu implementovaného v rámci předmětu KIV/ZSWI. Nová podoba tohoto nástroje je značně přepracovaná a obohacena o implementaci nových testovacích scénářů.

Nejprve dojde k seznámení se s testovaným systémem a problémovou doménou. Pak bude prostudována problematika softwarového testování a kontinuální integrace a bude navržen nástroj pro automatické testování. Ten bude implementován a integrován do vybraného systému pro kontrolu kontinuální integrace. V diskusi dojde ke zhodnocení dosažených výsledků, ze kterých bude vyvozen závěr.

2 Diabetes mellitus

Diabetes mellitus (DM), česky úplavice cukrová, krátce „cukrovka“, je chronické metabolické onemocnění, způsobené absolutním nebo relativním nedostatkem inzulinu. Absolutní nedostatek inzulinu je zapříčiněn tím, že ho tělo nevyrábí tolik, kolik by mělo. Naproti tomu při relativním nedostatku je produkce inzulinu v normě, ale buňky ho potřebují mnohem větší množství. Vyznačuje se především chronickou hyperglykémií a výskytem glukózy v moči. Rozsah změn při DM je ale mnohem širší. DM představuje poruchu látkové přeměny, která postihuje metabolismus nejen cukrů, tuků a bílkovin, ale i metabolismus vody a elektrolytů. Většina poruch je způsobena nedostatečným množstvím inzulinu v krvi, nebo jeho nedostatečným působením na cílovou buňku.

2.1 Hlavní příznaky

Základním příznakem DM je hyperglykémie. Hranice glykémie, kdy se již jedná o hyperglykémii, je asi 7 $mmol/l$ na lačno a 10 $mmol/l$ po jídle a během dne [8]. Ta je způsobena především zvýšenou produkcí glukózy játry a ledvinami a zvýšeným rozpadem glykogenu. Pokud koncentrace glukózy v krvi překročí cca 10 $mmol/l$, dochází u paceinta ke glykosurii - vylučování glukózy močí, a ke zvýšenému vylučování tekutin - polydipsii. Tento stav nazýváme překročením tzv. *renálního prahu*[22].

Diabetes patří k celoživotním onemocněním, které mají tendenci postupného rozvoje komplikací. K dlouhodobým komplikacím při diabetu patří například *retinopatie* - onemocnění sítnice, při kterém dochází k poškození vyživujících cév, vedoucí v krajních případech až ke slepotě; *nefropatie*, vedoucí k selhání ledvin; periferální neuropatie, přinášející riziko amputací (diabetická noha), a další [2].

2.2 Typy diabetu

DM 1. typu je autoimunitní onemocnění, při kterém dochází k absolutnímu nedostatku inzulinu. K tomuto nedostatku dochází destrukcí β-buněk Langerhansových ostrůvků, která je způsobena právě autoimunitní reakcí. U pacientů bývá diagnostikován v mladistvém věku a pacienti jsou doživotně odkázáni na umělý přísun inzulinu. Rozvoj tohoto typu je dán genetickou

predispozicí [22].

DM 2. typu je nejrozšířenějším typem diabetu. Kolem 90 % pacientů trpí právě tímto typem cukrovky. Zde se již nejedná o absolutní, nýbrž o relativní nedostatek inzulinu. Produkce inzulinu u diabetika typu 2 je většinou v normě, ale jeho periferní buňky jsou na inzulin necitlivé a vyžadují jeho nadměrné množství. Tento typ je často spojen s obezitou. Až 80 % pacientů, diagnostikovaných s DM 2. typu, je obézních [22]. Obezita sama o sobě způsobuje určitou úroveň necitlivosti periferálních buněk k inzulinu.

Tento typ diabetu se vyvíjí pomaleji a v raných fázích tohoto onemocnění nemusí být hyperglykémie tak výrazná. Pacient tedy nemusí zaregistrovat žádné ze známých příznaků, nebo jim nevěnuje takovou pozornost, jelikož jsou minimální. Léčba tohoto typu často není spojena s umělým přísunem inzulinu, nýbrž s dietou a výživovými opatřeními.

2.3 Gestační diabetes

Jako Gestační diabetes (GDM) je označován Diabetes mellitus, který je diagnostikován u těhotných žen. V období těhotenství totiž přirozeně stoupá necitlivost periferních buněk vůči inzulinu. Tímto typem DM je diagnostikováno kolem 10 % těhotných žen [22]. Po porodu GDM zpravidla odezní, ale zůstává zde zvýšené riziko DM 2. typu v budoucnu. Obezita a další faktory, které zvyšují necitlivost vůči inzulinu, toto riziko výrazně zvyšují. U potomků žen, které během těhotenství prodělaly GDM, pozorujeme zvýšené riziko výskytu obezity a diabetu během dospívání [1].

2.4 Měření glykémie

Glykémie u pacienta je kontinuálně měřena pomocí *continuous glucose monitoring* (CGM) systémů. Tyto systémy neměří hladinu glukózy v krvi, ale v podkožní tkáni. Výskyt glukózy v podkožní tkáni, společně s enzymem, který je aplikován na jehlu senzoru CGM systému, vyvolá chemickou reakci. Tato reakce způsobí vytvoření elektrického proudu, který je následně CGM systémy měřen. Tento proud je dále konvertován na intersticiální hladinu glukózy. Tato konverze je kalibrována pacientem samotným tím, že si měří hladinu cukru v krvi. Pacient si manuálně měří hladinu glukózy v krvi, ať už právě kvůli kalibraci senzoru CGM systému, nebo aby si byl jist správností měření tohoto systému. Navíc jde o dvě rozdílné hodnoty, jelikož krev na výkyvy reaguje rychleji. Toto měření je prováděno cca dvakrát nebo třikrát

denně [20] pomocí vytlačení kapky krve z prstu horní končetiny pacienta. CGM systém měří hladinu glukózy v podkožní tkáni pravidelně v krátkých intervalech, zatímco zvýšená hladina cukru v krvi je pacientem měřena pouze sporadicky. Díky naměřeným záznamům je software inzulinové pumpy schopen vypočítat vhodné množství inzulinu, které je třeba pacientovi do krve doplnit a následně ho vpraví pacientovi do podkožní tkáně [17].

Stejně jako každý systém, i tato zařízení mohou vykazovat různé chyby. Měřící systémy se mohou z různých důvodů odpojovat, signál může být zašuměný a podobně. Software, který je výstupem této práce, může tyto chyby emulovat a následně sledovat a testovat chování jednotlivých komponent a jejich reakce na emulovanou chybu.

3 SmartCGMS

SmartCGMS (SCGMS) je systém, vyvíjený na Katedře informatiky a výpočetní techniky Západočeské univerzity v Plzni, který slouží ke zpracování fyziologických signálů a vyvození patřičných závěrů (predikce hladiny glukózy v krvi, výpočet dávek inzulinu a podobně). *SmartCGMS* staví na principech *High-level architecture* (HLA). HLA je standard pro distribuované simulace, pomocí kterého lze provádět ko-simulace. Ko-simulace sestává z dílčích reálných a simulovaných zařízení a může obsahovat i prototypovaná zařízení. Tato zařízení, ať už reálná nebo simulovaná, mohou pracovat zároveň v jedné simulaci. Lze tedy například simulovat inzulinovou pumpu s reálným senzorem, nebo naopak, simulovat senzor s reálnou inzulinovou pumpou. Tento princip je velmi důležitý, jelikož dovoluje nahradit jednotlivé komponenty systému jinou virtuální komponentou, která se bude chovat jako kontext systému. Tento standard byl vyvinut v devadesátých letech v USA pod záštitou ministerstva obrany [4] a později se z něj stal mezinárodní IEEE standard.

3.1 Architektura

Architektura *SmartCGMS* sestává z takzvaných *filtrů*. Každý filtr si lze představit jako samostatný nezávislý modul. Tyto filtry jsou lineárně propojeny a každý má svou jedinečnou funkci, například vstupní filtr načítá hladinu glukózy, kterou naměřil CGM systém v podkožní tkáni pacienta pomocí senzoru. Hladina může být načtena také pomocí ovladače z připojené databáze. Funkcionalita filtrů je různá. Některé modifikují předávané údaje o naměřených hodnotách, některé tyto údaje zobrazují uživateli do přehledných grafů, jiné zas ukládají data do databáze, souboru na pevném disku a tak dále [23]. Propojení dvou po sobě jdoucích filtrů si lze představit jako rouru, kterou prochází jednotlivé události, které jsou uvnitř filtrů zpracovávány. Roura pracuje na principu předávání zpráv. Filtr na jedné straně událost přijme, zpracuje ji a provede nutné akce a událost pošle dál na výstup, kde ji přijme filtr následující. Při zpracování události může filtr vyprodukovat další nové události a ty také odeslat na výstup.

Komunikace všech filtrů probíhá synchronně. Pokud je to třeba, filtr si může interně vytvořit vlastní vlákno v rámci konfigurace. Předávání událostí je realizováno také synchronně, dojde k němu tedy okamžitě. Zpracování události si lze představit jako rekurzivní proces, jelikož filtr po jejím zpracování nepřímo zavolá zpracování události dalším filtrem. To je vhodné

pro zaručení stability v rámci systémů reálného času, kde jsou často požadavky na fixní velikost předem alokované paměti.

Jedním z možných scénářů může být to, že na začátku řetězu filtrů je vstupní filtr, který načte data o kontinuálně naměřené hladině intersticiální glukózy z CGM senzoru nebo z databáze. Následuje filtr výpočetní, který z příchozích dat vypočte hladinu glukózy podle jednoho z modelů pro rekonstrukci hladiny glukózy v krvi. Tato data pak dále do mapovacích filtrů, které tyto vypočtené hodnoty přemapují na virtuální signál. Poté predikční filtr vypočte předpokládanou budoucí hladinu glukózy a data odešle do vizualizačních filtrů, které data zobrazí do uživatelského rozhraní ve formě přehledného grafu. Na konci tohoto řetězu může být připojen filtr, který slouží jako ovladač inzulinové pumpy. Z vypočtených dat je třeba určit vhodné množství inzulinu k aplikaci. Buď jsou data odeslána do inzulinové pumpy, která si to vypočte sama, nebo je množství vypočteno specializovaným filtrem v rámci *SmartCGMS*.

3.2 Ostatní entity

Za nejobecnější entity lze považovat *moduly*. Moduly jsou distribuovány ve formě sdílených knihoven. Díky tomu je možné je zavádět do systému až za běhu, což zaručuje jejich vzájemnou nezávislost. V modulech jsou obsaženy tzv. *deskriptory* entit, které tento modul distribuuje. Ty obsahují informace o tom, jaký `GUID` [19] reprezentuje danou entitu, její popis, jaké vstupní parametry vyžaduje a podobně. Zároveň poskytuje funkce k vytvoření entit, jejichž deskriptory obsahuje.

Moduly mohou kromě filtrů obsahovat i jiné entity, jako například *model, metrika, aproximátor, solver* nebo *signál*. *Solvery* a *signály*, nebudou v této práci testovány.

Entita `Metric` poskytuje rozhraní, přes které jí lze předávat množinu referenčních a spočtených hodnot. Na základně těchto vstupních dat dokáže vnitřně akumulovat a vypočítat metriku (vzdálenost), tedy zobrazení, mezi předanými množinami hodnot.

Entity typu `Signal` jsou reprezentací libovolného signálu, jako je hladina glukózy v krvi, množství intersticiální hladiny glukózy a dalších. Každý signál nabývá diskrétních hodnot v určitých časech. Není ale možné mít uloženy hodnoty signálu v každém časovém okamžiku, protože bychom museli ukládat nekonečné množství dat. Proto jsou implementovány *aproximátory*, které jsou schopné na základě několika předaných diskrétních hodnot signálu v daných časech aproximovat hodnotu signálu v libovolném čase mezi dvěma

hodnotami.

Velmi specifickou entitou ve *SmartCGMS* je `Model`. Díky nim lze simulovat například určitý stav pacienta a na základě toho buď simulovat funkcionalitu v daných podmínkách, nebo jen systém ladit ve stabilním prostředí. Modely se dělí na dva typy:

1. **Diskrétní modely**, jejichž stavové proměnné se v čase mění nespojitě v určitých okamžicích (skokově)

2. **Spojité modely**, jejichž stavové proměnné mění svoje hodnoty spojitě s časem

3.3 Druhy událostí

Událost, která je mezi filtry předávána, je reprezentována generickou datovou strukturou. Každá událost obsahuje logické hodiny, jedinečné pro danou událost. Dále obsahuje `GUID` [19] zařízení, které tuto událost vyprodukovalo a `GUID` signálu, kterým se tato událost zaobírá (zvýšená hladina glukózy v krvi, vypočtená hladina glukózy, inzulin, předpokládaná budoucí hladina glukózy v krvi a další) [17]. K dalším informacím, které událost přenáší, patří fyzické hodiny, podle kterých lze zjistit časovou posloupnost jednotlivých událostí.

Každá událost má přiřazen i svůj kód, který značí k čemu se událost využívá. Základními typy událostí jsou informativní události `Information`, `Warning` a `Error`. Tyto události jsou pouze informativní a slouží pro analyzátory logů. `Warm_Reset` událost uvede filtr do původního stavu, jako kdyby byl právě vytvořen. Speciálním typem informativní události je `Shut_Down` událost. Ta je využívána pro ukončení činnosti filtru. Pokud filtr spouštěl dodatečné vlákno, je toto vlákno ukončeno. Celý filtr je pak uveden do stavu „zombie“. Po přijetí této události již filtr neprovádí žádné výpočetní operace, ani nepřijímá další události.

Dalším důležitým typem je `Level` událost, která udává hodnotu naměřeného nebo vypočítaného signálu.

3.4 Přenositelnost

SmartCGMS je implementován jako přenositelný systém. Je tedy možné beze změny zdrojového kódu tento systém spustit na široké škále zařízení a operačních systémů. Většina filtrů navíc pracuje synchronně, díky čemuž

systém přichází o značnou část výpočetní složitosti, se kterou by se musel potýkat, pokud by všechny filtry pracovaly asynchronně [23]. Systém je implementován tak, aby ho bylo možno využívat i na zařízeních s omezeným množstvím elektrické energie. Mezi ně patří například mobilní telefon a chytré hodinky.

Mobilní telefon je dnes v podstatě nejpoužívanějším mobilním zařízením. Pomocí technologie Bluetooth, konkrétně Bluetooth Low Energy (BLE) se lze mobilním telefonem připojit k CGM systému.

Naopak k mobilnímu telefonu se lze připojit pomocí fenoménu posledních let, chytrých hodinek. Proto se implementace *SmartCGMS* zaměřila i na tato zařízení.

Implementace na všech zařízeních sdílí společné zdrojové kódy,
které pro běh systému na různých zařízeních není třeba upravovat. Systém je implementován v jazyce `C++` ve standardu `C++17`. Tento jazyk byl vybrán z důvodu jeho efektivity, přenositelnosti a vysokému optimalizačnímu potenciálu [18].

4 Softwarové testování

Softwarové testování je proces průzkumu softwaru, který vývojářům poskytuje informaci o kvalitě vyvíjeného produktu. Součástí testování je spouštění tohoto softwaru nebo jeho částí v různých podmínkách tak, aby byly otestovány jednotlivé vlastnosti produktu. Otestováním těchto vlastností můžeme obecně zjistit, že software:

- odpovídá specifikaci a předchozímu návrhu
- správně reaguje na širokou škálu vstupů
- vykonává svoje funkce v rozumném čase
- dá se uživatelem pohodlně používat
- je ho možné spustit v prostředí, do kterého je určen
- opravdu dělá to, co si cílový zákazník přeje, aby dělal

K přístupům testování patří například spouštění programu s cílem nalezení chyb a defektů a ověření, zda je software vhodný k používání. Testováním nelze potvrdit, že produkt pracuje správně za všech podmínek, ale lze jím potvrdit fakt, že produkt za některých podmínek správně nepracuje [15].

V reálných podmínkách ovšem už i s malým a poměrně jednoduchým systémem rapidně roste počet testů, které je třeba vykonat, aby byly tyto vlastnosti ověřeny. V praxi se software nikdy nedá otestovat úplně. Doména všech možných vstupů je na to příliš široká. [15]. Proto existuje široká škála přístupů k testování, kdy každý přístup nabízí vhodnou strategii k testování produktu, která je proveditelná s aktuálně dostupným množstvím času a prostředků. Otestování tohoto produktu dokáže poskytnout objektivní informaci o jeho kvalitě a riziku chyby jeho uživatelům [14]. Obecně ale mají všechny společné to, že není třeba vykonat vyčerpávající množství testů, ale pouze tolik testů, kdy už se nevyplatí provádět další za účelem nalezení další chyby.

Testování se provádí v různých fázích vývoje podle použitého vývojového cyklu. Například v sekvenčním procesu, jako je vodopádový model, se testování provádí v příslušné fázi, kdy jsou požadavky na systém již sepsány a implementovány v testovatelném programu. Naproti tomu v agilních metodikách jsou požadavky, implementace a testování prováděny paralelně.

V dnešních softwarových společnostech jsou často vyčleněna samostatná oddělení, která mají testování softwaru jako primární úkol. Zde jsou prováděny testy větších rozměrů, které samotný vývojář nemůže v průběhu implementace provést. Vývojář by naopak měl v průběhu implementace provádět jednoduché dílčí testy nad jednotlivými částmi zdrojového kódu, aby ověřil, že jednotlivé procedury nebo funkce fungují podle očekávání.

4.1 Validace a verifikace

Mezi základní pojmy v softwarovém testování patří *validace* a *verifikace* [21]. Tyto dva pojmy jsou rozdílné, ale jsou často chybně interpretované a zaměňované.

V rámci validace ověřujeme, jestli produkt, který vyvíjíme je opravdu takový produkt, který bychom vyvíjet měli. To znamená ověření toho, jestli software umí to, co zákazník a uživatel chce. Pokud software nesplňuje požadavky ve specifikaci, nemusí být koncovým zákazníkem akceptován a zaplacen.

Verifikace naproti tomu zjišťuje, jestli je vyvíjený produkt vyvíjen správně, tedy jestli jsou specifikované požadavky implementovány správně. Oba tyto pojmy vedou k tomu, že před samotným zahájením vývoje softwarového produktu bývá sepisována *specifikace požadavků*. V ní se zákazník a dodavatel softwarového produktu dohodnou na jednotlivých funkčních a mimofunkčních požadavcích, které zákazník požaduje a které budou dodavatelem poskytnuty. Specifikace mimofunkčních požadavků bývá často složitá, jelikož zahrnuje pojmy jako testovatelnost, rozšiřitelnost, výkon a bezpečnost. Tyto pojmy mohou být vnímány subjektivně a může docházet ke konfliktům. Bezpečnost produktu může jednomu zákazníkovi vyhovovat, druhému už nemusí.

4.2 Defekt a selhání

Často používané pojmy v softwarovém testování jsou *defekt* a *selhání*. K chybám softwaru dojde následujícím procesem: Vývojář udělá chybu, která vyústí v *defekt*, neboli tzv. *bug* ve zdrojovém kódu. Pokud je pak kód, ve kterém se tento defekt nachází, spuštěn, dojde k *selhání* programu. Ne každý defekt musí nutně vyústit v selhání programu, například defekt v *mrtvém kódu*[1] nikdy v selhání nevyústí. Defekt se také může projevit například až při změně prostředí, ve kterém je program spouštěn, nebo při změně vstupních dat.

[1] kód, který se nikdy nevykoná, nebo jehož výsledek se nikdy nepoužije

Jeden defekt může mít širokou škálu následků při vykonání kódu, který ho obsahuje.

4.3 Přístupy k testování

Existuje mnoho přístupů k testování softwaru. Základní rozdělení tvoří *statické* a *dynamické* testy [9]. Ke statickým testům často řadíme procházení kódu, kontroly struktury kódu a podobně. Toto testování za nás často provádí již překladač nebo integrované vývojové prostředí (IDE). Také sem řadíme takzvaná *code-review*, kdy často zkušenější vývojář prochází kód napsaný méně zkušeným vývojářem a poskytuje mu zpětnou vazbu.

Statické testy jsou tedy takové testy, které nevyžadují spuštění zdrojového kódu. Naopak dynamické testy probíhají, když je zdrojový kód spuštěný. K těmto testům je často vytvořena sada testovacích scénářů nebo případů (*test-case*), které jsou za běhu programu otestovány.

Jako další typy testovacích přístupů lze zmínit například testování kódu jako černé (*black-box testing*) nebo bílé (*white-box testing*) skříňky. Při black-box testování nemá tester žádné informace o vnitřní implementaci testovaného programu. Má k dispozici pouze sadu vstupních dat a očekávaná výstupní data, která porovnává s reálnými výstupními daty. U white-box testování se již navíc provádí další testování na různých úrovních zdrojového kódu, jako jsou *jednotkové* testy [3].

4.4 Jednotkové testování

Jednotkové testy jsou metodou testování softwaru, která se zaměřuje na jednotlivé části zdrojového kódu - jednotky, a ověřuje, jestli pracují podle specifikace [11]. Tyto testy jsou automatizované a jsou prováděny vývojáři, kteří jednotkové testy píší pro každou jednotlivou část zdrojového kódu. Tyto části mohou být v procedurálním programování celé moduly nebo jednotlivé procedury a funkce. V objektově orientovaném programování se může jednat o celá rozhraní, nebo jen jednotlivé metody třídy. Jednotkově lze například testovat černou skříňku nějaké entity, jejíž rozhraní je jediné, co známe.

Psaní jednotkových testů často úzce souvisí s parametrizovanými testy. Tyto testy jsou zobecněné jednotkové testy, které samy očekávají vstupní parametry a tím umožňují vícenásobné použití jednoho jednotkového testu pro více různých případů v závislosti na vstupních parametrech.

4.5 Regresní testování

V softwarovém inženýrství se regresí rozumí chyba softwaru, která nastala po nějaké události ve vývojovém procesu, jako je oprava jiného zavlečeného defektu, změna prostředí, ve kterém testovaný produkt běží nebo i zapracování nové funkce. Regresní testování je sestava takových testů, které ověřují, že se software i po takovýchto změnách chová stále stejně a žádná z jeho dříve fungujících částí v aktuální podobě neselhává. S každým nalezeným defektem se sada regresních testů rozšiřuje. Často je tedy s regresními testy spojena i jejich automatizace, aby se tyto testy daly pohodlně spouštět.

4.6 Přístup k testování SmartCGMS

Testování systému *SmartCGMS* bude prováděno formou černé skříňky. Známo bude pouze rozhraní klíčových modulů a jednotek, ale vnitřní implementace zůstane skryta. Bude implementována sada jednotkových testů, které *verifikují* funkcionalitu těchto jednotek vůči dodané specifikaci. Ačkoli může systém projít všemi jednotkovými testy, stále je zde možnost zavlečení *regrese*. Proto bude zapracován i mechanismus *regresních testů*, ověřující celkovou funkčnost systému na dodaných scénářích. Ke každému scénáři bude znám očekávaný výsledek, se kterým bude výstup systému porovnáván.

Jelikož bude testování prováděno formou černé skříňky, nebude známa detailní implementace. Proto nebude možné otestovat přímo konkrétní dílčí metody a funkce jednotlivých filtrů. V tomto případě bude muset být testování prováděno pomocí zasílání zpráv, což je základní princip fungování architektury *SmartCGMS*.

5 Kontinuální integrace a nasazení

Kontinuální integrace (*angl. continuous integration*) je proces průběžného začleňování změn zdrojového kódu od více vývojářů v jednom projektu. Tento proces, jehož základem je verzovací systém (například *Git*), sestává z automatických nástrojů pro kontrolu správnosti nového kódu před začleněním do starší, funkční verze. Tyto nástroje mohou kontrolovat sestavitelnost kódu nebo například správnost syntaxe.

Síla postupné integrace spočívá v tom, že je prováděna velmi často. Všichni vývojáři, kteří na projektu pracují, integrují své změny zdrojového kódu do centrálního repozitáře prakticky denně a jelikož tyto změny jsou ze zásady co nejmenší, dojde díky tomu k odhalení chyb velmi rychle.

Systém průběžné integrace je vhodné místo, kam začlenit systém automatického přeložení zdrojových kódů a automatické testy, jelikož dokáže spustit všechny testy a ověřit všechny scénáře asynchronně bez jakékoli závislosti na vývojáři. Repozitář zdrojového kódu by měl být napojen na systém automatického přeložení, který běží na dostupném serveru a je schopen překládat zdrojový kód ve svém integrovaném prostředí. Toto prostředí je nastaveno vývojáři tak, aby simulovalo takové prostředí, ve kterém bude aplikace následně fungovat. Pokud sestavovací systém úspěšně přeloží zdrojový kód v tomto prostředí, s velkou pravděpodobností naše aplikace poběží po nasazení tak jak má. Součástí procesu sestavení jsou i automatizované testy, které jsou po úspěšném přeložení spuštěny nad funkční aplikací. K tomuto procesu dochází zpravidla poté, co vývojář nahraje změny ve zdrojovém kódu do centrálního repozitáře. Některé systémy umožňují tento proces spouštět automaticky například každou noc. Pokud kód projde tímto procesem bez chyb, je označen za validní. Pokud ne, je označen jako nevalidní a často je odeslána notifikace správci repozitáře, nebo vývojáři, jehož změny způsobily chybu sestavení. Notifikace obsahuje informace o tom, že se kód nepodařilo sestavit a z jakého důvodu. Vývojář má poté povinnost tuto chybu co nejdříve opravit a zajistit, aby byl kód opět přeložitelný a prošel automatizovanými testy [7].

Tyto systémy jsou poskytovány buď v cloudové verzi nebo je lze spouštět na vlastním serveru. Cloudové řešení odpovídá principu *software jako služba*, kdy je software nasazen přímo vývojářem aplikace a zákazníkům je poskytován přes webové stránky prostřednictvím sítě Internet.

5.1 Systémy pro správu repozitářů

S kontinuální integrací úzce souvisí systémy pro správu repozitářů zdrojových kódů. Jsou to webové aplikace, které umožňují prohlížení repozitářů pomocí uživatelského rozhraní. Pro správu využívají verzovací systémy, jako jsou *Git* nebo *Mercurial*. Častou vlastností těchto nástrojů je systém sledování chyb. Do něj lze přidávat záznamy o chybách a sledovat průběh jejich řešení.

K nejznámějším systémům pro správu repozitářů patří *GitHub* a *GitLab*. GitHub je vlastněn společností *Microsoft* a využívá verzovací systém Git. Je převážně využíván individuálními vývojáři pro vývoj open-source projektů a lze v něm procházet veřejné repozitáře všech registrovaných uživatelů. GitLab je open-source systém, využívající verzovací systém Git. Umožňuje integrování v softwaru třetích stran a je často využíván vývojovými týmy a společnostmi v rámci interního vývoje produktů.

Dalšími systémy jsou například *Bitbucket*, který cílí na vývojové týmy, a *Assembla*, který je určen pro využití v rámci podniků.

5.2 Jenkins

Jenkins, původně známý jako Hudson, je open-source nástroj pro kontinuální integraci napsaný v jazyce Java. Je možné ho nainstalovat na vzdálený server pomocí technologií jako jsou *Docker* nebo *Kubernetes* nebo je možné ho používat samostatně jako aplikaci na prostředí, které využívá *Java Runtime Environment (JRE)* [13]. Jenkins je vysoce modulární a podporuje rozšiřování svých instancí uživatelem pomocí zásuvných modulů.

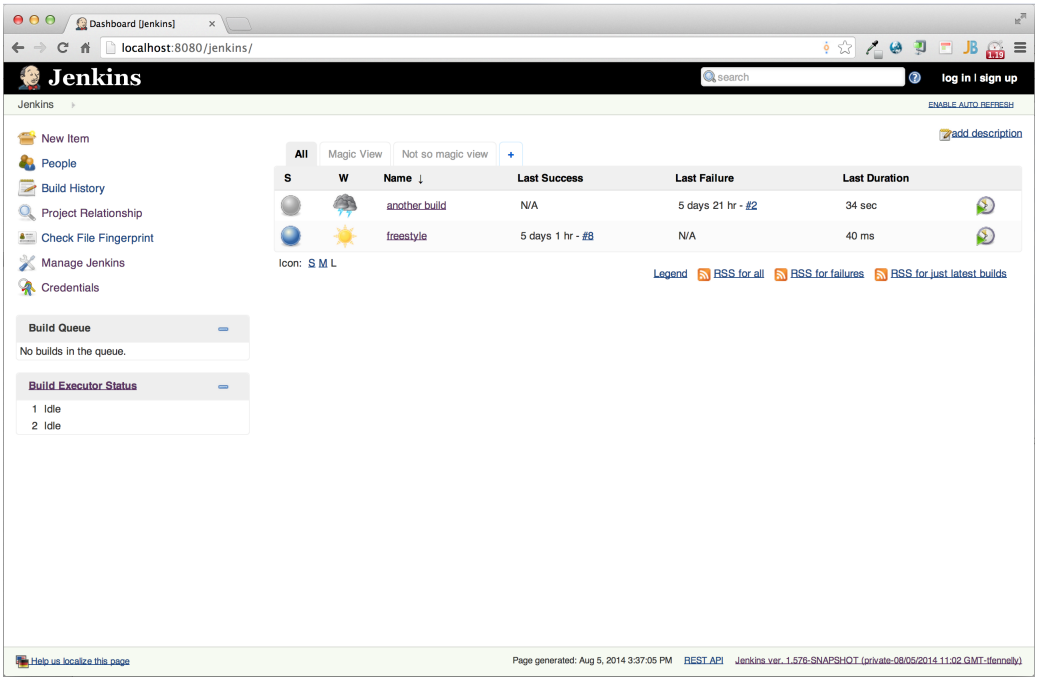
Základem nástroje Jenkins je tzv. *Jenkins Pipeline*, nebo krátce pouze *Pipeline*. Pipeline je soubor zásuvných modulů, který umožňuje integraci a implementaci kontinuálního nasazování do Jenkinsu. Podoba této implementace je definována ve skriptu v souboru zvaném `Jenkinsfile` [12]. Tento soubor je vložen do repozitáře zdrojového kódu, který je napojen na běžící instanci systému Jenkins. Po aktualizování provedených změn v centrálním repozitáři je Jenkins notifikován systémem pro správu zdrojových kódů nebo zpětným webovým voláním o změně. Podle přiložené definice pipeline v `Jenkinsfile` pak provede nadefinované operace. Tyto operace mohou zahrnovat sestavení, otestování nebo nasazení aplikace k zákazníkovi.

Projekt se může nacházet v několika stavech:

- **rozbitý** - projekt je rozbitý, pokud jeho poslední sestavení selhalo

- **nestabilní** - projekt je nestabilní, pokud byl sestaven úspěšně, ale některá další část procesu selhala - například neprošel jeden nebo více testů
- **stabilní** - projekt je stabilní, pokud sestavení i všechny ostatní části procesu proběhly úspěšně

Jenkins poskytuje jednoduché a intuitivní uživatelské rozhraní, viz obrázek 5.1, ve kterém lze snadno spravovat všechny pipeline, procházet záznamy jednotlivých kroků i komunikovat s celým týmem.

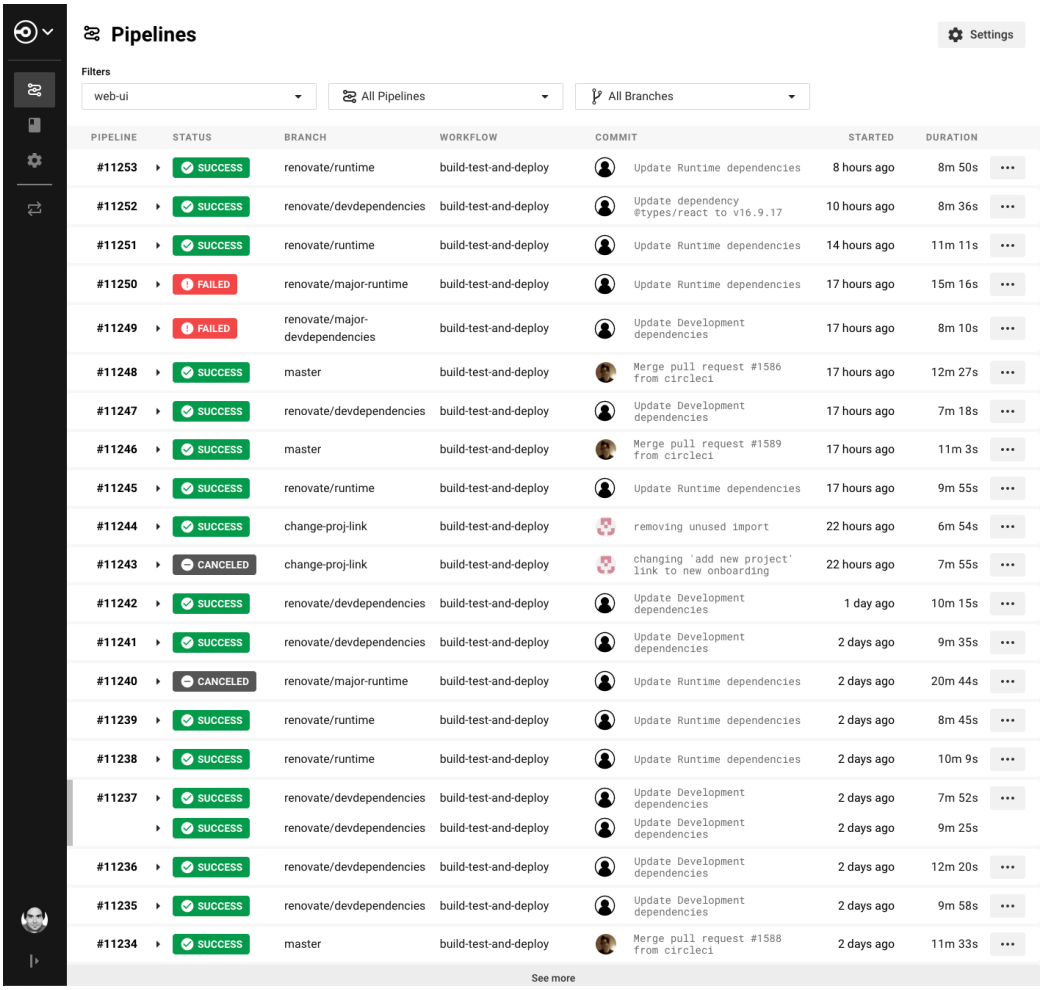


Obrázek 5.1: Uživatelské rozhraní nástroje Jenkins. zdroj: https://www.jenkins.io/blog/2014/08/11/user-interface-refresh/

5.3 CircleCI

Dalším z mnoha nástrojů pro kontinuální integraci je *CirecleCI*. CircleCI je moderní nástroj pro správu automatizovaných testů a sestavení repozitáře zdrojových kódů. Nabízí velmi jednoduchou integraci se službami *GitHub* a *Bitbucket* a vykonávání takzvaných *workflows*. Tyto procesy jsou ve své podstatě obdobou pipeline v systému Jenkins. Nastavují se v adresáři `.circleci` v konfiguračním souboru `config.yaml` přidaným do repozitáře,

který je napojen na aplikační programovací rozhraní CircleCI a ten se o zbytek postará. CircleCI nabízí cloudové řešení, které je zdarma pro každého, ale s jistými omezeními. Například je možné sestavovat pouze na platformách Linux a Windows, nikoli na MacOS a počet paralelně běžících akcí je limitován na 1. Další rozšíření benefitů, jako soukromé projekty, nebo více souběžně vykonávaných akcí je zpoplatněno. Webové rozhraní je vidět na obrázku 5.2. Pro uživatele, kteří chtějí službu spravovat na svém vlastním serveru je možnost si stáhnout komerční verzi CircleCI, která je zdarma na zkušební dobu 14 dní. Poté služba přechází na model předplatného, obdobně jako její cloudová verze.



Obrázek 5.2: Pohled na seznam proběhlých pipelines v nástroji CircleCI zdroj: https://github.com/marketplace/circleci

5.4 TravisCI

TravisCI je další z kategorie komerčních nástrojů pro kontinuální integraci. Podobně jako CircleCI nabízí cloudové i komerční řešení. Cloudové řešení je zdarma pro kohokoli, ale uživatel je limitován jistými omezeními. Každý účet má ze začátku přiřazeno 10000 kreditů, které se uživateli ubírají s každou započatou minutou, kterou Travis stráví sestavováním nebo testováním projektů. Po vyčerpání je třeba kredity dokoupit nebo přejít na plnou verzi této služby, která je zpoplatněna. Stejně jako u dříve zmíněných systémů, jednotlivé pipelines jsou nastavitelné z konfiguračního souboru `.travis.yml`, který je vložen do připojeného repozitáře. Oproti CircleCI nabízí integraci nejen se službami *GitHub* a *Bitbucket*, ale i se službami *GitLab* a *Assembla*. Travis také poskytuje webové uživatelské rozhraní, které je viditelné na obrázku 5.3.

Obrázek 5.3: Uživatelské rozhraní nástroje TravisCI. zdroj: https://dzone.com/articles/how-to-manage-continuous-releases-with-travis-ci-a-1

6 Návrh řešení

V této kapitole bude navržen nástroj *SmartTester* tak, aby vyhovoval požadavku testování černé skřínky. Bude navržen proces regresních testů a vybrán vhodný systém pro kontrolu kontinuální integrace, do kterého bude navržený nástroj začleněn. Dále bude představeno, co je od jednotlivých entit v rámci jejich funkcionality očekáváno.

6.1 Volba technologií

Vzhledem k úzkému provázání vyvíjeného testovacího nástroje se systémem *SmartCGMS*, který je vyvíjen v jazyce `C++`, bude i pro tento nástroj zvolen jazyk `C++`, konkrétně jeho standard `C++17`. Pro správu automatického sestavení projektu a správu vnějších závislostí bude využit nástroj CMake [16]. Nástroj bude prakticky využíván v rámci kontinuální integrace, tedy je třeba, aby byla jeho obsluha co nejjednodušší a šla snadno automatizovat. Proto nebude vyvíjeno žádné grafické uživatelské rozhraní a program bude spouštěn z příkazové řádky s využitím vstupních parametrů pro rozlišení jednotlivých funkcí tohoto systému.

6.2 Způsob testování

Jelikož bude testování probíhat formou černé skřínky a detaily implementace jsou skryty, bude testování prováděno z širšího pohledu. K ověření korektní funkce filtrů bude třeba využít princip, na kterém stojí architektura *SmartCGMS* - posílání zpráv. Na začátku bude vytvořena událost a ta bude předána filtru prostřednictvím jeho metody `Execute`. Výstup bude očekáván v navázaném filtru. Proto musíme implementovat vlastní terminální testovací filtr, který bude na právě testovaný filtr navázán. To nám umožní podle výstupu analyzovat funkci testovaného filtru. Ostatní entity, které nevrací výstup do navázané entity, budou testovány pouze zavoláním příslušné metody. Její výstup pak bude porovnán s předpokládaným výsledkem.

6.3 Hierarchie testovacích tříd

Pro každou z testovaných entit je třeba vytvořit vlastní testovací třídu se sadou testů, specifických pro ni. Tyto třídy však budou mít nějaké společné

chování, jako spouštění jednotlivých testů, tudíž budou seskupeny do hierarchie, viz obrázek 6.1.

Testy lze spouštět v hlavním vlákně, ve vedlejším vlákně nebo jako samostatný proces. Pro tuto práci bude zvoleno spouštění ve vedlejším vlákně, které umožní kontrolu překročení časového limitu testu. Navíc, pokud nastane kritická chyba, jako například neoprávněný zásah do paměti, automaticky to povede k selhání testů.

Všechny entity jsou distribuovány v různých modulech ve formě sdílených knihoven. Pro sdílené knihovny bude také implementována vlastní sada testů.

6.4 Testování modulů

Moduly se ve *SmartCGMS* rozumí sdílené knihovny. Moduly distribuují desktiptory entit. Aby bylo možné tyto deskriptory číst a podle nich načítat příslušné entity, moduly exportují funkce, jejichž symboly lze načíst za běhu programu do paměti a s jejich pomocí deskriptory nahrávat. Ke každé metodě pro výběr deskriptorů existuje i tovární metoda, se kterou lze příslušnou entitu vytvořit, viz tabulka 6.1. Základem pro testování modulů tedy bude načítání exportovaných symbolů a ověřování jejich funkčnosti.

Metoda pro výběr deskriptorů	Tovární metoda
`do_get_filter_descriptors`	`do_create_filter`
`do_get_signal_descriptors`	`do_create_signal`
`do_get_metric_descriptors`	`do_create_metric`
`do_get_solver_descriptors`	`do_solve_generic`
`do_get_approximator_descriptors`	`do_create_approximator`

Tabulka 6.1: Metody pro čtení deskriptorů entit a jim odpovídající tovární metody.

6.4.1 Testování metod pro čtení deskriptorů

Základním požadavkem na moduly je to, že musí exportovat alespoň jednu `do_get_*_descriptors` metodu. Pokud by žádnou neexportoval, bylo by třeba předem znát `GUID` jednotlivých entit, které distribuuje, aby ho bylo možné vůbec použít. Navíc by nebylo možné získat další podrobnosti o této entitě.

6.4.2 Testování továrních metod modulů

Pokud je některá z *descriptors* metod nalezena, musí modul exportovat i odpovídající tovární funkci, aby bylo možné danou entitu zkonstruovat. Aby nedocházelo k nepředvídatelnému chování, tovární funkce nesmí umožnit vytvoření entity, která není mezi exportovanými deskriptory. V tom případě musí vrátit chybový kód. Samozřejmostí je předpoklad korektní validace vstupních parametrů tovární metody, aby systém nehavaroval při nestandardním vstupu.

6.5 Testování metrik

Jak už bylo zmíněno v kapitole 3.1, metrika je zobrazení. Tato funkce musí splňovat tři základní matematické předpoklady, které budou nástrojem SmartTester testovány:

- **totožnost** - pokud je referenční a vypočtená hodnota v jeden čas stejná, hodnota metriky je nulová

$$d(a, a) = 0$$

- **symetrii** - metrika mezi dvěma množinami hodnot je v jeden čas stejná i tehdy, pokud zaměním jejich význam z hlediska referenční a vypočtené hodnoty

$$d(a, b) = d(b, a)$$

- **trojúhelníkovou nerovnost**

$$d(a, c) \leq d(a, b) + d(b, c)$$

6.6 Testování aproximátorů

Aproximátor by určitě měl umět vrátit aproximaci v polovině intervalu (například v čase 10.5 mezi časy 10 a 11). To samé platí pro derivaci v tomto bodě. Jelikož každý aproximátor výsledné hodnoty vrací na základě svojí vlastní vnitřní logiky, která není předem známa, nebude testováno to, zda-li aproximátor vrátí konkrétní hodnotu nebo hodnotu v nějakém očekávaném intervalu. Očekáváme pouze to, že aproximátor vrátí nějaký nenulový výsledek.

6.7 Obecné testování rozhraní filtru

Základ architektury *SmartCGMS* je postaven na řetězu entit, implementujících rozhraní filtru. Tyto entity jsou schopné přijmout událost, interně ji zpracovat a následně ji předat navázanému filtru. Každá událost má přiřazený jednoznačný kód, který říká, k čemu slouží. Dále obsahuje další atributy, jako je `GUID` zařízení, které ji vyprodukovalo, fyzické hodiny, logický čas a další.

Informativní události, tedy události `Information`, `Warning` a `Error`, by měly každým filtrem projít bez toho, aniž by byly nějak modifikovány kterékoliv jejich atributy. To samé platí pro `Warm_Reset` a `Shut_Down` události.

6.8 Testování modelů

Stejně jako ostatní entity, i modely jsou reprezentovány deskriptory, které lze z modulu číst. Tento modul pak disponuje i příslušnou tovární metodou pro jejich vytvoření. *SmartCGMS* podporuje pouze vytváření diskrétních modelů. Pokud má tedy model v deskriptoru informaci o tom, že se jedná o diskrétní typ, měl by jít pomocí tovární metody zkonstruovat. Naopak, pokud se jedná o spojitý model, nesmí jít touto metodou vytvořit.

Při jeho konstrukci je modelu předáván vektor číselných parametrů, které dále ovlivňují jeho chování. Velikost tohoto vektoru je taktéž uložena v deskriptoru. Model pak musí jít vytvořit pouze tehdy, pokud je velikost vstupního vektoru parametrů totožná s očekávanou délkou.

Model implementuje rozhraní filtru, tudíž musí mimo jiné projít všemi obecnými testy tohoto rozhraní, které už byly zmíněny v kapitole 6.7.

Navíc implementuje rozhraní `scgms::IDiscrete_Model`, které vyznačuje diskrétní modely. Poskytuje jim dvě základní funkce:

- **`Initialize`** - provede inicializaci modelu v referenčním čase pro určitý segment (celé číslo)
- **`Step`** - provádí krokování modelu o předaný časový úsek, čímž posouvá jeho stav do budoucího času

Inicializace smí být provedena nejvýše jednou. Pokud by k ní docházelo opakovaně, systém by nebyl schopen korektně reagovat.

Při krokování dochází k odesílání událostí, reprezentujících stav modelu v daném časovém okamžiku. Simulace musí postupovat v čase dopředně,

proto krokování se zápornou časovou hodnotou musí skončit chybou. Pokud je provedeno krokování s časovým rozdílem 0, musí být vždy uvolněna sada událostí, která reprezentuje aktuální moment. Při krokování s kladnou hodnotou budou uvolněny události pro čas o tolik větší.

6.9 Logování

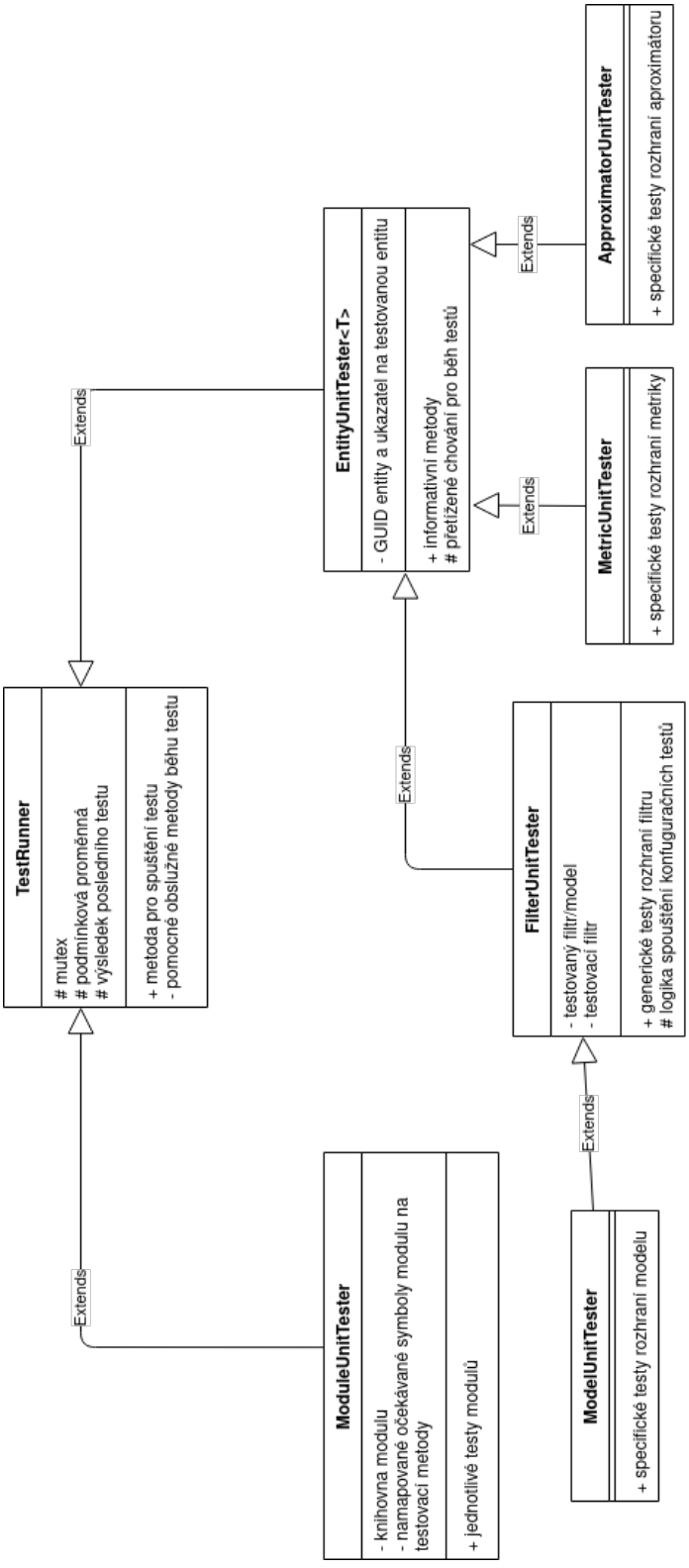
Jednou ze zásadních funkcí testovacího nástroje je podávat uživateli zprávy o tom, které testy proběhly správně a které ne, případně zobrazit reálné výsledky a porovnat je s očekávanými hodnotami. Proto bude implementován systém logování ve formě jednoduchého loggeru, který bude zaznamenávat detailní informace o běhu programu do výstupního souboru. Zde budou zaznamenávány nejen výsledky jednotlivých testů, ale také průběhy dílčích procedur, detailní informace pro účely ladění a průběhy všech testů. V případě selhání některého z testů zde budou zaznamenány obdržené výsledky testovací procedury a také očekávané výsledky. Toto logování bude rozděleno do několika úrovní podle závažnosti a důležitosti zaznamenávaných událostí.

- `DEBUG` - Záznamy na úrovni `debug` budou určeny pouze pro ladění. Mohou to být například hodnoty vstupních parametrů do jednotlivých dílčích funkcí. Tyto záznamy budou v produkčním prostředí vynechány.
- `INFO` - Informativní záznamy budou určeny pro uživatele tohoto nástroje k informování o obecném průběhu testů.
- `WARN` - Úroveň `warn` bude indikovat libovolné neočekávané události, které mohou mít vliv na výsledek testů nebo na chování aplikace. Tyto události nejsou ale natolik kritické, aby kvůli nim byly testovací procedury zastaveny.
- `ERROR` - Chybové hlášky budou popisovat kritické události, které bezprostředně předchází selhání testů nebo ukončení systému.

6.10 Výběr nástroje pro kontinuální integraci

Pro správu sestavení *SmartCGMS* bude využit nástroj ***Jenkins***. Je to open-source software, který poskytuje mnoho možností instalace a běhu na různých platformách. Navíc podporuje běh vlastních skriptů pro sestavení projektů, které jsou ušité na míru našemu softwaru. Proto se jedná o ideální volbu.

Aby bylo možné tento systém sestavení a testů snadno migrovat na jiné zařízení, pokud by to bylo třeba, bude spouštěn v rámci *Docker* kontejneru, který je postaven nad hardwarem zařízení, na kterém běží a nahrazuje celý operační systém a běhové prostředí [5]. Obraz tohoto kontejneru bude postaven nad obrazem platformy *Debian GNU/Linux 10 (buster)*. Systém *SmartCGMS* využívá externí knihovny, které musí být pro správný překlad nainstalovány. Proto bude vytvořen vlastní Docker obraz pomocí `Dockerfile` [6], ve kterém budou všechny tyto závislosti manuálně nainstalovány. Zároveň nám to umožní do kontejneru vložit vlastní skripty, potřebné pro sestavení a testování *SmartCGMS*.



Obrázek 6.1: Zjednodušený model hierarchie testovacích tříd nástroje Smart-Tester.

7 Implementace nástroje

V této části bude představena implementace klíčových částí vyvíjeného nástroje. Bude zde ukázán postup akcí, které systém po spuštění vykoná v závislosti na předaných parametrech.

7.1 Vstupní bod aplikace

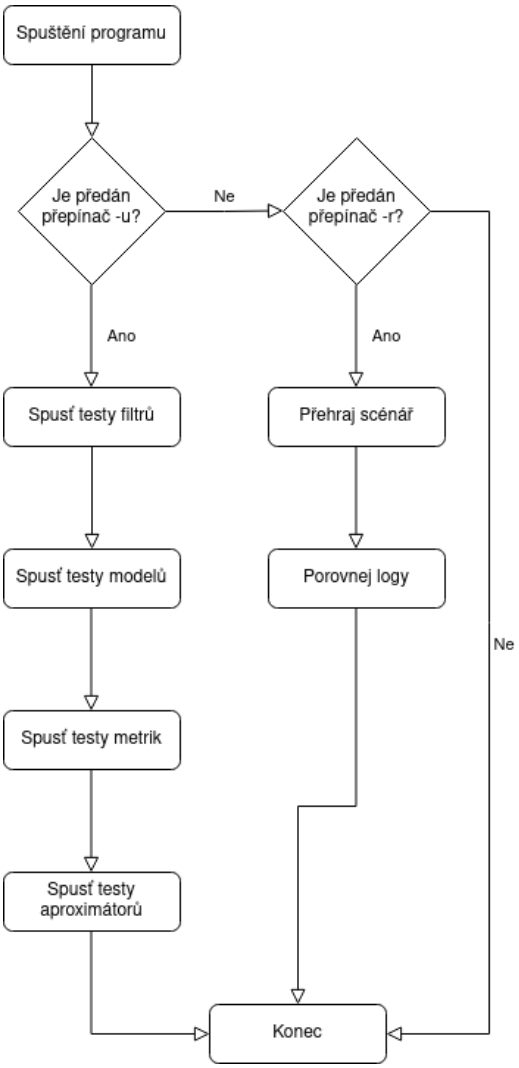
SmartTester při startu očekává předané vstupní parametry. Předán musí být minimálně přepínač, který systému sdělí, kterou sadu testů má vykonat. Spuštěna může být buď sada jednotkových testů, kdy budou otestovány všechny dostupné moduly a entity, které jsou v těchto modulech distribuované, nebo regresní testy. Při testech regrese je očekáván další parametr a to cesta ke scénáři, který se má přehrát. Celý běh programu je znázorněn na obrázku 7.1. Pro jednotkové testy existuje ještě alternativní varianta spuštění, kdy je předáno i `GUID` filtru, který má být otestován. Následně budou spuštěny pouze testy tohoto konkrétního filtru a modulu, ve kterém je filtr obsažen. V nástroji pro kontinuální integraci se očekává výhradně využití kompletní sady jednotkových testů.

7.2 Integrace se SmartCGMS

Aby bylo možné vůbec testovat, je třeba mít informace o tom, jak vypadá aplikační rozhraní jednotlivých komponent testovaného systému. Rozhraní potřebné pro implementaci všech testů je obsaženo v sadě vývojových nástrojů `common` systému *SmartCGMS*. Vyvíjený nástroj je na těchto nástrojích přímo závislý, jelikož potřebuje pracovat s třídami a rozhraními, které jsou zde definovány. Sada vývojových nástrojů `common` obsahuje podpůrné implementace nebo základy implementací pro snazší vývoj. Většina implementace je ale nedostupná a je vyvíjena samostatně pomocí implementací v `common`.

7.3 Spouštěč testů

Základem celé hierarchie testovacích tříd je spouštěč testů `TestRunner`. Spouštěč disponuje funkcemi, potřebnými pro spouštění jednotlivých testovacích metod. Pro správné otestování jednotlivých entit je třeba testovací



Obrázek 7.1: Vývojový diagram zobrazující průběh spuštění nástroje SmartTester.

rutiny spouštět v samostatném vlákně pro případ, že testovaný kód nedokončí běh do stanoveného časového limitu. Ten je nastaven na jednu sekundu. Všechny definované testy nejsou výpočetně náročné a delší čas nesmí vyžadovat. Z tohoto důvodu spouštěč obsahuje i logiku obsluhy paralelního provádění. Konkrétně synchronizaci s hlavním vláknem a kontrolu překročení časového limitu. Výsledky těchto testů také vypisuje do logu a do konzole.

7.4 Tester modulů

Pro otestování sdílených knihoven byla vytvořena třída jako potomek spouštěče testů. `ModuleUnitTester` obsahuje zásadní informace o právě testova-

ném modulu, jako je cesta k němu nebo přístupový bod k dekorátoru sdílené knihovny. Instance testeru modulů jsou vytvářeny až v průběhu testování komponent modulu. Proto dále obsahuje informace o tom, které moduly už byly v daném běhu nástroje otestovány, aby nedocházelo k opakovanému testování stejného modulu.

7.5 Tester entit

Primárním cílem této práce je testování entit distribuovaných v samostatných modulech, zmíněných výše. Proto je implementován `EntityUnitTester`, který toto testování spravuje v obecné rovině. Jelikož se proces liší od jednoduššího testování obecných modulů, je tento tester vytvořen jako potomek obecného spouštěče testů. Ten přetěžuje metody pro spouštění testů tak, aby vyhovovaly požadavkům testů entit. Oproti základnímu spouštěči testů je obohacen o informace o testované entitě a modulu, ve kterém je entita distribuována. Tester entit je implementován pomocí generického programování, konkrétně pomocí šablon jazyka `C++`. Je tedy možné ho využít pro testování libovolné entity, která je v systému *SmartCGMS* distribuována. V systému tuto implementaci využívají `MetricUnitTester` pro testování metrik, `ApproximatorUnitTester` pro testování aproximací a `FilterUnitTester`, který ověřuje funkčnost filtrů.

7.6 Testovací filtr

Základním předpokladem pro vytvoření kteréhokoli filtru je předání výstupního filtru, do kterého budou odesílány zpracované události. Proto je implementován `TestFilter`, který slouží jako testovací náhrada výstupního filtru. Tím, že implementuje rozhraní filtru, splňuje vlastnosti filtru, ale objekt se tak nemusí nutně chovat nebo být funkční[10]. Sám přitom další výstupní filtr pro konstrukci nevyžaduje. Všechny přijaté události během probíhajícího testu uchovává a poskytuje metody pro jejich čtení a manipulaci s nimi. Ty jsou pak využity pro kontrolu chování testovaného filtru a vyhodnocování testů.

7.7 Tester rozhraní filtru

Nejvíce zastoupeny jsou v systému *SmartCGMS* entity implementující rozhraní filtru `scgms::IFilter`. Tyto entity vyžadují speciální přístup, proto je třída `FilterUnitTester` postavena nad testerem entit a zároveň přetěžuje

chování spouštěče testů tak, že doplňuje logiku načítání entit před spuštěním testu a po skončení testu je uvolňuje. K vytvoření filtru je třeba do tovární metody předávat i filtr výstupní, do kterého vytvořený filtr bude odesílat události. V něm budeme kontrolovat chování testovaného filtru. Proto je uchováván i odkaz na instanci třídy `TestFilter`, který slouží jako náhrada za tento výstupní filtr.

Dalším obohacením je systém spouštění konfiguračních testů. Každá entita, implementující rozhraní filtru, je konfigurovatelná přes toto rozhraní. Konfigurační testy jsou pro každou entitu v zásadě totožné, pouze s rozdílným vstupem. Proto je tento mechanismus implementován jako parametrizovaný test, který přebírá obecnou konfiguraci, a je dostupný pro všechny odvozené třídy.

V neposlední řadě obsahuje `FilterUnitTester` obecné testy pro všechny entity s rozhraním filtru. Těmito testy musí projít každá entita, implementující rozhraní filtru a jsou spouštěny pokaždé v rámci testovací sady.

Rozhraní filtru ve *SmartCGMS* implementují entity `Filter` a `Model`. Pro výčet filtrů, které budou v této práci testovány, jsou implementovány specifické třídy, odvozené od třídy `FilterUnitTester`, které obsahují specifické testy pro konkrétní filtry. Pro testování modelů je vytvořena třída `ModelUnitTester`, který spouští nejen obecné testy pro rozhraní filtru, jelikož ho modely implementují, ale i obecné testy rozhraní modelů.

7.8 Tester regrese

Pro spouštění regresních testů je implementována třída `RegressionTester`. Podle vstupní konfigurace přehraje zadaný scénář, který vyprodukuje výstupní log. Tento log pak porovná s přiloženým referenčním logem a rozhodne, zda-li byla do systému zanesena regrese, nebo ne.

V průběhu porovnávání jsou oba logy načteny do paměti a jejich záznamy jsou seřazeny podle logických hodin, aby byla zachována chronologická posloupnost událostí. Porovnávání funguje tak, že všechny záznamy z referenčního logu musí být nalezeny ve výstupním logu. Tento log může obsahovat záznamy navíc, které v referenčním logu nejsou, ale nesmí v něm chybět žádný očekávaný záznam. V rámci jednotlivých záznamů jsou porovnány všechny uvedené hodnoty, například kód události, logický čas, kód segmentu a podobně. Tyto hodnoty se musí v obou souborech shodovat. Pokud je nalezena neshoda, test selže a jsou vypsány záznamy, u kterých byla neshoda nalezena.

8 Implementace testů

V této kapitole bude představeno, jak jsou jednotlivé testy implementovány. V každém testu je ověřen požadavek na funkčnost entity. Při nesplnění požadavku pak konkrétní test selže.

8.1 Testy modulů

Tester modulů obsahuje mapované názvy symbolů pro načtení deskriptorů na jim odpovídající názvy symbolů továrních metod. Přítomnost alespoň jednoho symbolu v modulu je otestována tak, že jsou všechny postupně načítány. Pokud se ani jeden symbol pro načtení deskriptorů nepodaří načíst, test selže a žádné další testy nebudou vykonány. Nalezené symboly jsou uloženy, jelikož se podle nich vykonají další příslušné testy.

Podle nalezených symbolů pro načtení deskriptorů proběhne načtení příslušných symbolů továrních metod. Test selže, pokud tovární metoda není nalezena.

V testeru modulů jsou názvy symbolů pro načtení deskriptorů mapovány i na testy vytvoření příslušných entit a na testy validace vstupních parametrů těchto symbolů. V testech vytvoření entit je vždy načten symbol tovární metody a každá entita, jejíž deskriptor je nalezen, je pomocí tovární funkce vytvořena. Pokud její vytvoření selže, selže i celý test. Zároveň je v rámci testu ověřena validace vstupních parametrů. Pokud tovární metoda nevrátí chybový kód při nevalidním vstupním parametru, test selže.

V testu validace vstupních parametrů pro metody čtení deskriptorů je metodě předán jeden ukazatel místo dvou různých. Pokud metoda nevrátí chybový kód, test selže.

8.2 Testy metrik

V testu totožnosti metriky je zkonstruován vektor referenčních hodnot. Tento vektor je předán do metody `Acumulate` testované metriky jako vektor referenčních i vypočtených hodnot. Tím dojde ke zpracování předaných dat. Samotná metrika je pak vypočtena pomocí metody `Calculate`. Pokud není výsledná hodnota rovna nule, test selže.

Při testu symetrie jsou již vektory hodnot dva - vektor referenčních a vektor vypočtených hodnot. Ty obsahují navzájem různé hodnoty. Opět dojde

k předání vektorů do metody `Acumulate` a vypočtení hodnoty metriky metodou `Calculate`. Vektory jsou poté zaměněny a vektor referenčních hodnot je předán jako vektor hodnot vypočtených a naopak. Výsledná hodnota metriky je vrácena jako reálné číslo. Proto nejsou hodnoty těchto výpočtů testovány na rovnost, ale jejich rozdíl musí být menší než zvolené ϵ. Jeho hodnota je zvolena jako 0.0001.

Stejným postupem je otestován i předpoklad trojúhelníkové nerovnosti s třemi různými vektory hodnot. Pokud není trojúhelníková nerovnost splněna, test selže.

8.3 Testy aproximátorů

Tester aproximátorů nejprve vytvoří signál s deseti hodnotami v časech 1 až 10. Následně je vytvořen vektor s hodnotami časů v polovině intervalu sousedních časů (tedy 0.5, 1.5, 2.5 atd.). V těchto časech je nejdříve aproximována hodnota signálu a v dalším testu hodnota její derivace. Výpočet aproximace je vykonán metodou `Get_Levels`. Test selže, pokud při aproximaci dojde k chybě nebo není vrácena žádná hodnota.

8.4 Obecné testy rozhraní filtru

V testech informativních událostí je vždy nejdříve zkonstruována samotná událost tovární metodou. Pokud její vytvoření selže, například když se nepodaří načíst potřebnou sdílenou knihovnu, test selže. Vytvořená událost je poté předána do metody `Execute`. Pokud metoda vrátí chybový kód, test selže. Z navázaného testovacího filtru je přijatá událost vybrána a její kód je porovnán s kódem odeslané události. Pokud se kódy neshodují, test selže.

U testu `Shut_Down` události je tento proces opakován dvakrát. Ovšem při druhém odeslání události se již očekává, že metoda `Execute` vrátí chybový kód. Pokud ne, test selže.

Druhou formou obecných testů filtrů jsou testy konfigurační. Tento test je implementován jako parametrizovaný test, který jako vstupní parametry přebírá konfiguraci filtru a očekávaný výsledek konfigurace. Konfigurace je následně předána do metody `Configure` testovaného filtru. Pokud se návratová hodnota konfigurace neshoduje s očekávanou hodnotou, test selže.

8.5 Testy modelů

V testu příznaku modelu je načten deskriptor, ze kterého je zjištěno, jestli se jedná o diskrétní nebo spojitý model. Pokud v deskriptoru tato informace není, test selže a žádné další testy nejsou vykonány.

V následném testu vytvoření modelu je model zkonstruován tovární metodou. Jedná-li se o diskrétní model, jeho konstrukce musí uspět, jinak test selže. Pokud se jedná o spojitý model, jeho vytvoření tovární metodou nesmí uspět, jinak test selže.

Další testy jsou určeny pouze pro diskrétní modely. Následuje test nevalidního vektoru vstupních parametrů. Z deskriptoru je přečtena vyžadovaná velikost vektoru a je vygenerováno náhodné číslo od 1 do této vyžadované hodnoty. Díky tomu lze test zobecnit pro kterýkoli diskrétní model. Následně je model vytvořen tovární metodou, které je předán vektor parametrů o nevalidní velikosti. Test selže, pokud konstrukce modelu nevrátí chybový kód.

Inicializace modelu je prováděna s kladným výchozím časem. Inicializace musí projít úspěšně, jinak test selže. V dalším testu je model inicializován dvakrát za sebou. Druhá inicializace musí vrátit chybový kód, jinak test selže.

Model je krokován metodou `Step`. Ta smí být volána až v tu chvíli, kdy je model již inicializován. V dalším testu je metoda `Step` zavolána bez předchozího zavolání `Initialize` a je očekáváno vrácení chybového kódu.

Dále je metoda `Step` volána až po inicializaci modelu, ale se záporným časem krokování. V tom případě musí metoda vrátit chybový kód, jinak test selže.

Následují testy krokování s nulovou a kladnou časovou změnou. V obou testech je model inicializován a je zavolána metoda `Step` s odpovídající časovou změnou. Metoda nesmí vrátit chybový kód, jinak test selže. Jelikož model implementuje rozhraní filtru, je na něj navázán náš testovací filtr. Z něj jsou přečteny obdržené události po krokování. Pokud žádná událost do filtru nepřišla, test selže. Fyzické hodiny přijatých událostí se musí shodovat s hodnotou, o kterou byl model doposud krokován. Test selže, pokud se časy neshodují.

8.6 Testy konkrétních filtrů

Specifickou částí testování jsou testy konkrétních filtrů. Pro každý z nich je vytvořena vlastní specializovaná testovací třída.

8.6.1 Log filtr

Pro zaznamenávání událostí do výstupního souboru je určen **`LogFilter`**. Konfigurace zahrnuje pouze cestu k výstupnímu souboru. Ta musí být platná a uživatel musí mít oprávnění k zápisu. Výstupní soubor může a nemusí existovat. Případný existující soubor bude přepsán.

Každá událost, která tímto filtrem projde musí být zaznamenána ve výstupním souboru. Počet odeslaných událostí tedy musí odpovídat počtu záznamů. Události také musí být zaznamenány v takovém pořadí, v jakém do filtru přišly.

Filtr zaznamenává události i v paměti a z té je lze vybrat pomocí inspekce jeho rozhraní, které poskytuje metodu `Pop`. Volání `Pop` vždy vrátí seznam událostí, které ještě předchozím voláním nebyly z vyrovnávací paměti vybrány. Zároveň, pokud se to ještě nestalo, jsou zapsány do výstupního souboru. Nikdy nesmí dojít k tomu, že jedna událost bude voláním `Pop` vrácena dvakrát.

8.6.2 Filtr pro zpětné přehrání záznamů

Vygenerované výstupní logy je možné zpětně přehrát pomocí entity **`LogReplayFilter`**. Ten log analyzuje, ze záznamů vytvoří jednotlivé události a pošle je do dalších navázaných filtrů.

Filtr je třeba nakonfigurovat, podobně jako `LogFilter`, pomocí cesty ke vstupnímu souboru. Na této cestě musí mít uživatel právo ke čtení. Pokud zadaná cesta ukazuje na soubor, bude přehrán pouze tento soubor. Pokud cesta ukazuje na adresář, budou přehrány všechny validní logy uvnitř něj. Chyby při analýze vstupního souboru musí být filtrem hlášeny. Při chybě analýzy hlavičky musí být odeslána `Warning` událost, při chybě analýzy těla `Error` událost. Zároveň počet událostí musí odpovídat počtu záznamů ve vstupním souboru.

Konfigurace `LogReplay` filtru dále obsahuje parametry `Emit_Shutdown_Message` a `Interpret_Filename_As_Segment_Id`. Pokud je nakonfigurován první parametr, na konci přehrávání souboru je ještě uvolněna `Shut_Down` událost, aby došlo k automatickému ukončení všech filtrů. Při konfiguraci druhého zmíněného parametru bude pro každý přehrávaný soubor v adresáři nahrazeno `Segment_Id` odesílaných událostí na unikátní hodnotu.

8.6.3 Filtr pro vykreslování dat

SmartCGMS umožňuje vykreslování dat, reprezentovaných předávanými událostmi, do přehledných grafů ve formátu `SVG`. Tento proces je zajišťován entitou **`DrawingFilter`**.

V konfiguraci je možné filtru zadat několik cest, kde budou uložené výsledné grafy. Každá cesta se odvíjí od typu vykreslovaných dat. Všechny tyto cesty musí být validní, tedy uživatel musí mít právo zápisu a nesmí se jednat o složku. Zároveň musí všechny být navzájem odlišné, jinak by se grafy navzájem přepisovaly a docházelo by k chybám.

Po nakonfigurování filtru a přijmutí alespoň jedné události musí v cílových cestách vzniknout `SVG` soubory pro výsledná data.

8.6.4 Mapovací filtr

V řetězu filtrů je často třeba přemapovat jeden typ signálu na jiný. Například naměřenou hladinu glukózy na vypočtenou. Někdy je třeba signál úplně odstranit a zahazovat události, které ho nesou. Od toho je **`MappingFilter`**.

Konfigurovat lze zdrojový a cílový `GUID` signálu. Ty musejí nabývat platných hodnot pro konkrétní signál, nebo speciálních hodnot pro takzvaný *Null* a *All* signál.

V případě, že jsou nakonfigurované parametry konkrétní signály, po průchodu filtrem musí být všechny události se `Signal_Id` shodným se zdrojovým signálem, přemapovány na `GUID` cílového signálu. Pokud je cílovým signálem *Null* signál, tyto události jsou při průchodu filtrem zahozeny a do navázaného filtru již nesmí projít. V případě, že je jako zdrojový nastaven signál *All*, mapovány musí být všechny příchozí události.

8.6.5 Maskovací filtr

Událost typu `Level` nese nově naměřenou nebo vypočtenou hladinu signálu. Tyto události jsou pak zohledněny při výpočtech nebo hledání parametrů modelů. Pokud je třeba některé tyto události z výpočtů vynechat, je třeba je přeměnit na typ `Masked_Level`. Ten je ze všech výpočtů vynechán. Tuto přeměnu zajišťuje **`MaskingFilter`**.

V konfiguraci je třeba filtru předávat `GUID` signálu, jehož události chceme maskovat a bitmasku, reprezentující sekvenci, podle které mají být tyto události maskovány. Pokud je bit nulový, událost bude maskována. Pokud je nastavený, maskování události bude přeskočeno. `GUID` signálu musí být vždy validní konkrétní signál a bitmaska musí být vždy o délce násobků osmi, jelikož se interně ukládá jako posloupnost bitů v jednom nebo více bytech.

8.6.6 Generátor signálu

Pro správnou funkčnost kteréhokoliv modelu je třeba, aby byl vytvořen v rámci generátoru signálu. `SignalGeneratorFilter` interně vytváří model a obaluje ho vlastní logikou.

V rámci konfigurace je třeba generátoru vždy předat `GUID` modelu, podle kterého bude generovat signál. To musí být vždy validní, přiřazeno konkrétnímu modelu. Dále jsou konfigurovány hodnoty `Stepping` a `Maximum_Time`, které udávají časový úsek, o který bude model vždy krokován a maximální čas, do kterého bude krokování probíhat. Z logiky věci musí být akceptována pouze kladná hodnota pro krokování a pro maximální čas vyšší hodnota než ta, zvolená pro krokování, aby došlo alespoň k jednomu posunu v čase.

Model uvnitř generátoru může být vzhledem k signálu **synchronní** nebo **asynchronní**. Pokud je konfigurován asynchronně, po konfiguraci generátoru bude model automaticky krokován o hodnotu `Stepping` tak dlouho, dokud nepřekročí hodnotu `Maximum_Time`. Po překročení této hodnoty se musí generátor ukončit.

Při synchronní konfiguraci není model krokován automaticky, ale v závislosti na událostech, posílaných do generátoru. Ke krokování modelu dojde pouze tehdy, pokud do generátoru přijde událost v čase alespoň o `Stepping` vyšším, než poslední událost, při které ke krokování došlo. Při ostatních dílčích událostech nesmí být model krokován. Krokuje se buď na konkrétní signál nebo na všechny signály. Pokud chceme krokovat na jeden konkrétní signál, je třeba generátor nakonfigurovat s odpovídajícím `GUID`. Při krokování na všechny signály je třeba filtr konfigurovat hodnotou *All*.

8.7 Regresní testy SmartCGMS

V rámci regresních testů jsou přehrávány 3 referenční scénáře. Ke každému scénáři je přiložen log, se kterým bude porovnán výstupní log přehraného scénáře. Všechny 3 přiložené scénáře a jejich referenční logy jsou validní a korektně funkční systém by měl těmito testy projít.

Ve scénářích je například simulováno monitorování pacienta po dobu jednoho dne, nebo průběh jeho léčby. V rámci toho je realizováno zpětné přehrávání výstupních záznamů, generování událostí podle modelu nebo například jejich maskování. Jde o vytvoření, kombinaci a spolupráci více entit. Proto částečně slouží i jako náhrada integračních testů.

Tyto testy jsou pro ověření celkové funkčnosti systému velmi důležité. Při zavlečení regrese je možné, že testovaný systém projde všemi jednotkovými testy, ale stejně obsahuje chybu.

9 Integrace do nástroje Jenkins

Pomocí uživatelského rozhraní *Jenkins* je vytvořen projekt, v rámci kterého lze spustit celý proces sestavení a otestování *SmartCGMS*. Sestavení je realizováno vlastním skriptem, který stáhne všechny potřebné vzdálené repozitáře a systém sestaví. Další fází tohoto procesu je otestování pomocí nástroje *SmartTester*. Ještě před spuštěním implementované sady testů je zkontrolována aktuální verze SmartTesteru. Pokud je mezi vydanými verzemi v repozitáři projektu nalezena novější verze nástroje, je stažena a rozbalena. Poté dojde k přesunutí zkompilovaných binárních souborů *SmartCGMS* do nástroje *SmartTester* a jsou spuštěny jednotkové a všechny regresní testy. Pokud alespoň jeden z těchto testů neprojde, Jenkins to dokáže rozeznat pomocí specializovaného zásuvného modulu, který na základě regulárního výrazu analyzuje konzolový výstup testovacího procesu. Je-li nalezena chybová hláška, projekt je označen za nestabilní.

Spouštění tohoto procesu je možné po dohodě s vedoucím práce pouze manuálně.

9.1 Nasazení pomocí Docker obrazu

Pomocí `Dockerfile` je vytvořen nový Docker obraz, postavený nad oficiálním obrazem nástroje Jenkins. Ten je následně zveřejněn do vzdáleného repozitáře obrazů *Docker Hub*[1]. Odtud je možné ho stáhnout pod označením `markovda/jenkins-smarttester:latest`. Pro spuštění je třeba mít nainstalován *Docker* verze alespoň `20.10`, se kterou byl obraz sestaven. Při spuštění kontejneru je třeba ho zpřístupnit na dostupném portu. Celé spuštění lze provést příkazem `docker run -u root -p 8080:8080`. Webové rozhraní nástroje Jenkins je pak dostupné na síťové adrese zařízení, na kterém je spuštěn, na zvoleném portu. Na této adrese je nutné při prvním spuštění nastavit nástroj Jenkins podle pokynů průvodce a vytvořit projekt. V projektu již lze v rámci procesu sestavení spouštět vytvořené skripty, které provedou sestavení a otestování *SmartCGMS*.

[1] Vzdálený repozitář Docker obrazů. Obdoba GitHub pro Docker obrazy.

10 Diskuse

V této kapitole budou prezentovány dosažené výsledky. Dále bude zhodnocena implementace nástroje a a budou popsány jeho nedostatky a omezení.

10.1 Dosažené výsledky

Na základě výsledků testování momentálního stavu *SmartCGMS* lze konstatovat, že systém má hlavně mezery ve validování vstupních parametrů všech rozhraní a konfigurací. Celkový přehled provedených testů je uveden v příloze A.

10.1.1 Obecné testy modulů a entit

Metody pro výběr deskriptorů entit vyžadují jako parametry dva ukazatele, označující začátek a konec pole. Tyto ukazatele jsou pouze nasměrovány do paměti, kde jsou deskriptory uloženy. Pokud je ale na místě obou parametrů předán ten samý ukazatel, tedy ukazatel počátku se rovná ukazateli konce, metoda i tak nevrátí chybový kód. Vynechaná validace vstupních ukazatelů je přítomna i v továrních metodách entit. Každá tovární metoda vyžaduje ukazatel na entitu jako vstupně výstupní parametr. Tento parametr ale vůbec není validovaný a pokud je místo ukazatele předán `nullptr` (ukazatel nikam), aplikace spadne. Toto chování se projevuje při vytváření kterékoli entity. Stejně tak při vytváření filtrů je povinným parametrem ukazatel na další filtr, na který se má vytvářený navázat. Pokud místo něj pošleme opět `nullptr`, filtr je i přesto vytvořen korektně bez chybového kódu.

Všechny metriky, kromě jedné, prošly všemi třemi testy. Pouze metrika *Crosswalk* selhala při testu identity.

Aproximátory jsou v systému *SmartCGMS* celkem 3. Dva prošly všemi testy.

Obecné testy rozhraní filtru dopadly u všech entit totožně. Informativní události se chovaly vždy správně. Testem funkčnosti `Shut_Down` události neprošla žádná entita. Po průchodu této události byly všechny i nadále funkční a umožňovaly průchod dalších událostí.

Spojité modely měly pouze požadavek toho, že je nesmělo být možno vytvořit. Tento požadavek všechny splňují. Diskrétní modely všechny selhaly v testu nevalidního počtu číselných parametrů ve vstupním vektoru parametrů. Zásadní opravy vyžaduje *Bolus calculator model*, jelikož kromě jednoho

selhal ve všech testech modelů.

10.1.2 Testy konkrétních filtrů

Log filtr Log filtr prošel všemi specifickými testy, kromě jednoho. Selhal při testu konfigurace prázdnou cestou k výstupnímu souboru.

Filtr pro zpětné přehrání logů Tento filtr neprošel řadou testů. V první řadě selže konfigurace cestou k validnímu adresáři. Následně kvůli tomu selžou i ostatní testy, které jsou na této konfiguraci závislé. Při chybě analyzování hlavičky vstupního souboru jsou `Warning` události odesílány poměrně nedeterministicky. `Error` události při nevalidním obsahu nejsou odeslány nikdy. Odeslání `Shut_Down` události po dokončení čtení funguje správně.

Filtr vykreslování dat Filtr vykreslování nejvíce trpí na špatně validované konfiguraci. Selhal ve všech testech nevalidních cest, ať už se jednalo o prázdnou cestu, o cestu, kde uživatel nemá právo zápisu (například `/root`), nebo o předání totožných cest k několika grafům.

Mapovací filtr Tento filtr selhal pouze na validaci konfiguračních parametrů. Při konfiguraci nevalidními nebo žádnými signály nevrací chybové kódy. Funkčními testy filtr prošel v pořádku.

Maskovací filtr Filtr pro maskování událostí nedopadl nejlépe z hlediska funkčních testů. Události, které jím prošly se vrátily s označením, které neodpovídalo ani jedné vyjmenované hodnotě. Chybné konfigurace filtr odhalil správně, pouze vracel nesprávnou chybovou hodnotu.

Generátor signálu Základní nalezenou chybou generátoru signálu bylo pravidelné spadnutí systému při asynchronní konfiguraci. Z tohoto důvodu se nepodařilo otestovat generátor v asynchronním stavu. V synchronním módu funguje generátor bez chyb. Stejně jako ostatní filtry ale opět špatně validuje vstupní parametry konfigurace. Umožnil například konfiguraci se záporným časem krokování, záporným maximálním časem, nebo maximálním časem menším, než je krokování.

10.1.3 Regresní testy

Systém prošel regresními testy napříč všemi dodanými binárními distribucemi v průběhu realizace práce.

10.2 Zhodnocení

Systém *SmartTester* je založen na hierarchii testovacích tříd, kdy pro každou konkrétní entitu je třeba vytvořit odvozenou třídu. Tato hierarchie je vytvořena dostatečně obecně na to, aby bylo možné pro přidání testů další entity pouze jednoduše vytvořit novou odvozenou třídu a implementovat sadu testů. Přidání nových testů k již existujícím entitám je také možné pouze jednoduchým doplněním nové testovací metody.

Systém prošel přísnou kontrolou překladače `Clang`, který pomocí konfiguračních proměnných, jako jsou `-Wall`, `-Wextra`, apod. kontroluje zdrojový kód a upozorňuje na logické chyby. Tyto kontroly zaručují určitou kvalitu kódu pro případná další rozšíření.

Program stavěl na základech vyvinutých v rámci předmětu KIV/ZSWI, kdy bylo předpokládáno pouze testování entit typu filtr. Tomu také byla přizpůsobena architektura systému. Ten primárně hledá filtry k testování a vyžaduje předem známé některé údaje, jako například název modulu, ve kterém se nachází. Do budoucna by bylo vhodné architekturu pozměnit tak, aby cílila místo toho primárně na testování modulů, které bude cyklicky procházet v předem definované složce, a cyklicky otestuje všechny nalezené entity v tomto modulu.

10.3 Nedostatky a omezení

Po dohodě s vedoucím práce nebyly implementovány testy pro několik filtrů. Zároveň nástroj neumí otestovat entity typu `Solver` ani `Signal`.

11 Závěr

V této práci se podařilo vytvořit nástroj, který implementuje sadu jednotkových a regresních testů systému *SmartCGMS*. Následně byl integrován do vybraného nástroje pro kontrolu kontinuální integrace.

V teoretické části práce pojednává o programátorském rozhraní testovaného programu a jeho funkčnosti. Následují základní informace o problémové doméně.

Dále byly prostudovány zásadní informace o testování softwaru. Byly popsány jednotlivé přístupy k testování a typy testů, které bylo třeba v rámci práce realizovat. V této části došlo ke zvolení přístupu, který bude pro implementaci nástroje využíván.

Byla sepsána rešerše několika nástrojů pro správu automatických sestavení a automatizovaných testů v rámci kontinuální integrace. Z nich byl vybrán ten nejvhodnější pro dané řešení, do kterého byl nakonec program začleněn.

V návrhu řešení byl zvolen programovací jazyk a vhodné nástroje k implementaci zadaného problému. Byl navrhnut základní model aplikace a přístup k řešení.

V implementaci byla vytvořena hierarchie testovacích tříd pro spouštění jednotkových testů nad individuálními entitami ze systému *SmartCGMS* a vedlejší funkcionalita pro spouštění definovaných scénářů v rámci regresních testů. Vyvinutý nástroj byl nakonec začleněn do systému *Jenkins*, který umožňuje spouštění procesu sestavení a otestování sledovaného systému. Zadání bylo splněno v celém rozsahu.

V závěru byly shrnuty dosažené výsledky testování a bylo konstatováno, kde jsou v testovaném systému nejčastější chyby.

Seznam zkratek

CGM Continuous Glucose Monitoring. 3–6, 8

CGMS Continuous Glucose Monitoring System. 5

CI Continuous Integration. 15–17

DM Diabetes Mellitus. 2, 3

GDM Gestační Diabetes Mellitus. 3

GUID Globally Unique Identifier. 6, 7, 20, 22, 26, 34, 35

HLA High-Level Architecture. 5

IEEE Institute of Electrical and Electronics Engineers. 5

SVG Scalable Vector Graphics. 34

Literatura

[1] American_Diabetes_Association: Gestational Diabetes mellitus. *Diabetes Care*, roèník 27, Leden 2004: str. 88, [cit. 2021-01-26].

[2] American_Diabetes_Association: Diagnosis and Classification of Diabetes Mellitus. *Diabetes Care*, roèník 28, Leden 2005: s. 37–42, [cit. 2021-01-26].

[3] AMMANN, P.; OFFUTT, J.: *Introduction to Software Testing*. Cambridge University Press, 2016, ISBN 9781316773123, [cit. 2021-01-27].
URL `<https://books.google.cz/books?id=58LeDQAAQBAJ>`

[4] DAHMAN, J.: The Department of Defense High Level Architecture. *Proceedings of the 1997 Winter Simulation Conference*, 1997: s. 142–149, doi:10.1145/268437.268465 ISBN 078034278X, [cit. 2021-01-26].

[5] DOCKER_Inc.: Docker overview. `<https://docs.docker.com/get-started/overview/>`, 2021, [Online], [cit. 2021-04-22].

[6] DOCKER_Inc.: Dockerfile reference. `<https://docs.docker.com/engine/reference/builder/>`, 2021, [Online], [cit. 2021-04-22].

[7] FOWLER, M.: Continuous Integration. `<https://martinfowler.com/articles/continuousIntegration.html>`, 5 2006, [Online], [cit. 2020-10-28].

[8] GIULIANO, D.; CERIELLO, A.; ESPOSITO, K.: Glucose metabolism and hyperglycemia. *The American Journal of Clinical Nutrition*, roèník 87, Leden 2008, [cit. 2021-01-28].

[9] GRAHAM, D.; VEENENDAAL, E. V.; EVANS, I.: *Foundations of Software Testing: ISTQB Certification*. Course Technology Cengage Learning, 2008, ISBN 9781844809899, [cit. 2021-01-27].
URL `<https://books.google.cz/books?id=Ss62LSqCa1MC>`

[10] HEROUT, P.: *Testování pro programátory*. Kopp, 2016, ISBN 978-80-7232-481-1, [cit. 2021-04-28].

[11] HUIZINGA, D.; KOLAWA, A.: *Automated Defect Prevention: Best Practices in Software Management*. Wiley-IEEE Computer Society Press, Srpen 2007, ISBN 978-0-470-16516-4, [cit. 2021-01-27].

[12] Jenkins_IO: Creating your first Pipeline. <https://www.jenkins.io/doc/pipeline/tour/hello-world/>, Listopad 2020, [Online], [cit. 2021-01-27].

[13] Jenkins_IO: Jenkins User Documentation. <https://www.jenkins.io/doc/>, Listopad 2020, [Online], [cit. 2021-01-27].

[14] KANER, C.: Exploratory testing. *Quality Assurance Institute Worldwide Annual Software Testing Conference*, Listopad 2006, [cit. 2021-01-27].

[15] KANER, C.; FALK, J.; NGUYEN, H. Q.: *Testing Computer Software, 2nd Ed.* New York, et al.: John Wiley Sons Inc., 1999, iSBN 978-0-471-35846-6, [cit. 2021-01-27].

[16] Kitware: About CMake. <https://cmake.org/overview/>, 2021, [Online], [cit. 2021-04-15].

[17] KOUTNÝ, T.; ÚBL, M.: Parallel software architecture for the next generation of glucose monitoring. *Procedia Computer Science*, roèník 141, 2018: s. 279–286, [cit. 2021-01-26].

[18] KOUTNÝ, T.; ŠIROKÝ, D.: nalyzing energy requirements of meta-differential evolution for future wearable medical devices. *World Congress on Medical Physics and Biomedical Engineering 2018, Prague, Czech Republic*, [cit. 2021-01-26].

[19] LEACH, P.; MEALLING, M.; SALZ, R.: A Universally Unique IDentifier (UUID) URN Namespace. *Internet Engineering Task Force*, 2008, doi:10.17487/RFC4122, [cit. 2021-01-28].

[20] MURATA, T.; TSUZAKI, K.; YOSHIOKA, F.; aj.: The relationship between the frequency of self-monitoring of blood glucose and glycemic control in patients with type 1 diabetes mellitus on continuous subcutaneous insulin infusion or on multiple daily injections. *J Diabetes Investig.*, Listopad 2015, ;6(6):687–691. doi:10.1111/jdi.12362, [cit. 2021-01-26].

[21] TRAN, E.: Verification/Validation/Certification. *Carnegie Mellon University*, 1999, [cit. 2021-01-27].

[22] VOKURKA, M.; et al.: *Patofyziologie pro nelékařské směry*. Praha: Univerzita Karlova v Praze, nakladatelství Karolinum, 2012, 233–235 s., [cit. 2021-01-26].

[23] ÚBL, M.; KOUTNÝ, T.: SmartCGMS as an Environment for an Insulin-Pump Development with FDA-Accepted In-Silico Pre-Clinical Trials. *Procedia Computer Science*, roèník 160, 2019: s. 322–329, [cit. 2021-01-26].

A Seznam provedených testů

Obecné testy modulů	
Název testu	Očekávaný výsledek
Test metod deskriptorů	Modul exportuje alespoň jednu metodu pro výběr deskriptorů
Test továrních metod	Modul exportuje všechny tovární metody, odpovídající nalezeným metodám výběru deskriptorů
Test vytvoření entit	Tovární metody dokáží vytvořit všechny entity, jejichž deskriptory modul obsahuje a validují vstupní parametry
Test validace parametrů metod deskriptorů	Všechny metody pro výběr deskriptorů validují vstupní parametry

Tabulka A.1: Testy modulů.

Výsledky testů modulů		
Název testu	Počet úspěchů	Počet selhání
Test metod deskriptorů	4	0
Test továrních metod	4	0
Test vytvoření entit	0	4
Test validace parametrů metod deskriptorů	0	4

Tabulka A.2: Výsledky testů modulů.

Obecné testy metrik	
Název testu	Očekávaný výsledek
Test kritéria totožnosti	Pokud jsou dodané vypočtené a referenční hodnoty stejné, metrika je 0
Test kritéria symetrie	Metrika, mezi dvěma množinami hodnot je v jeden čas stejná i tehdy, pokud zaměním jejich význam z hlediska referenční a vypočtené hodnoty.
Test trojúhelníkové nerovnosti	Metrika mezi množinou bodů A a B je menší nebo rovna součtu metrik mezi body B a C, a C a A.

Tabulka A.3: Testy metrik.

Výsledky testů metrik		
Název testu	Počet úspěchů	Počet selhání
Test kritéria totožnosti	12	1
Test kritéria symetrie	13	0
Test trojúhelníkové nerovnosti	13	0

Tabulka A.4: Výsledky testů metrik.

Obecné testy aproximátorů	
Název testu	Očekávaný výsledek
Test aproximace středu intervalu časového rozdílu	Aproximátor je schopen aproximovat hodnotu signálu v polovině časového intervalu mezi body A a B.
Test aproximace první derivace středu intervalu časového rozdílu	Aproximátor je schopen aproximovat první derivaci hodnoty signálu v polovině časového intervalu mezi body A a B.

Tabulka A.5: Testy aproximátorů.

Výsledky testů aproximátorů		
Název testu	Počet úspěchů	Počet selhání
Test aproximace středu intervalu časového rozdílu	2	0
Test aproximace první derivace středu intervalu časového rozdílu	2	0

Tabulka A.6: Výsledky testů Aproximátorů.

Obecné testy rozhraní filtrů	
Název testu	Očekávaný výsledek
Test `Information` události	Událost musí být odeslána do navázaného filtru a nesmí být nijak modifikována.
Test `Warning` události	Událost musí být odeslána do navázaného filtru a nesmí být nijak modifikována.
Test `Error` události	Událost musí být odeslána do navázaného filtru a nesmí být nijak modifikována.
Test `Warm_Reset` události	Událost musí být odeslána do navázaného filtru a nesmí být nijak modifikována.
Test `Shut_Down` události	Událost musí být odeslána do navázaného filtru a nesmí být nijak modifikována. Filtr poté nesmí přijímat další události.

Tabulka A.7: Obecné testy rozhraní filtrů. Výsledky jsou spojeny s testy diskrétních modelů a konkrétních filtrů.

Testy diskrétních modelů	
Název testu	Očekávaný výsledek
Test typu modelu	Každý model musí nést informaci o tom, že se jedná o diskrétní model.
Test vytvoření modelu	Model lze vytvořit pomocí tovární metody.
Test nevalidní velikosti vektoru vstupních parametrů	Model nelze vytvořit, pokud je mu předán vektor číselných parametrů o nesprávné délce.
Testy rozhraní filtru	Model musí projít všemi obecnými testy filtrů.
Test inicializace s kladným výchozím časem	Model musí jít inicializovat s kladným výchozím časem.
Test opakované inicializace	Opakovaná inicializace modelu musí vždy selhat.
Test krokování před inicializací	Model nesmí umožnit krokování před inicializací.
Test krokování se zápornou časovou změnou	Model nesmí umožnit krokování se zápornou časovou změnou.
Test krokování s nulovou časovou změnou	Model musí odeslat události reprezentující stav modelu v aktuálním čase.
Test krokování s kladnou časovou změnou	Model musí odeslat události reprezentující stav modelu v čase o tolik větším.

Tabulka A.8: Obecné testy diskrétních modelů.

Výsledky testů diskrétních modelů		
Název testu	Počet úspěchů	Počet selhání
Test typu modelu	4	0
Test vytvoření modelu	4	0
Test nevalidní velikosti vektoru vstupních parametrů	0	4
Test `Information` události	4	0
Test `Warning` události	4	0
Test `Error` události	4	0
Test `Warm_Reset` události	4	0
Test `Shut_Down` události	0	4
Test inicializace s kladným výchozím časem	4	0
Test opakované inicializace	3	1
Test krokování před inicializací	3	1
Test krokování se zápornou časovou změnou	3	1
Test krokování s nulovou časovou změnou	3	1
Test krokování s kladnou časovou změnou	3	1

Tabulka A.9: Výsledky testů diskrétních modelů.

Testy spojitých modelů	
Název testu	Očekávaný výsledek
Test typu modelu	Každý model musí nést informaci o tom, že se jedná o spojitý model.
Test vytvoření modelu	Model nelze vytvořit pomocí tovární metody.

Tabulka A.10: Obecné testy spojitých modelů.

Výsledky testů spojitých modelů		
Název testu	Počet úspěchů	Počet selhání
Test typu modelu	7	0
Test vytvoření modelu	7	0

Tabulka A.11: Výsledky testů spojitých modelů.

Testy entity `LogFilter`		
Název testu	Očekávaný výsledek	Výsledek
Test `Information` události	Událost musí být odeslána do navázaného filtru a nesmí být nijak modifikována.	OK
Test `Warning` události	Událost musí být odeslána do navázaného filtru a nesmí být nijak modifikována.	OK
Test `Error` události	Událost musí být odeslána do navázaného filtru a nesmí být nijak modifikována.	OK
Test `Warm_Reset` události	Událost musí být odeslána do navázaného filtru a nesmí být nijak modifikována.	OK
Test `Shut_Down` události	Událost musí být odeslána do navázaného filtru a nesmí být nijak modifikována. Filtr poté nesmí přijímat další události.	CHYBA
Test konfigurace prázdnou cestou	Konfigurace musí selhat.	CHYBA
Test konfigurace validní cestou	Konfigurace musí projít.	OK
Test konfigurace nevalidní cestou	Konfigurace musí selhat.	CHYBA
Test generace výstupního souboru	Po úspěšné konfiguraci musí v cílové cestě vzniknout výstupní soubor.	OK
Test počtu záznamů v logu	Počet záznamů v logu musí odpovídat počtu zpracovaných událostí.	OK
Test záznamu události	Každá zpracovaná událost musí mít v logu odpovídající záznam.	OK
Test pořadí záznamů událostí	Záznamy ve výstupním logu musí být ve stejném pořadí, jako zpracované události.	OK
Test opakovaného volání `Pop`	Žádná událost nesmí být metodou vrácena dvakrát.	OK
Test počtu událostí vrácených voláním `Pop`	Metoda musí vrátit všechny přijaté události, které nevrátila při předchozím volání.	OK

Tabulka A.12: Testy Log filtru.

Testy entity `LogReplayFilter`		
Název testu	Očekávaný výsledek	Výsledek
Test `Information` události	Událost musí být odeslána do navázaného filtru a nesmí být nijak modifikována.	OK
Test `Warning` události	Událost musí být odeslána do navázaného filtru a nesmí být nijak modifikována.	OK
Test `Error` události	Událost musí být odeslána do navázaného filtru a nesmí být nijak modifikována.	OK
Test `Warm_Reset` události	Událost musí být odeslána do navázaného filtru a nesmí být nijak modifikována.	OK
Test `Shut_Down` události	Událost musí být odeslána do navázaného filtru a nesmí být nijak modifikována. Filtr poté nesmí přijímat další události.	CHYBA
Test konfigurace prázdnou cestou	Konfigurace musí selhat.	CHYBA
Test konfigurace validní cestou	Konfigurace musí projít.	OK
Test konfigurace nevalidní cestou	Konfigurace musí selhat.	CHYBA
Test generace výstupního souboru	Po úspěšné konfiguraci musí v cílové cestě vzniknout výstupní soubor.	OK
Test počtu záznamů v logu	Počet záznamů v logu musí odpovídat počtu zpracovaných událostí.	OK
Test záznamu události	Každá zpracovaná událost musí mít v logu odpovídající záznam.	OK
Test pořadí záznamů událostí	Záznamy ve výstupním logu musí být ve stejném pořadí, jako zpracované události.	OK
Test opakovaného volání `Pop`	Žádná událost nesmí být metodou vrácena dvakrát.	OK
Test počtu událostí vrácených voláním `Pop`	Metoda musí vrátit všechny přijaté události, které nevrátila při předchozím volání.	OK

Tabulka A.13: Testy filtru pro zpětné přehrávání logů.

Testy entity `DrawingFilter`		
Název testu	Očekávaný výsledek	Výsledek
Test `Information` události	Událost musí být odeslána do navázaného filtru a nesmí být nijak modifikována.	OK
Test `Warning` události	Událost musí být odeslána do navázaného filtru a nesmí být nijak modifikována.	OK
Test `Error` události	Událost musí být odeslána do navázaného filtru a nesmí být nijak modifikována.	OK
Test `Warm_Reset` události	Událost musí být odeslána do navázaného filtru a nesmí být nijak modifikována.	OK
Test `Shut_Down` události	Událost musí být odeslána do navázaného filtru a nesmí být nijak modifikována. Filtr poté nesmí přijímat další události.	CHYBA
Test prázdné konfigurace rozměrů plátna	Konfigurace musí selhat.	CHYBA
Test prázdné konfigurace šířky plátna	Konfigurace musí selhat.	CHYBA
Test prázdné konfigurace výšky plátna	Konfigurace musí selhat.	CHYBA
Test konfigurace rozměrů plátna zápornými hodnotami	Konfigurace musí selhat	CHYBA
Test validní konfigurace rozměrů plátna	Konfigurace musí projít	OK
Test konfigurace nevalidní cestou ke grafu	Konfigurace musí selhat	CHYBA
Test konfigurace nevalidní cestou k dennímu grafu	Konfigurace musí selhat	CHYBA
Test konfigurace nevalidní cestou ke grafu AGP	Konfigurace musí selhat	CHYBA
Test konfigurace nevalidní cestou ke grafu Parkes	Konfigurace musí selhat	CHYBA
Test konfigurace nevalidní cestou ke grafu Clark	Konfigurace musí selhat	CHYBA

Test konfigurace nevalidní cestou ke grafu EDCF	Konfigurace musí selhat	CHYBA
Test konfigurace validní cestou ke grafu	Konfigurace musí projít	OK
Test konfigurace identickými nevalidními cestami k více grafům	Konfigurace musí selhat	CHYBA
Test konfigurace identickými validními cestami k více grafům	Konfigurace musí selhat	CHYBA
Test konfigurace různými validními cestami k více grafům	Konfigurace musí projít	OK
Test vytvoření graf	Po konfiguraci a odeslání alespoň jedné události musí být v cílové cestě vytvořen soubor grafu	CHYBA
Test přečtení SVG metodou `Draw`	Po zpracování události musí metoda vrátit řetězec SVG	OK
Test kontroly nových dat	Po zpracování události musí metoda `New_Data_Available` vracet hodnotu `true`. Při vícenásobném volání musí vracet `false`.	OK

Tabulka A.14: Testy filtru pro vykreslování.

Testy entity `MappingFilter`		
Název testu	Očekávaný výsledek	Výsledek
Test `Information` události	Událost musí být odeslána do navázaného filtru a nesmí být nijak modifikována.	OK
Test `Warning` události	Událost musí být odeslána do navázaného filtru a nesmí být nijak modifikována.	OK
Test `Error` události	Událost musí být odeslána do navázaného filtru a nesmí být nijak modifikována.	OK
Test `Warm_Reset` události	Událost musí být odeslána do navázaného filtru a nesmí být nijak modifikována.	OK
Test `Shut_Down` události	Událost musí být odeslána do navázaného filtru a nesmí být nijak modifikována. Filtr poté nesmí přijímat další události.	CHYBA
Test konfigurace prázdným zdrojovým `GUID` signálu	Konfigurace musí selhat	CHYBA
Test konfigurace prázdným cílovým `GUID` signálu	Konfigurace musí selhat	OK
Test konfigurace nevalidním zdrojovým `GUID` signálu	Konfigurace musí selhat	CHYBA
Test konfigurace nevalidním cílovým `GUID` signálu	Konfigurace musí selhat	CHYBA
Test konfigurace validními `GUID` signálů	Konfigurace musí projít	OK
Test konfigurace zdrojového signálu *All* signálem	Konfigurace musí projít	OK
Test konfigurace cílového signálu *All* signálem	Konfigurace musí selhat	CHYBA
Test konfigurace zdrojového signálu *Null* signálem	Konfigurace musí selhat	CHYBA
Test konfigurace cílového signálu *Null* signálem	Konfigurace musí projít	OK
Test mapování `Level` události	`Signal_Id` události s nakonfigurovaným `GUID` zdrojového signálu bude změněno na `GUID` cílového signálu.	OK

Test mapování `Information` události	`Signal_Id` události s nakonfigurovaným `GUID` zdrojového signálu bude změněno na `GUID` cílového signálu.	OK
Test mapování `Parameters` události	`Signal_Id` události s nakonfigurovaným `GUID` zdrojového signálu bude změněno na `GUID` cílového signálu.	OK
Test mapování události s nenakonfigurovaným `GUID` zdrojového signálu	`Signal_Id` události nebude změněno.	OK
Test mapování události *Null* signálem	Událost s nakonfigurovaným `GUID` zdrojového signálu neprojde do výstupního filtru	OK
Test mapování neplatné události *Null* signálem	Událost se `Signal_Id` rozdílným od zdrojového signálu projde do výstupního filtru	OK
Test mapování *All* signálem	`Signal_Id` všech zpracovaných událostí bude změněno na cílový signál	CHYBA

Tabulka A.15: Testy mapovacího filtru.

Testy entity `MaskingFilter`		
Název testu	Očekávaný výsledek	Výsledek
Test `Information` události	Událost musí být odeslána do navázaného filtru a nesmí být nijak modifikována.	OK
Test `Warning` události	Událost musí být odeslána do navázaného filtru a nesmí být nijak modifikována.	OK
Test `Error` události	Událost musí být odeslána do navázaného filtru a nesmí být nijak modifikována.	OK
Test `Warm_Reset` události	Událost musí být odeslána do navázaného filtru a nesmí být nijak modifikována.	OK
Test `Shut_Down` události	Událost musí být odeslána do navázaného filtru a nesmí být nijak modifikována. Filtr poté nesmí přijímat další události.	CHYBA

Test prázdné konfigurace	Konfigurace musí selhat	OK
Test konfigurace nevalidním `GUID` signálu	Konfigurace musí selhat	CHYBA
Test validní konfigurace	Konfigurace musí uspět	OK
Test konfigurace dvoubitovou maskou	Konfigurace musí selhat	CHYBA
Test konfigurace šestnáctibitovou maskou	Konfigurace musí uspět	OK
Test konfigurace dvanáctibitovou maskou	Konfigurace musí selhat	CHYBA
Test nevalidní konfigurace	Konfigurace musí selhat	CHYBA
Test maskování podle bitových masek	Pokud je bit nulový, `Level` událost bude maskována na `Masked_Level`, jinak nebude maskována.	CHYBA
Test maskování `Information` události	Událost nesmí být nikdy maskována	CHYBA

Tabulka A.16: Testy maskovacího filtru.

Testy entity `SignalGeneratorFilter`		
Název testu	Očekávaný výsledek	Výsledek
Test `Information` události	Událost musí být odeslána do navázaného filtru a nesmí být nijak modifikována.	OK
Test `Warning` události	Událost musí být odeslána do navázaného filtru a nesmí být nijak modifikována.	OK
Test `Error` události	Událost musí být odeslána do navázaného filtru a nesmí být nijak modifikována.	OK
Test `Warm_Reset` události	Událost musí být odeslána do navázaného filtru a nesmí být nijak modifikována.	OK
Test `Shut_Down` události	Událost musí být odeslána do navázaného filtru a nesmí být nijak modifikována. Filtr poté nesmí přijímat další události.	CHYBA

Test prázdné konfigurace	Konfigurace musí selhat	OK
Test konfigurace nevalidním modelem	Konfigurace musí selhat	CHYBA
Test konfigurace Bergmanovým modelem	Konfigurace musí uspět	OK
Test konfigurace nevalidním signálem k synchronizaci	Konfigurace musí selhat	CHYBA
Test konfigurace validním signálem k synchronizaci	Konfigurace musí uspět	OK
Test konfigurace bez synchronizace na signál	Konfigurace musí uspět	CHYBA
Test konfigurace *All* signálem k synchronizaci	Konfigurace musí uspět	OK
Test konfigurace *Null* signálem k synchronizace	Konfigurace musí uspět.	OK
Test konfigurace záporným krokováním	Konfigurace musí selhat	CHYBA
Test konfigurace kladným krokováním	Konfigurace musí uspět	OK
Test konfigurace záporným maximálním časem krokování	Konfigurace musí selhat.	CHYBA
Test konfigurace kladným maximálním časem krokování	Konfigurace musí uspět.	OK
Test asynchronního režimu	Model bude krokován do maximálního času.	CHYBA
Test konfigurace maximálním časem krokování nižším než krokování	Konfigurace musí selhat.	CHYBA
Test `Time_Segmant_Start` události	V asynchronním režimu budou odeslány události, reprezentující momentální stav modelu pro daný segment.	OK
Test vícenásobného startu jednoho segmentu	Kromě prvního odeslání události `Time_Segment_Start` se stejným `Segment_Id` musí všechna další skončit s chybou.	CHYBA

Test krokování o méně než je nakonfigurované krokování	Model nebude krokován a nebudou odeslány události reprezentující čas v dalším kroku.	OK
Test krokování o nakonfigurovanou hodnotu	Model bude krokován jednou a budou odeslány události reprezentující čas v po tomto kroku.	OK
Test krokování o násobek nakonfigurované hodnoty	Model bude krokován o násobek nakonfigurované hodnoty a budou odeslány události reprezentující stavy ve všech krocích.	OK
Test výchozích parametrů jako události	Pokud je parametr nastaven, jako první bude odeslána `Parameters` událost s předanými parametry v konfiguraci.	OK

Tabulka A.17: Testy generátoru signálu.

B Struktura přiloženého CD-ROM

- `CMakeLists.txt` - soubor, potřebný k přeložení projektu pomocí nástroje *CMake*
- `debian_64`
  - `x86_64` - složka s binárními soubory pro 64-bit platformu *GNU/Linux*, obsahuje i referenční binární soubory systému *SmartCGMS*
- `doc` - složka s programátorskou dokumentací ve formátu HTML, vygenerovanou pomocí nástroje *Doxygen*
- `docker` - složka se soubory, potřebnými k nasazení aplikace v *Docker* kontejneru
- `scenarios` - složka se scénáři pro regresní testování
- `smartcgms`
  - `src`
    - `common` - složka se sadou vývojových nástrojů systému *SmartCGMS*, nutná k přeložení vyvíjeného nástroje v odpovídající cestě
- `src` - složka se zdrojovými soubory aplikace
  - `app`
  - `mappers`
  - `testers`
  - `utils`
- `testFiles` - složka s pomocnými soubory k jednotkovým testům
- `Testovani_SW_komponent_SmartCGMS.pdf` - text bakalářské práce
- `windows_64`
  - `x86_64` - složka s binárními soubory pro 64-bit platformu *Windows*, neobsahuje referenční binární soubory systému *SmartCGMS*

C Uživatelská příručka

V rámci uživatelské příručky bude popsáno spuštění kontejneru, ve kterém lze spustit nasazenou verzi vyvinutého nástroje.

C.1 Spuštění kontejneru

Ke spuštění je třeba mít nainstalovaný *Docker*. Kontejner lze snadno spustit pomocí `docker run -name jenkins-smarttester -u root -d -p 8080:8080 -v jenkins-data:/var/jenkins_home -v /var/run/docker.sock:/var/run/docker.sock markovda/jenkins-smarttester:latest`.

Vytvořit *volume*, do kterého se uloží data z `/var/jenkins_home` je žádoucí, aby se zachovala perzistentní data při více spuštěních. Při dalších spuštěních pak lze spouštět již pomocí `docker start jenkins-smarttester`.

C.2 Vytvoření projektu

Pro vytvoření nového projektu je třeba otevřít `localhost:8080` v prohlížeči. Zde běží webová aplikace Jenkins. Nejdříve je třeba se přihlásit pomocí jednorázového admin hesla. Toto heslo je uloženo ve složce `/var/jenkins_home/secrets/`. Přesná cesta je uvedena ve webovém rozhraní. Pak stačí pouze postupovat podle pokynů, vytvořit admin účet a Jenkins je funkční.

V menu Jenkinse je třeba vytvořit nový projekt. Typ projektu je pouze ten nejzákladnější. Chceme pouze spouštět naše vlastní skripty. V nastavení projektu je třeba nastavit spuštění sestavovacího skriptu v části **Build**. Sestavovací skript je třeba spustit pomocí: `bash -x /var/jenkins_home/install.sh master`. Tento skript pouze stáhne a sestaví projekt *SmartCGMS*.

Pro jeho otestování je třeba přidat další krok pomocí tlačítka *Add build step* a výběru *Execute shell*. Zde je třeba spustit skript pro spuštění testů pomocí: `bash -x /var/jenskins_home/run-tests.sh`. Skript stáhne nejnovější verzi nástroje *SmartTester*, pokud je třeba a starší smaže. Zkopíruje přeložené binární soubory *SmartCGMS*, potřebné pro spuštění testů, ke spustitelnému souboru `SmartTester` a spustí všechny jednotkové testy. Momentálně jsou navíc spouštěny regresní testy scénářů 1 - 3. Spouštěné

testy lze upravovat v souboru `run-tests.sh`.