Vysoká škola báňská – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Framework pro automatizaci strojů v polovodičovém průmyslu

Framework for Machine Automation in Semiconductor Industry

2020 **Bc. Aleš Křenek**

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Aleš Křenek**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Framework pro automatizaci strojů v polovodičovém průmyslu
Framework for Machine Automation in Semiconductor Industry

Jazyk vypracování: čeština

Zásady pro vypracování:

Vytvořit framework v jazyce Java, který se stane zakladem pro nové řešení automatizace strojů v polovodičovém průmyslu. Jeho cílem je poskytnou obecně použitelné řešení pro definici a řízení business flow a interakce se strojem. Díky čemuž bude možné redukovat aktuální velké množství duplicitní funkcionality a snížit tím čas potřebný na udržbu aktuální automatizace a využít jej na vývoj nově požadované funkcionality.

1. Tvorba jednotného rozhraní pro komunikaci s klientem.
2. Tvorba jádra jež umožní spouštění příkazů od klienta a držení kontextových informací.
3. Tvorba jednotného rozhraní pro komunikaci se strojem (TCP/IP, RS232, Soubory, ...).
4. Tvorba obecné definice komunikačního protokolu a jednotného přístupu k provádění příkazů a kontrola jejich vstupů a výstupů.
5. Tvorba funkcionality pro zpracování a validaci vstupních a výstupních dat.
6. Sjednocení práce s proměnnými skze celý framework, včetně vstupních a výstupních dat.
7. Vytvoření jednotné práce s výjimkami a jejich propagace klientovi.
8. Synchronizace mezi aplikacemi.
9. Praktické využití při nahrazení aktuální automatizace vyrobní linky.

Seznam doporučené odborné literatury:

Podle požadavků vedoucího diplomové práce.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Stanislav Martínek**

Konzultant diplomové práce: Ing. David Ježek, Ph.D.

Datum zadání: 01.09.2018

Datum odevzdání: 30.04.2019

doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlášení studenta

Prohlašuji, že jsem tuto bakalářskou/diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě dne: *15. května 2020*

.....
podpis studenta

Prohlášení zástupce spolupracující právnické nebo fyzické osoby

„Souhlasím se zveřejněním této bakalářské/diplomové práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v bakalářských/magisterských programech VŠB-TU Ostrava."

Dne: *13. května 2020*

.....
podpis zástupce

Poděkování

Rád bych poděkoval Ing. Davidu Ježkovi, Ph.D. za odbornou pomoc a konzultaci při vytváření této diplomové práce.

Rád bych poděkoval Ing. Stanislavu Martínkovi za příležitost pracovat s jeho pomocí na této diplomové práci.

Rád bych poděkoval firmě ON Semiconductor za možnost dokončit při zaměstnání vysokou školu a pracovat na diplomové práci z prostředí ve kterém jsem pracoval.

Abstrakt

Cílem diplomové práce bylo na základně existujících aplikací pro automatizaci strojů v polovodičovém průmyslu vytvořit analýzu současného stavu. Dle ní poté identifikovat části s opakující se funkcionalitou, nebo části náchylné k programátorským a uživatelským chybám. Poté pro tyto části vytvořit pře použitelnou implementaci, která by jako celek mohla sloužit jako framework pro automatizaci těchto strojů a díky tomu dosáhnou rychlejšího vývoje a spolehlivějších aplikací.

Práce bude realizovaná v interpretovaném programovacím jazyce Java, díky své přenositelnosti a popularitě. Během vývoje bude kladen důraz na výslednou spolehlivost řešení a použití ověřených postupů.

Klíčová slova

polovodičový průmysl; automatizace výroby; framework; tester; handler; probe; final test; návrhové vzory

Abstract

Goal of this thesis was to create analysis of current applications used for factory automation. Find parts with duplicate functionality, or parts prone to errors caused by developer or application user. Create new implementation for these cases, which will be more generic and as such usable as framework for factory automation. This approach should lead to shorter development cycle and more reliable automation software.

Thesis is going to be implemented in Java programing language, thanks to its portability and popularity. During development primary goal is going to be reliability of final product as well as usage of already proven procedures.

Key words

semiconductor industry; factory automation; framework; tester; handler; probe; final test; design patterns

Seznam použitých zkratek

Zkratka	Význam
IoC	Inversion of Control

Seznam použitých termínů

Termín	Význam termínu
Tester	Stroj v polovodičové výrobě, jehož cílem je změřit elektrické vlastnosti polovodičových součástek
Handler	Stroj v polovodičové výrobě, jehož cílem je manipulace s polovodičovými součástkami během měření
Laserový popisovač	Stroj v polovodičové výrobě, jehož cílem je označení výrobku čitelným popiskem
Optická inspekce	Stroj v polovodičové výrobě, jehož cílem je kontrola označení výrobku provedeného Laserovým popisovačem
Testovací Stanice	Sestava strojů ve výrobě sloužící pro provedení kroku výroby určeného pro otestování polovodičových součástek
Lot	Základní jednotka ve výrobě identifikující skupinu součástek a kroky výrobního procesu, kterými mají projít
Sublot	Část Lotu, která byla oddělena z procesních důvodů, nebo kvůli nemožnosti výroby pracovat s celým Lotem současně
Wafer	Křemíková deska, na které jsou vyráběny polovodičové součástky
Probe	Fáze výroby kdy jsou součástky na Waferu
Final Test	Fáze výroby kdy jsou součástky již zapouzdřené do samostatných pouzder

Obsah

Úvod

Tato diplomová práce vznikla jako snaha zjednodušit tvorbu aplikací pro automatizaci polovodičové výroby a vychází ze zkušeností, jež jsem nabyl během praxe a z teoretických i praktických příkladu řešení obdobných problémů ve světě. Přestože při návrhu a realizaci projektu byl kladen důraz na rozšiřitelnost a pře použitelnost, byl projekt cílený tak aby splnil potřeby vývoje v rámci specifické firmy. Zde se jedná hlavně o procesní a hardwarové omezení, či specifika.

Diplomová práce vznikla pro firmu ON Semiconductor® která je nadnárodní korporací zabývající se návrhem, vývojem a výrobou polovodičových součástek na zakázku. Díky tomu že firma vyrábí součástky na zakázku je její portfolio velice rozmanité a zahrnuje i výrobky určené pro odvětví kde chybný produkt může ohrozit životy lidí, jakými jsou automobilový průmysl, lékařství a armáda. Zvláště tyto segmenty výroby jsou průkopníky nových technologií a výrobních standardů, které zákazníci od firmy začínají žádat ve stále větší míře. To je také jeden se zásadních důvodů pro vznik této diplomové práce. Jelikož nutnost uvedení těchto změn do výrobního procesu na stále více strojů klade neustálý tlak na týmy jež jsou za vývoj a uvedení těchto změn do výroby odpovědné.

Tým, jehož jsem ve firmě ON Semiconductor® součástí má na starosti podporu a vývoj automatizace pro výrobní stroje. Mou specializací jsou měřící stroje, které dále budu nazývány Tester. Jejich účelem jak název napovídá je měření elektrických vlastností součástek, pro potřebu odstranění těch součástek u kterých došlo během procesu výroby k poškození, které by vedlo k nefunkčnosti výsledného produktu. Framework který v rámci této práce vznikl je na nich aktivně používán a dále pro ně vyvíjen. Samotný Tester je součástí Stanice která kromě jednoho nebo více Testerů obsahuje i další součásti. Nejdůležitější z nich je Handler jehož účelem je manipulace se součástkami. Dále pak Stanice může obsahovat například zařízení pro Optickou Inspekci, nebo Laserový popisovač a jiné.

Z pohledu výrobního procesu pak rozlišujeme dvě fáze výroby ve kterých se Testery používají. Prvním z nich je fáze výroby ve které se součástky nachází na křemíkové desce nazývané Wafer a jsou připraveny na rozřezání a zapouzdření. Tuto fázi výroby obecně nazýváme Probe. Druhou fází výroby ve které se používají Testery je výsledná kontrola před tím než výrobek opustí továrnu. v této fázi je již výrobek zapouzdřen v plastovém pouzdru a je nazývána Final Test.

Diplomová práce začala jako snaha postupně sjednotit existujících aplikace jež měly na starost automatizaci Testerů do podoby, která by byla udržitelná s menším týmem lidí a současně dovolila existující aplikace rozšiřovat případně v nich plošně opravovat chyby. Udržitelnost existujících řešení totiž představovala jednu z největších výzev, protože původně vznikaly nezávisle na sobě v různých týmech, které se staraly o jejich vývoj. Tento model se ukázal jako komplikace před několika lety kdy se firma rozhodla o plošném zvýšení kvality automatizace výroby. s tímto rozhodnutím přišla taky potřeba sjednotit existující řešení a vývoj nových do jednoho týmu. To znamenalo, že na vývoj a údržbu zůstalo méně lidí, než kolik jich

bylo původně. To znamenalo, že bylo třeba najít způsob jak zvýšit efektivitu práce vývojářů a tím se stal mnou vytvořený framework.

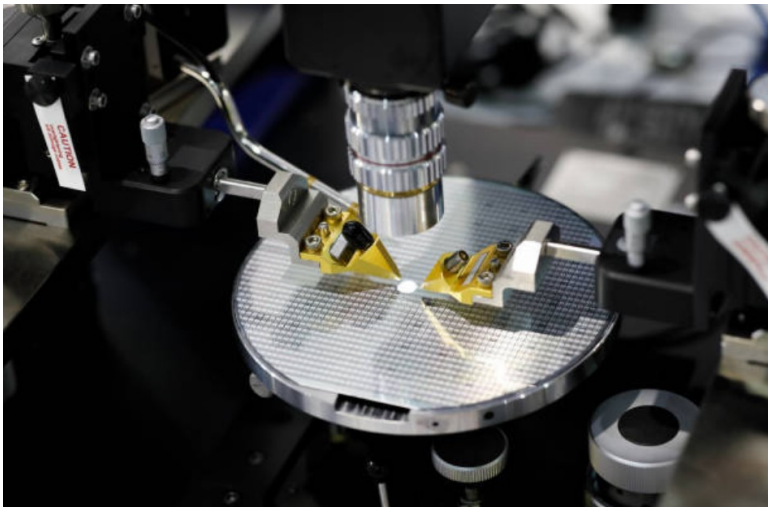
Framework samotný vznikal na nákladně funkčních zdrojových kódů existujících aplikací, které jsem nevytvořil já, ale pouze jsem je převzal a postupně přepisoval do nově vznikajícího frameworku. Aktuálně framework využívají desítky různých typů Testerů, kdy každý z nich je v rámci firmy zastoupen obvykle v desítkách.

1 Představení výrobního procesu

V této kapitole si představíme pro projekt relevantní části procesu, kterými polovodičová součástka prochází v továrně, aby čtenář lépe pochopil cíle a rozhodnutí, které jsem v rámci projektu dělal. Jsou jimi Probe během kterého jsou součástky na křemíkové desce a Final Test kdy jsou součástky již zapouzdřené a připravené na odeslání k zákazníkovi.

1.1 Probe

Prvním krokem výroby, pro který jsem framework vytvářel je znám jako Probe, do této fáze výroby přichází součástky na křemíkové desce zvané Wafer. Součástky samotné jsou v této chvíli připravené na otestování, zdali byly všechny předchozí kroky výroby provedeny korektně a součástka je funkční.



Obrázek 1.1: *Probe - Měření součástky na Waferu*

Počet součástek na jednom Waferu je závislý na jejich složitosti a samotném průměru Waferu, který může být 6/8/12 palců (přibližně 15/20/30cm). Obecně ale můžeme říci, že se bavíme o rozsahu od jednotek kusů po statisíce, dle složitosti součástky. v rámci výroby se Wafery spojují do sad které se nazývají Lot.

Lot je základním procesní jednotkou výroby, reprezentuje součástky procházející celým výrobním procesem současně. Slouží pro identifikaci výrobků během celého procesu výroby a kromě fyzických součástek jsou mu přiřazené také informace o tom, jaké kroky výroby mají být během výrobního procesu provedeny a v kterém kroku se Lot právě nahází. Z pohledu tohoto projektu nás bude zajímat primárně typ součástky zvaný Device a testovací program, kterým je kontrolována funkčnost jednotlivých součástek.

Samotné testování provádí Operátor výroby na Stanici která mimo samotného Testeru obsahuje i Handler který slouží k manipulaci s Wafery, Inker sloužící k označení špatných součástek a řídící Software, který koordinuje souhru těchto a případně i dalších strojů jež jsou

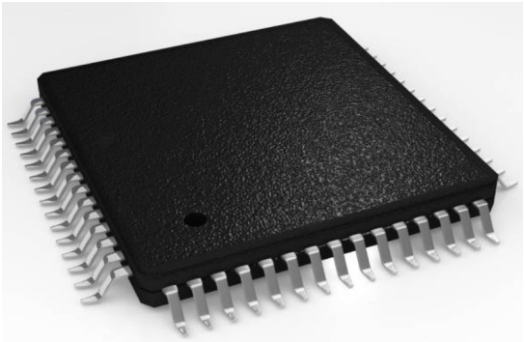
součástí Stanice. Tento Software bývá umístěn na svém vlastním počítači a se zařízeními komunikuje skrze definovaný protokol. Na straně Testeru se jedná o Tester Driver, což je aplikace, která je nainstalovaná na počítači patřící Testeru. Jejím primárním účelem je vytvořit obecné komunikační rozhraní mezi řídícím Software stanice a Testerem. Díky tomu může řídící Software být obecný a použitelný na libovolné Stanici. Oproti tomu Tester Driver je unikátní pro každý typ Testeru a je s ním úzce spjat.

Výsledkem tohoto kroku výroby jsou Wafery které mají označené špatné součástky. Ty budou v následujícím kroku výroby, ve kterém je Wafer rozřezán na jednotlivé součástky odstraněny. Zatímco ty které prošly testováním, budou zapouzdřeny.

OBR pouzdra

1.2 Final Test

Druhým krokem výroby, pro který byl framework vytvořen se nazývá Final Test a v této fázi výroby jsou již součástky zapouzdřeny do pouzder, obvykle černé barvy, ve kterých je možné s nimi manipulovat.



Obrázek 1.2: *Final Test - Zapouzdřená součástka*

Součástky příchozí do této fáze výroby, jsou stále součástí stejného Lotu. v porovnání s fází Probe zde nehrají Wafery již žádnou roli, jelikož z nich byly součástky vyřezány v předešlém kroku výroby. Namísto nich se nyní součástky mohou dělit na Subloty.

Sublot představuje část z celkového množství součástek jež tvoří Lot. Dělení na Subloty obvykle vzniká z následujících důvodů. Prvním z nich může být zrychlení výrobního procesu. Různé Subloty pak mohou být testovány na různých Stanicích, čímž se ušetří čas. Druhým důvodem může být limitace samotné Stanice. Ten může být buď Softwarový a představovat ho může například omezení počtu testovaných součástek kvůli limitům vnitřních čítačů, nebo Hardwarový kde nejjednodušším příkladem může být limitní velikost vstupního podavače součástek. Třetím důvodem pak mohou být čistě procesní rozhodnutí, kdy se například určité kontrolní testy provádí pouze pro jeden ze Sublotů daného Lotu.

Final Test Stanice se oproti Probe Stanicím liší nejenom svým vstupem ale i primárním cílem. Cílem obou Stanic je sice vyřadit špatné součástky, ale ve fázi Probe je měření složitější a Wafer je mnohem zranitelnější než zapouzdřená součástka a proto je snaha minimalizovat kontakt s ním i za cenu že skrze kontrolu projdou i vadné součástky. Naproti tomu ve fázi Final Test je zapotřebí odstranit všechny chybné nebo potencionálně chybné součástky. Proto je zde testování mnohem důslednější. To klade důraz i na Stanici samotnou, nachází se v ní obvykle více než jeden Tester, protože obvykle neexistuje Tester který by dokázal provést všechny potřebné testy sám. Proto se obvykle stanice doplňuje o specializované Testery pro měření elektrického napětí, proudu, nebo odporu. Případně se doplňuje o Tester který měří součástky ve specifických podmínkách. Těmi může být například vysoká či nízká teplota. Součástky, které projdou skrze tyto testy, bývají obvykle změřeny ještě jednou na Testeru kontroly kvality, jež slouží jako poslední kontrola před tím než součástky zamíří k zákazníkovi a bývá kvůli úspoře času a prostoru součástí Stanice.

1.3 Testovací Stanice

Testovací Stanice je sestavení zařízení plnících specifické účely, jejichž souhrnným cílem je identifikace špatných součástek během procesu výroby. Nejběžnější z nich jsou Testery a Handlery, dále pak existují zařízení pro popisování, optickou inspekci, pece, chladící komory a mimo jiných také například Signalizační maják.

Vzhledem k tomu že zařízení použité v rámci stanice mohou být diametrálně odlišné v závislosti na typu zapouzdření a požadavcích na součástku je automatizace Stanice rozdělena na centrální řídící jednotku a řídící prvky jednotlivých strojů.

Každé zařízení jež je součástí stanice má vlastní aplikaci řídící automatizaci tohoto zařízení. Základními vlastnostmi těchto automatizačních řešení jsou snaha poskytnout jednotný protokol pro standardizované vzdálené řízení stroje a nezávislost na okolí. Samotné aplikaci musí být poskytnuty všechny potřebné informace skrze řídící rozhraní. Díky tomu jsou tyto automatizační řešení lehce přenositelné a nejsou závislé na zdrojích dat, jež se mohou lišit mezi továrnami, nebo i jednotlivými stanicemi.

Společně s nimi existuje pro každou stanici aplikace jež celou stanici řídí. Ta zajišťuje komunikaci se systémem řízení výroby, který poskytuje informace o zvolených testech, které mají být na součástkách provedeny. Dále předpřipravuje informace potřebné pro spuštění automatizace jednotlivých strojů a celou automatizaci řídí. Celý proces končí kontrolou výstupních dat z měření, zaznamenání výsledků do systému řízení výroby a připravení stanice na další měření. Takováto aplikace bývá obvykle jedna pro daný krok výrobního procesu. Její pře použitelnost na různé Stanice je pak dosažena modularitou jejího řešení a jednotným komunikačním rozhraním s daným typem zařízení.

Jednotnost komunikačního rozhraní může být zajištěna buď přímo jeho výrobcem, to je v praxi ale pouze ideální stav a platí pro jednodušší zařízení jakými jsou například Laserové popisovače, nebo Optická inspekce. Pro složitější zařízení jakými jsou Testery a Handlery výrobci neposkytují takové rozhraní a pouze dodávají své vlastní řešení, které ale není možné použít přímo. v praxi je třeba vytvořit aplikaci která za pomoci nástrojů dodaných výrobcem vytvoří na venek jednotné rozhraní, se kterým může pracovat řídící systém Stanice.

Tato projekt vznikl se zaměřením na jeden takový typ zařízení. Tímto typem je Tester. Pod tímto názvem se skrývá skupina strojů jež dokáží změřit elektrické vlastnosti součástky v předdefinovaných situacích. Takováto sada měření se obecně nazývá Testovací Program a způsob realizace je dán výrobcem Testeru. Může být realizována jak konfiguračním souborem tak i například jako DLL. Na základě výsledku pak může stanice součástku zahodit, nebo poslat na následující krok výrobního procesu. Tyto stroje jsou z pohledu kvality výroby na prvním místě, protože chyby způsobené výběrem špatného Testovacího Programu, či případná ztráta výsledů měření, může způsobit že k zákazníkovi odejdou špatné součástky, případně nebude možné dohledat kde se stala chyba pokud zákazník vrátí součástky jako vadné.

Proto je kvalita automatizace Testerů kritickou částí výrobního procesu a proto i tento projekt vychází v základech z jejich požadavků. Tyto požadavky budou ukázány v kapitole popisující reálný příklad použití.

Obecně ale můžeme říci, že základními cíli automatizace Testerů je kompletní odstranění interakce člověka s Testerem, zajištění korektního nastavení Testovacího programu, zajištění že během testování nedojde k jeho výměně (útok/snaha obejít proces), sběr a kontrola výstupních dat poskytnutých Testerem a jejich odeslání do centrálního úložiště a příprava Testeru pro další měření.

2 Představení frameworku

V této kapitole bude postupně představeno řešení které existovalo před zavedením frameworku a důvody proč bylo rozhodnuto pro sjednocení přístupu k automatizaci za pomoci mnou vyvinutého frameworku. Následně představím omezení se kterými se musel framework překonat a jeho jednotlivé vývojové kroky kterými během vývoje prošel, jeho rozsah a možnosti postupně rostly s tím jak se zvětšoval rozsah jeho použití.

2.1 Automatizace před příchodem frameworku

Pro pochopení situace před frameworkem je třeba říci že firma ON Semiconductor roste primárně skrze akvizice menších společností se kterými přichází i jejich vlastní pojetí automatizace a její potřeby.

Proto byla automatizace Testerů před příchodem frameworku řešena z velké části dle možností dané továrny. Některé z nich nedisponovaly automatizací vůbec a spoléhaly se pouze na software dodaný společně s Testerem, jiné využily vlastních lidských zdrojů a automatizovali alespoň část procesu testování. Některé měly možnost použít zdrojů globálního IT oddělení, jehož jsem se stal součástí i já, dostaly řešení automatizace, které sice vycházelo z jednotných idejí. Jednalo se ale převážně o unikátní řešení jež mezi sebou sdílely když už tak pouze malé fragmenty zdrojových kódů. Samotná funkcionalita v rámci jednotlivých řešení se pak implementovala pokaždé znovu a to navíc v různých úrovních kvality, dle schopností programátorů a množství času jež na práci dostal.

Nejnovější řešení měly několik zásadních výhod oproti původním řešení které se staly základem pro framework samotný. Tím nejzásadnějším byl definovaný protokol pro komunikaci s řídícím softwarem Stanice, díky tomu byly nejenom řešení zaměnitelná mezi sebou což přineslo možnost větší variability výrobě, ale také bylo možno odstranit uživatelské rozhraní automatizace. Uživatel tak mohl celý Tester ovládat z řídícího softwaru Stanice. Druhou výhodou bylo pře použití implementace příjmu a odesílání příkazů v rámci tohoto rozhraní.

Naopak nedostatkem těchto řešení byl jejich rozsah a individuální implementace. Původní cíl automatizace bylo pouze načtení Test programu a odstranění jeho kopie z Testeru na konci měření.

2.2 Úvodní cíl a začátek vývoje

Vývoj frameworku začal nedlouho po té co se firma rozhodla plošně zlepšit kvalitu automatizace, bylo rozhodnuto že základem se stane již existující rozhraní pro automatizaci, společně s již existující implementací zpracování příchozích a odchozích dat.

Problémem se velice záhy stal rozsah změn a chyby v existujících řešeních, které se uživatelé v průběhu let naučili obcházet a v rámci projektu se snažili využít zvýšený zájem pro získání času na opravu. Při této práci se ukázal první problém spojený s individuálními

implementacemi a to že nejenom bylo některé chyby testovat na s každou verzí znovu, současně ale také nebylo vždy možné aplikovat stejný postup pro její opravu. Velice rychle se díky tomu ukázalo že při zachování lidských zdrojů a termínů bude třeba začít funkcionalitu sjednocovat.

To co začalo, jako pouhé sjednocení částí zdrojových kódů se díky mě změnilo v plnohodnotný framework který obsahuje komponenty pro tvorbu automatizace pro libovolný Tester.

Pro tvoru automatizace bylo rozhodnuto pro interpretovaný jazyk, který nabízí nezávislost na verzi a typu operačního systému a použitých součástek počítače. Mezi možnými kandidáty byl zvolen jazyk Java, díky jeho rozšíření a znalosti programátorů. Pro sestavení byl zvolen Maven. Který byl již ve firmě zaveden.

Při tvorbě frameworku bylo stanoveno několik cílů. Nejdůležitějším z nich byla cíl nechat v samotné implementaci automatizace pro daný Tester, bude nazýván obecně Tester Driver, pouze logiku aplikace a funkcionalitu, kterou představují dodat z frameworku. Díky tomu budou jasně odděleny možné logické a funkcionální chyby. Kde logické chyby budou zcela jasně relevantní pouze pro daný Tester Driver a po jejich opravení bude jisté že se znovu neobjeví. Naopak chyby ve funkcionalitě díky tomu že jsou součástí frameworku bude stačit opravit jednou a do ostatních Tester Driverů se dostanou automaticky s aktualizací. Druhým kritériem při návrhu byla rozšiřitelnost, v rámci celého frameworku jsem se rozhodl dle doporučení z knihy Design patterns: elements of reusable object-oriented software (1) pro delegování a kompozitní skládání funkcionality, namísto alternativy kterou byla dědičnost. Druhým zdrojem praktických zkušeností a postupů byla kniha Code complete(2).

Toto rozhodnutí sice přineslo nutnost napsat více zdrojového kódu, ale současně umožnilo v případě potřeby vytvořit z existujících komponent nový kompozit pro funkcionalitu, kterou bylo potřeba rozšířit, při zachování čitelnosti zdrojového kódu a to i v případě kdy by podobné řešení dědičností bylo komplikované, či nepřehledné. Třetí zásadou bylo využít již existující funkcionality frameworku pro tvorbu nové funkcionality, to přineslo potřebu vyšší úroveň obecnosti a složitější zdrojový kód. Současně ale díky tomu framework obsahuje méně zdrojových kódů a nová funkcionalita se velice jednoduše zavádí i do existující funkcionality.

3 Komponenty frameworku

V této kapitole budou detailně představeny jednotlivé části frameworku. Probrán bude důvod jejich vzniku, vývoj kterým prošly a plánované rozšíření funkcionality. Jejich využití v rámci celku pak bude demonstrováno v následující kapitole.

následujících kapitolách budou popsány jednotlivé části realizovaného frameworku. Pořadí odpovídá interním závislostem vně frameworku. Samotný framework vznikal v několika iteracích a během doby než byl uznán za framework byl několikrát zásadně rozšířen a starší funkcionalita aktualizována aby zapadala do celkového konceptu. Součástí popisu jednotlivých komponent budou popsány i některé chybné rozhodnutí jež obvykle vedly k těmto změnám. Praktické ukázky kódů budou vždy ve verzi, jež je součástí této diplomové práce.

V rámci kapitol předpokládám že čtenář má základní podvědomí o programovacím jazyku Java. Jelikož jeho možnosti a limitace byly zásadními faktory při tvorbě frameworku a stejné podmínky nemusí v jiných jazycích existovat.

3.1 Parametry

Když jsem nastoupil do praxe zjistil jsem zanedlouho že používat Vstupně/Výstupní rozhraní s metodami mající pevně dané argumenty není nejlepší řešení. Zvlášť pokud se jedná například o webovou službu. Namísto nich je lepší použít zástupnou třídu/strukturu jež umožní předat libovolné množství parametrů beze změny rozhraní a tudíž bez případné potřeby měnit klientské aplikace.

Pro vytvořený framework je tímto prvkem třída Parameters. Na začátku vývoje vznikla jako jednoduché zapouzdření Hash Mapy ve které se vyhledávala textová hodnota podle jména proměnné. Řešení sice splnilo podstatu neměnného rozhraní, ale neposkytlo nic navíc. Uživatel byl tak stále nucen provádět kontroly existence proměnných, přetypování a realizovat přenos neatomických hodnot v rámci zdrojového kódu Tester Driveru. Všechny tři zmíněné funkcionální nedostatky byly běžně používané a existující zdrojové kódy byly náchylné na chyby, proto jsem se na ně zaměřil a funkcionalitu integrovat do frameworku.

Prvním krokem bylo přetypování, díky tomu že vstupními hodnotami byly textové řetězce, vznikala běžně potřeba převést je na jiný datový typ. Tyto konverze neposkytuje Java implicitně a bylo je tedy třeba dělat v aplikaci. v mnoha případech pak toto řešení bylo náchylné na chyby, protože obvykle předpokládalo, že hodnota parametru existuje a skutečně je v očekávaném datovém typu. Rozhodl jsem se tedy rozšířit funkcionalitu třídy Parameters o typovou konverzi.

Cílem bylo umožnit přetypovat mezi sebou libovolné datové typy, díky tomu se stane zdroj a cíl dat nezávislý na přenášeném datovém typu. To se ukázalo jako v průběhu práce jako zásadní vlastnost, která zjednodušila pře použitelnost částí frameworku.

Druhým obvyklým problémem bylo nastavení defaultních hodnot. Většina parametrů měla v rámci každého specifického Tester Driver uloženou defaultní hodnotu, která se měla

použít pokud nebyla explicitně poskytnuta. Mali aplikace takových hodnot pár, obvykle to není problém. v případně Tester Driverů na kterých pracuji se ale jedná o desítky parametrů a v takových případech si developer má tendenci práci ulehčovat. Navíc zdrojové kódy pak obsahují spoustu ne zcela přínosného kódu. Mým dalším krokem tedy byla snaha poskytnout defaultní hodnotu přímo skrze třídu Parameters pokud není definovaná, bez toho aby o tom věděl uživatel. Samotný zdroj výsledné hodnoty pak bude součástí logu Tester Driveru. Společně s tím jsem chtěl mít možnost mít více než jeden zdroj protože v průběhu běhu Tester Driveru se mohou vznikat a zanikat zdroje defaultních hodnot které jsou závislé na kontextu ve kterém se právě aplikace nachází. Tuto vlastnost jsem zahrnul mezi základní požadavky, dále tedy bude předpokládáno, že bude existovat více než jeden zdroj defaultních hodnot.

Pro řešení jsem zvažoval několik možností. První bylo držet defaultní hodnoty v dalších instancích třídy Parameters. Poté by se dalo získání hodnoty zjednodušit na dotazování v rámci seřazené množiny tříd Parameters. Tato možnost se jevila jako nejjednodušší, protože stačilo zapouzdřit množinu existujících instancí třídy Parameters do třídy se stejným rozhraním. Tato možnost ale vyžadovala, aby aplikace defaultní hodnoty předpřipravila, což by přineslo zvýšení komplexity mimo třídu Parameters. To byl zásadní problém, jelikož jsem chtěl zdrojový kód mimo framework udržet co nejjednodušší.

Proto jsem nakonec zvolil z pohledu realizace složitější variantu. To za využití reflexe v rámci jazyka Java. Reflexe umožnuje získat informace o struktuře libovolné třídy. Abych neohrozil aktuální funkcionalitu, vytvořil jsem novou Runtime Anotaci, jež umožnila popsat libovolnou getter metodu nebo přímo proměnnou v rámci třídy jako defaultní hodnotu pro určitý parametr. Z pohledu tříd které obsahovaly defaultní hodnoty šlo o jednoduché rozšíření jež neporušilo zpětnou kompatibilitu. Samotné třídě Parameters pak bylo možno poskytnout instanci libovolné třídy jako zdroj defaultních hodnot. Jakmile aplikace požádala o hodnotu a ta nebyla nalezena, prošla třída Parameters všechny poskytnuté zdroje defaultních hodnot a skrze výše zmíněnou anotaci pak dokázala získat defaultní hodnotu získat skrze reflexi. Během testování tohoto řešení jsem narazil na problém s přístupovými právy k proměnným. Aby toho řešení fungovalo bylo potřeba před přístupem k hodnotě změnit její typ na public a poté vrátit originální nastavení zpět. Tento postup mi ukázal jak silným nástrojem může reflexe být a proto jsem ji použil během vývoje na více místech.

Posledním krokem bylo přidání možnosti uložit do daného parametru více jak jednu hodnotu. Zde se plně využila výhoda rozhodnutí nepublikovat interní datový model třídy Parameters. Díky tomu bylo jednoduše možné změnit interní model při zachování původního rozhraní. v rámci třídy pak vznikla jediná obecná metoda pro získání dat, samotný interface třídy pak pouze mapuje své vstupy a výstupy na ty její.

Celkový výsledek slouží úspěšně a je plošně používán v celém frameworku. Bohužel se ukázalo, že rozhodnutí zachovat původní konfigurační třídy a pouze je rozšířit o přístup skrze anotace, výrazně komplikuje možnost měnit nastavení asynchronně vzhledem k probíhajícímu běhu procesu aplikace. Příkladem takové situace může být externí zdroj defaultních hodnot,

bude popsán v kapitole 3.8. Pokud se takto poskytnutá hodnota změní, nedokáže nyní framework jednoduše říci jestli se od posledního načtení změnila, kdy ke změně došlo.

Jedním z plánovaných rozšíření je tedy centralizace celé konfigurace a její správa včetně časových informací o tom jak dlouho jsou hodnoty platné. Tato funkcionalita je zásadní pokud chceme současně mít možnost za běhu změnit nastavení, například TCP/IP spojení a přitom neztratit již navázané spojení, nebo znovu načíst aktualizované části aplikace bez nutnosti jejího restartu.

3.2 Validace Parametrů

Společně s přístupem ke konfiguraci, který byl vyřešen v třídě Parameters popsané v kapitole 3.1, bylo potřeba sjednotit validace nastavení ty bylo do příchodu frameworku součástí Tester Driverů a stejně jako přístup a konverze hodnot se lišily mezi jednotlivými implementacemi, případně neexistovaly vůbec a kontrola probíhala za běhu aplikace, ve chvíli kdy byla hodnota potřeba. To byl zásadní problém ze dvou důvodů. Při běhu aplikace je u stroje pouze Operátor, který nemá znalosti a pravomoci problém odstranit. Druhým důvodem je nutnost restartovat aplikaci aby se některé změny projevily.

Bylo třeba tedy vytvořit validační systém který by mohl být použít plošně a současně byl zásah do již existujícího datového modelu co nejmenší. Díky tomuto požadavku jsem zvolil pro vyřešení validací kombinací anotací a reflexe. k tomuto přístupu jsem se rozhodl, protože aplikace do stávajících zdrojových kódů byla jednoduchá a práci s reflexí jsem měl naučenou z realizace Parametrů.

Jako základní validaci nevyžadující žádnou anotaci jsem zvolil kontrolu zda je proměnná nastavena na libovolnou hodnotu. Dále byly vytvořeny validace pro specifické účely. Mezi ty nejběžnější bych uvedl kontrola numerického rozsahu hodnoty, nebo kontrola že daná proměnná reprezentuje existující soubor, či složku na disku.

To pro potřeby čistě datových tříd stačilo. Ukázalo se však že řešení je velice praktické a našlo by využití i ve funkcionalitě, která nemá striktně oddělené nastavení ve specifické třídě. Bylo tedy třeba vytvořit podporu pro možnost ignorovat proměnou případně provést i rekurzivní kontrolu skrze několik tříd a doplnit kontrolu kolekcí. Vytvoření doplňkové anotace pro vypnutí kontroly bylo jednoduché. Práce s rekurzí a kolekcemi byla první výzva, na kterou jsem narazil, i když to bylo způsobené více abstrakcí reflexe, se kterou jsem pracoval ve větším rozsahu poprvé jak složitostí úkolu samotného.

3.3 Textové Vzory

Textové vzory jsou běžnou součástí mnoha aplikací. Poprvé jsem se s nimi setkal prakticky u definice formátu datových konverzí a logovacích zpráv. v případě vytvářeného frameworku bylo třeba vytvořit textové vzory ze začátku pouze pro tvorbu logovacích zpráv, kde se díky oddělené definice vzoru a dat mohlo jednoduše dojít ke změně jazyku zpráv

a zobrazit tak například chybovou hlášku v jazyce kterým hovoří operátor výroby a usnadnit mu tak pochopení případného problému.

Původní řešení před příchodem frameworku používalo jednoduché notace s proměnnými ve složených závorkách, který je běžně využíván v Java aplikacích jako je Maven, či Logback. Tato triviální implementace byla použitelná pro tvorbu záznamů v logu, či zobrazení uživateli. Pro náročnější použití se ale nehodila. Ve stejné době jsem ale hledal způsob jak zjednodušit definici a práci s komunikačními protokoly. Vzhledem k tomu že mou snahou bylo maximálně pře použít existující funkcionalitu, rozhodl jsem se rozšířit existující vzory. To se ukázalo jako moje první zásadní chyba, která vedla ke kompletnímu přepracování práce se vzory.

V úvodu jsem předpokládal že textová definice vzoru bude dostatečná, zde jsem vycházel z existujících aplikací využívající vzory pro svou potřebu. Úvodním vylepšením bylo přidání získání parametrů z hotového textu a vzoru. Tato operace byla potřeba pro základní implementaci deserializace zpráv jednoduchých protokolů. Pro realizaci jsem využil regulárních výrazů pro které Java poskytuje řadu let ověřenou funkcionalitu. Jako alternativu jsem zvažoval možnost překladu vzoru na nový jazyk a poté využit některý z existujících interpretů. Pro regulární výrazy rozhodla opatrnost, protože Tester Drivery jsou kritické aplikace v produkci a prakticky se od nich očekávají nulové výpadky. Kvůli tomu bylo rozhodnuto pro regulární výrazy jež podporuje Java nativně.

Jakmile byla první fáze hotova zaměřil jsem se na snahu rozšířit vzory o dodatečné formátování. Mým cílem bylo hlavně usnadnit práci se staršími komunikačními protokoly, které vyžadovaly striktní formátování co se zarovnání, velikosti znaků, výplně nevyužitých znaků, ořezání neviditelných znaků na koncích řetězce, ap. Dříve byla tato příprava součástí zdrojových kódů Tester Driveru, obvykle ale na ni nebyl kladen dostatečný důraz a jen těžko se přenášela, pokud bylo potřeba stejný protokol využít i v rámci jiného Tester Driveru. Rozšíření původního zápisu vzoru jsem provedl jednoduše pomocí argumentů samotné proměnné v rámci vzoru a i přenesení funkcionality do regulárního výrazu se ukázalo jako jednoduché, díky tomu že úpravu proměnných vzoru jsem dělal vždy před/po jeho aplikaci, díky tomu jsem prakticky touto funkcionalitou nezasáhl do funkcionality přidané v prvním kroku.

Poté co byly vzory v této podobě prakticky použity v několika Tester Driverech jsem nabyl dojmu, že jsem odvedl dobrou práci, i když složitost zápisu se začala ukazovat jako problematická, protože díky zápisu do textového řetězce nedokázalo vývojové prostředí tvůrci vzorů poradit, jaké jsou jeho možnosti a jednopísmenné identifikátory modifikací proměnných uživatel velice rychle zapomněl. Přesto se pro jednoduché vzory jednalo o funkční řešení.

Bohužel záhy na to přišly složitější protokoly vyžadující funkcionalitu nad rámec možností rozšiřitelnosti současného řešení, při zachování čitelnosti zápisu vzoru. Jednalo se o podmínky, cykly, práce s kolekcemi. Při potřebě je do sebe zanořit by byl textový zápis nejen nepřehledný ale i nad rámec možností regulárních výrazů. U nich jsem narazil na zásadní omezení kterým byl fakt že každá jeho proměnná může obsahovat pouze jednu hodnotu. To byl

zásadní problém při čtení kolekcí. Bylo tedy nutné změnit nejen způsob čtení dat za pomoci vzoru, ale stejně tak i jeho zápis.

Obojí dohromady vyžadovalo zahodit původní řešení a začít prakticky znovu. Zápis vzoru jsem v nové verzi udělal objektově, díky tomu bylo možné nejen definovat příslušné modifikátory jako metody, ale současně bylo díky hierarchické organizaci vzoru zapsat jednoduše cykly a podmínky. U čtení jsem přemýšlel déle zdali je vhodné dále pokračovat s regulárními výrazy, nebo použít jiný postup. Rozhodnutí zůstat u regulárních výrazů a obejít jejich nedostatky bylo založeno na nutnosti držet se ověřené a stabilní funkcionality, jelikož se práce se vzory stávala důležitou součástí celého frameworku. Bylo ale potřeba obejít omezení, toho jsem docílil díky rozdělení parsování vstupního řetězce do více kroků. Kdy nejprve je v rámci celého textu identifikována část patřící podmínce, nebo cyklu. Ta je vyňata a znovu parsována nezávisle na zbytku vstupu. Tento princip je aplikován rekurzivně. Díky tomu je sice parsování časově náročnější, ale je díky tomu možné použít regulární výrazy i tam kde by jednokrokové parsování nebylo použitelné.

Níže můžete porovnat původní a nový zápis vzoru.

```
// Původní zápis
// U - definuje že má hodnota být psána kapitálkami
// S(4) - definuje velikost přesně 4 znaky
// F(0) - definuje výplň chybějících znaků jako nulu
String vzor = "Error {TYPE:U} code is {CODE:S(4)F(0)}";


// Nový zápis
DataPattern vzor = new DataPattern(
      new PatternConstant("Error "),
      new PatternProperty("TYPE").toUpperCase(),
      new PatternConstant(" code is "),
      new PatternProperty("CODE").length(4).filler('0'));
```

3.4 Standardizace výjimek

Jakmile byly vytvořené vzory, mohl jsem vytvořit na jejich základě výjimky pro framework. Vzhledem k tomu že Tester Drivery nejsou aplikace, které by musely běžet v reálném čase. Mohl jsem si pro propagaci chyb použít Java výjimky, jelikož jejich náročnost na výkon můžeme zanedbat.

Prakticky jsem uvažoval nad třemi možnostmi, použít výjimky Javy, vytvořit sadu specifických tříd výjimek, nebo vytvořit pouze jednu parametrizovatelnou a společně s ní Tovární metody dle knihy Design patterns: elements of reusable object-oriented software (1) pro

vytvoření specifických instancí. Před rozhodnutím jsem si musel ujasnit, čeho mají výjimky frameworku dosáhnout. Jejich primární cíl byl rozlišit mezi neočekávanými/neošetřenými Java výjimkami a výjimkami na které dokáže framework reagovat. Tento požadavek zrušil první možnost. Druhým scénářem využití byly reakce frameworku na výjimky samotné. Pro to bylo potřeba primárně mít možnost rozšířit parametry chyby o dodatečné informace.

Díky těmto dvou scénářům a snaze nevytvářet zbytečně velké množství tříd jsem se rozhodl implementovat pouze jednu parametrizovanou výjimku a Tovární metody pro jednotlivé typy výjimek. To se ukázalo jako dobré řešení, protože aktuálně existují již desítky různých typů výjimek v rámci frameworku a pokud by nastala potřeba v budoucnu výjimky přepracovat, nebude to problém, jelikož můžu zachovat existující tovární metody.

Výjimky jsou definované výčtem, kde každý prvek obsahuje také chybovou hlášku.

```
public enum ExceptionCode implements ParametrizedException {
      FILES_MISSING_FILE(new DataPattern(...)), ...
```

Vytvoření chyby pak v rámci tovární metody vypadá následovně

```
public static TesterDaemonRuntimeException getFileNotFound(
            String pathToFile, String fileType) {
      Parameters p = new Parameters();
      p.put(ExceptionVariable.PATH,
            ObjectUtils.nullSafeToString(pathToFile));
      p.put(ExceptionVariable.TYPE,ObjectUtils.nullSafeToString(
            fileType != null ? fileType : "NOT_DEFINED"));
      return new TesterDaemonRuntimeException(
            ExceptionCode.FILES_MISSING_FILE, p);
```

3.5 Konektory

Jedním z hlavních problémů řešení vytvořených před příchodem standardizace, byla nejednotnost přístupu v rámci komunikace s jinými aplikacemi. To sebou přinášelo dva hlavní problémy. Prvním z nich byla špatná organizace zdrojových kódů, což znemožňovalo pře použít řešení na jiném stroji, protože komunikace samotná byla často pevně svázaná s komunikačním protokolem a nebylo možné je oddělit. Druhým problémem bylo ošetření chyb, které prakticky neexistovalo.

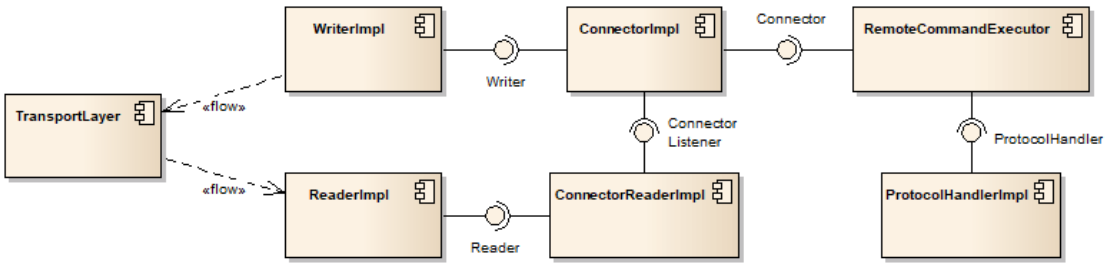
Proto mým prvním cílem v této oblasti bylo vytvořit rozhraní které by současně umožnilo komunikovat s klientem a skrýt implementační detaily samotného spojení. Kde spojením může být jakákoliv forma komunikace s jinou aplikací, běžící na libovolném stroji/počítači. Úvodní verze vycházela z návrhu, že bude existovat jednotné rozhraní pro asynchronní komunikaci s generickým datovým typem použitým pro posílaná a přijímaná data. Všechny třídy realizující komunikaci s klientem jej pak měly implementovat následující rozhraní.

```
public interface AsynchronousConnector<T> {
	String getIdentifier();
	void initialize();
	void terminate();
	boolean isValid();
	ConnectorState getConnectorState();
	// senderId pro rozlišení mezi unicastem a broadcastem
	int sendMessage(Long senderId, T outboundMessage);
	setMessageListener(AsynchronousConnectorMessageListener<T>
listener);
	// Java nemá runtime reflexi, typ musí byt explictně uveden
	Class<T> getMessageType();
```

Rozhraní poskytuje kromě metod pro příjem a zasílaní zpráv. Také metody pro navázání, zrušení spojení a informovaní o svém stavu. Nad tímto rozhraním jsem poté vytvořil nástavbu, jež umožnila pracovat s klientem synchronně.

```
public interface SynchronousConnector<T> {
      String getIdentifier();
      void initialize();
      void initialize(int retryLimit, int retryPeriodMilisec,
ExceptionCode... retryOnException) throws InterruptedException;
      void terminate();
      boolean isValid();
      ConnectorState getConnectorState();
      int sendMessage(Long senderId, T outboundMessage);
      T waitForResponse(Long senderId) throws
InterruptedException, TimeoutException;
      T getResponse(Long senderId);
      void discardUnreadResponses(Long senderId);
      Long getWaitTimeoutMilisec();
      Class<T> getMessageType();
```

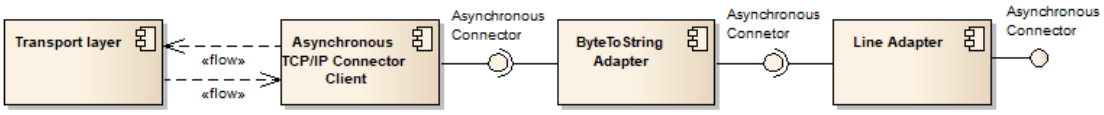
Rozhraní Synchronního Konektoru předpokládá, že příchozí data budou uložena ve frontě, dokud o ně nebude explicitně požádáno. v praxi bylo potřeba rozšířit rozhraní i o metodu jež zahodí všechny nepřečtené data. Protože v některých případech může během času, kdy není Konektor používán dojít k přijmutí zprávy, která nebyla očekávaná, příkladem může být Heart beat.



Obrázek 1.3: *Konektor úvodní verze*

Na obrázku je možné vidět samotný Konektor. Bohužel tento návrh se ukázal jako neprakticky a to kvůli rozhodnutí používat generický datový typ pro příchozí a odchozí data. Tato funkcionalita byla dosažena pomocí v Konektoru konfigurovatelných tříd pro zápis a čtení dat z transportní vrstvy. Na obrázku jsou zobrazeny jako ReaderImpl a WriterImpl. Jejich cílem bylo transformovat data mezi formáty transportní vrstvy a formátem požadovaným v aplikaci.

Takováto transformace mohla být například převod z pole bajtů na textová data v UTF-8 rozdělená po řádcích. Během prvních testů se řešení jevilo jako solidní, bohužel se ale v praxi ukázalo že tento přístup nebyl vhodný. a to kvůli nemožnosti jednoduše přidávat další funkcionalitu do již existujících tříd pro zápis a čtení dat z transportní vrstvy. Konektor totiž mohl mít pouze jednu z každého typu a přidání nové funkcionality tak znamenalo vytvořit novou třídu kde byla funkcionalita pevně daná. Tento přístup porušoval principy OOP, protože třídy plnily více než jednu funkci a důsledkem toho byly těžce rozšiřitelné. Musel jsem tedy změnit přístup a design upravit.



Obrázek 1.4: *Konektor verze s Adaptery*

Mým cílem bylo přesunout jakékoliv transformace posílaných/přijímaných dat pryč ze samotné třídy realizující Konektor. Přitom ale těmito transformacemi nezatěžovat uživatele.

Při novém návrhu jsem hledal inspiraci v návrhových vzorech. Při návrhu jsem čerpal z knihy Design patterns: elements of reusable object-oriented software (1). Na základě získaných informací jsem se rozhodl využít Design vzoru Adapter. Rozhraní Konektoru zůstalo stejné, ale z implementací jsem odstranil konfigurovatelné třídy pro posílaných/přijímaných dat. Konektory nově pracovaly pouze s datovým typem transportní vrstvy. Transformace dat se přesunuly do Adapterů. Každý s nich plní specifickou činnost, například transformaci pole bajtů na text. Na obou stranách mají Adaptery implementované rozhraní Asynchronního konektoru, díky čemu je možné je spojovat dohromady a dosáhnout tak požadované funkcionality, jež byla původně poskytována třídami typu Reader a Writer. Obrázek 1.2 ukazuje konfiguraci, pokud chcete přijímat a číst textové řádky přes TCP/IP.

Následujícím rozšířením, které plánuji je přidání propagace událostí. Těmi nejzákladnějšími bude změna stavu spojení, to umožní rychlejší odezvu při zjišťování stavu konektoru, nyní je třeba se periodicky doptávat na stav konektoru. Druhým důvodem je poté implementace konektorů pro spojení, které události aktivně používají v rámci aplikace jako například protokol SECS/GEM, který je celosvětově uznávaným standardem.

3.5.1 Inicializace a Terminace spojení

Jedním z úvodních cílů Konektoru bylo zapouzdření inicializace spojení, detekce ukončení a opětovné navázaní spojení. Tato funkcionalita chyběla prakticky ve všech aplikacích, ve kterých měly být nové Konektory být použité. Obvyklé řešení inicializovalo spojení jednou a poté už jeho stav nekontrolovalo. Vzhledem k tomu že v praxi pravidelně docházelo k situacím, kdy bylo spojení přerušeno, například při výpadku sítě, nebo restartu aplikace. Bylo potřeba, aby Konektory měly pro tento případ připravenou funkcionalitu pro obnovu spojení. Tato funkcionalita může být spuštěna buď explicitně metodou Konektoru, nebo je volána automaticky když začne být konektor aktivně využíván. Oba přístupy je možné volat

opakovaně, pokud se tak stane a Konektor je již v cílovém stavu, tak se volaní příkazu ignoruje. Úvodní verze Konektorů předpokládaly, že o volání těchto metod se bude starat uživatel, to se ale ukázalo jako zbytečně složité.

Jedním ze zásadních problémů se kterým jsem se ve verzi s adaptéry potýkal, byla generika Javy. Ta se totiž po překladu aplikace ztratí. To znamenalo výzvu, protože Tester Drivery spouštíme za pomoci Spring IOC. Jednoduše tak mohlo dojít k případu, kdy vlivem špatné konfigurace mohlo dojít ke spojení například Konektoru pracujícího s polem bajtů a Adapteru vyžadujícího textová data. Taková chyba se pak projevila, až když byly přijaty, nebo poslány první data. Mým cílem bylo aby takový problém byl odhalen při spuštění aplikace, kdy je u počítače přítomen Inženýr schopný konfiguraci opravit.

Jednou z možností bylo poslat testovací data do Konektoru a čekat na výjimku. Tato metoda ač jednoduchá na realizaci neposkytovala potřebnou jistotu. Výjimka mohla být omylem zahozena, nebo pokud nastala tak získat z ní potřebné informace bylo na aplikační úrovni nelehké.

Proto jsem nakonec přistoupil ke složitějšímu postupu. Kdy rozhraní Konektoru a Adaptéru publikuje třídu datového typu použitého na jeho vstupu a výstupu. Při spojování Konektoru s Adaptery se pak kontroluje, zda jsou jejich datové typy stejné, nebo převeditelné implicitní konverzí. Pokud ne aplikace vrátí chybu obsahující informace o místě, kde problém nastal.

3.5.2 Unicast, Broadcast, Multicast

Zásadní otázkou při návrhu Konektoru bylo, zda vytvořit jednotné rozhraní pro použití Konektoru jako Server/Klient/Peer, nebo zda pro každý z nich vytvořit vlastní rozhraní. Z pohledu Tester Driverů jsou tyto informace nepodstatné, aplikace chce pouze poslat/přijmout data a o víc se obvykle nechce starat. Proto jsem vytvořil jednotné rozhraní, které je možné použít pro všechny tři scénáře. Každému navázanému spojení je přiřazen identifikátor. Tento identifikátor se předává společně s přijatou zprávou skrze Konektor listener.

```
public interface AsynchronousConnectorMessageListener<T> {
      void receiveMessage(Long senderId, T inboundMessage);
      Class<T> getListenerMessageType();
}
```

Tento identifikátor si může klient uložit a při posílaní odpovědi pak předat konektoru společně s odesílanou zprávou. v takovém případě je zpráva poslána pouze skrze spojení s tímto identifikátorem. Díky tomu můžeme realizovat jednoduše unicast a multicast. Broadcastu pak docílíme, pokud není předán se zprávou žádný identifikátor. Zpráva je pak poslána všem aktuálně připojeným spojením.

Pokud tedy používáme Konektor jako Klient můžeme identifikátor ignorovat, protože budeme mít vždy maximálně jedno spojení. Ve scénáři kdy potřebujeme Konektor jako Server nebo Peer využijeme identifikátor dle potřeby.

3.5.3 Příjem a odesílání dat

Pro odeslání dat je třeba Konektoru předat data v transportním datovém typu, tím obvykle bývá pole bajtů nebo textový řetězec, společně s identifikátorem spojení. Pokud je identifikátor definován pak se data pošlou jen skrze specifické spojení, v opačném případě pak skrze všechny aktivní spojení.

Při příjmu dat je třeba rozlišit mezi synchronním a asynchronním příjmem. Asynchronní příjem dat zaručuje konektor listener který musí být implementován na straně klientské aplikace. Ten poskytuje jak zprávu samotnou tak identifikátor spojení skrze které byla přijata

V případě synchronního příjmu pak samotné rozhraní synchronního konektoru poskytuje metody pro příjem dat. U těch je možné zadat identifikátor spojení a v takovém případně budou poskytnuta data pouze příchozí z tohoto spojení.

3.5.4 Modifikace dat před odeslání a po přijetí

Aby uživatel nemusel pracovat přímo s přijatými daty je možné použít Konektor Adaptery které mohou data modifikovat. Důvodem pro jejich vznik bylo odstranit z aplikace kód který by se staral o přípravu dat. Přípravou dat mám na mysli například konverzi bajtů na textový řetězec na základně zvoleného enkodování, odstranění rozdílů způsobené konverzí mezi little a big endian, spojení fragmentovaných dat, případně rozdělení příchozích dat podle známého separátoru (například konec řádku). Tyto operace byly v původních Tester Driverech řešeny přímo v nich a mnohá řešení nepočítaly se všemi chybami a možnými nastaveními. Díky tomu bylo v případně nelezení chyby v jedné aplikaci těžké určit, které ze zbylých jsou také ovlivněny.

Obecně můj cíl byl umožnit změnit transportní vrstvu bez toho, aby se měnil zdrojový kód aplikace. Příkladem může být záměna transportní vrstvy mezi TCP/IP a Sériovou Linkou. Tato funkcionalita se v praxi ukázala jako jedna z nejdůležitějších součástí konektoru, protože nám umožňuje nasadit jednou otestovaný protokol na nový stoj, který nemusí mít stejnou transportní vrstvu.

Z pohledu aplikace implementují adaptéry rozhraní asynchronního konektoru a díky tomu je možné je přidat bez toho aby byly potřeba změny v aplikaci. Adapter skrze rozhraní sice poskytuje funkcionalitu asynchronního konektoru interně ale operace, které sám nemodifikuje, ale deleguje na svého podřízeného, kterým může být zdrojový konektor, nebo další adaptér. Díky tomu se adaptér z pohledu aplikace chová jako plnohodnotný konektor.

3.6 Komunikační protokoly

Tester Drivery tvoří rozhraní mezi stroji ve výrobě a řídícím softwarem. Proto je možnost jednoduše komunikovat s jinými zařízeními a aplikacemi velice důležitá. Komunikace samotná bývá obvykle definována v rámci protokolu, který není obecně uznávaným standardem. Důvodem jsou peníze, rozdíl mezi nákupní cenou stroje s globálně uznávaným protokolem jako je například SECS/GEM oproti stroji s protokolem který obsahuje minimální implementaci a byl vyvinut pouze pro daný stroj, může být i desítky tisíc dolarů. Bavíme-li se o nákupu desítek strojů, vyjde pro firmu levněji vzít druhou variantu a práci s integrací protokolu zadat firemnímu IT. To je primární důvod proč jsem se snažil komunikační protokoly podchytit v rámci frameworku.

Úvodní plán vycházel z následujících jednoduchých scénářů:

- Odeslání příkazu > bez odpovědi
- Odeslání příkazu > konečná odpověď
- Odeslání příkazu > průběžné odpovědi > konečná odpověď

Nejprve bylo třeba rozhodnou jakým způsobem spojit protokol, konektor kterým se bude komunikovat a uživatele který příkaz protokolu potřebuje zavolat. Zde jsem vycházel ze zkušeností mých kolegů kteří podobný problém řešili v minulosti v rámci jedné třídy což se ukázalo jako nevhodné řešení z důvodů rozšiřitelnosti. Proto jsem se rozhodl použít k dosažení cílové funkcionality kompozici. První bylo třeba definovat rozhraní kterým disponuje uživatel pojmenované RemoteCommandExecutor.

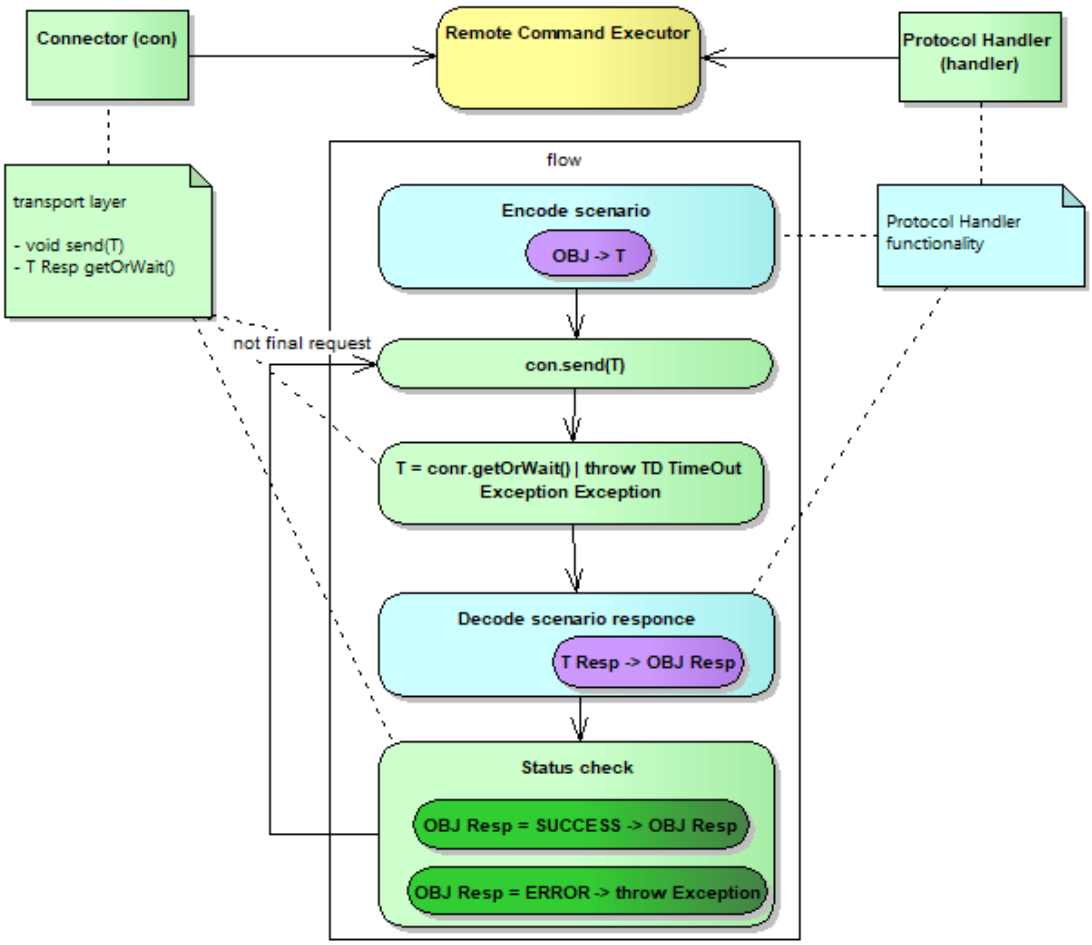
```
public class RemoteCommandExecutor {
      public static <T> CommonRemoteResponse perform(
            CommonRemoteRequest request,
            ProtocolHandler protocolHandler,
            SynchronousConnector<T> connector,
            boolean sendRequestOnlyOnce,
            boolean waitForResponse,
            boolean returnOnlySuccesses
      ) throws InterruptedException
}
```

Volání metody obsahuje tři důležité argumenty. CommonRemoteRequest je rozhraní přes které je možné získat jméno příkazu, vstupní parametry příkazu, vzory pro odesílané a přijímané data.

ProtocolHandler definuje metody pro převody mezi CommonRemoteRequest, CommonRemoteResponse a datovým typem používaným konektorem.

Posledním argumentem je poté konektor přes který bude komunikace probíhat.

Spojením těchto tří argumentů víme co chceme posílat, jak to chceme poslat a komu se má poslat. Samotný RemoteCommandExecutor pak zajistí převod příkazu do transportního formátu, jeho odeslání, přijetí případné odpovědi, její převod z transportního formátu na Java objekt, kontrola zda příkaz uspěl a vrácení odpovědi uživateli pro následné zpracování dat z odpovědi.



Obrázek 1.5: *Schéma funkcionality RemoteCommandExecutoru pro SimpleProtocolHandler*

```
// Ukázka volání příkazu
CommonRemoteResponse response = RemoteCommandExecutor.perform(
      new CommonRemoteRequest(OnTesterCommandV1.ABORT_TEST, p),
      protocolHandler, testerConnector);

// Příkazy definované v rámci výčtu
public enum OnTesterCommandV1 implements PatternCommand {
      ABORT_TEST(new DataPattern(
            new PatternConstant("ABRTEST-MID:"),
            new PatternProperty(CmdParameters.CMD_ID),
            new PatternProperty(CmdParameters.CMD_TERMINATOR))
      ), ...
}
```

Tímto přístupem se podařilo docílit volaní příkazu v rámci Tester Driveru z jediného řádku a pro výše uvedené příklady komunikace je použitelný, protokol handler a výčet příkazů může dle potřeby být definovaný v rámci Tester Driveru, nebo ve sdílené knihovně. Zbylé komponenty jsou dostupné vně frameworku.

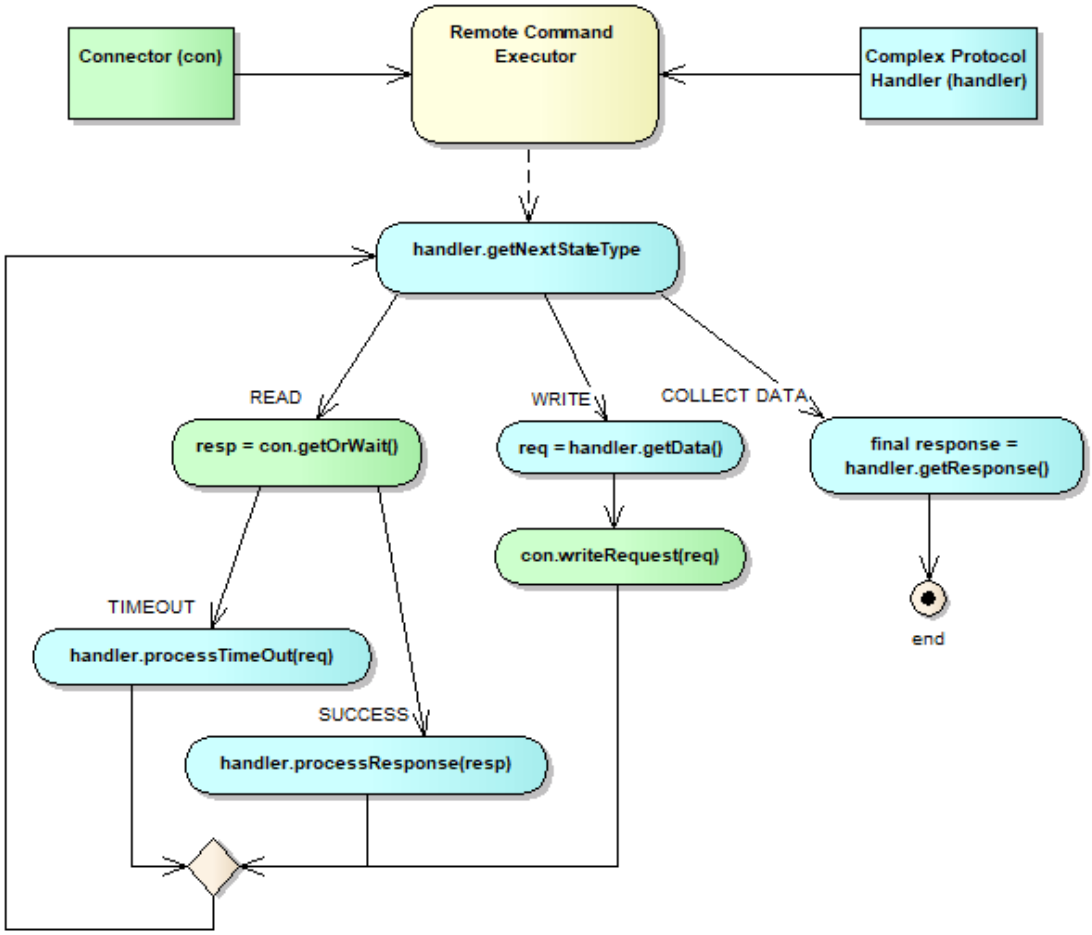
Bohužel reálné protokoly mohou být i mnohem složitější a snažit se vložit do výše popsaného modelu zpracování časovačů, signálů informující klienta o nutnosti počkat z důvodu vysokého vytížení a faktu že některé protokoly jsou tak detailní že to co je z pohledu uživatel například restart, může v rámci protokolu být několik příkazů. Proto bylo potřeba přijít s modelem který by mohl uživatel používat stejným způsobem, ovšem s rozšířenými možnostmi. Vzhledem k tomu že se první verze osvědčila snažil jsem vyjít z ní. Výsledkem byla definice příkazu za pomoci Stavového stroje, se kterým jsem se prakticky setkal ve škole. Ten se přemístil do protokol handlenu a namísto transformací nyní fungoval, jako stavový stroj který kontroluje celý proces komunikace.

```
public interface ComplexProtocolHandler <P, i extends Enum, O>{
      I getCommand();
      ComplexProtocolStepType getNextStep();
      P getNextMessage();
      void processReply(P reply);
      void processTimeout(TimeoutException e);
      O getResponse();
}

public enum ComplexProtocolStepType {
    SEND_MESSAGE,
    WAIT_FOR_REPLY,
    COLLECT_RESULT
}
```

V tomto novém protocol handleru je nejdůležitější metodou getNextStep ta vrací informaci o tom, která ze tří možných operací se má provádět. RemoteCommandExecutor poté provádí adekvátní operace odesílání, nebo čtení dat. Dokud se stavový stroj nedostane do terminálního stavu, ve kterém dokáže poskytnout odpověď na daný příkaz. Tento protokol se ukázal jako plně funkční díky vyjadřovací síle stavového stroje. Komplikací se ukázal pouze popis stroje samotného. Proto byly pro snadné použití potřeba předpřipravit funkcionalitu která by dokázala jednoduše provést jednodušší operace v rámci stavového stroje, jako je například příprava dat k odeslání a posun vnitřního stavu.

```
public interface ComplexProtocolCommandState {
      String getStateName();
      ComplexProtocolStepType getNextStep();
      DataPattern getSendPattern();
      ComplexProtocolCommandState getDefaultNextState();
      List<Pair<String, ComplexProtocolCommandState>>
            getNextStateTransitions();
      ComplexProtocolCommandState getTimeoutNextState();
      boolean equals(ComplexProtocolCommandState state);
}
```



Obrázek 1.6: *Schéma funkcionality RemoteCommandExecutoru pro ComplexProtocolHandler*

Tento postup se osvědčil vzhledem k tomu, že šel opět elegantně využít popis stavů pomocí výčtového typu. Bylo třeba ale vyřešit zásadní limitaci jazyku Java. Java bohužel během kompilace výčtového typu prochází pouze jednou, proto není v základu možné v něm udělat křížový odkaz. To se naštěstí dá obejít díky tomu že nejprve provedeme deklaraci výčtových prvků a až poté nastavení jejich parametrů, ve zdrojových kódech to vypadá zvláštně, ale je to jediný použitelný postup který jsem našel. Volání příkazu s ComplexProtokolHandlerem je prakticky stejné jak úvodní verze. Neobsahuje však dodatečné přepínače, které jsou součástí stavového stroje a pro každé volání příkazu se vytváří nová instance protokol handleru protože ten obsahuje stavový stroj a jedná se o bezpečnější přístup než resetování a pře použití jedné instance.

```
CommonRemoteResponse response = RemoteCommandExecutor.perform(
      new PowerTechHostLinkComplexProtocolHandler(
            PowerTechHostLinkCommand.OPEN_DATALOG_FILE, p),
      hostLinkConnector);
```

3.7 Práce s lokálními daty

Součástí frameworku jsou i části připravené pro práci s daty na samotném počítači který patří testeru. s návrhem a vývojem těchto komponent nebyl povětšinou žádný problém. Uživatele by ale mohlo napadnou proč nepoužít přímo funkcionalitu kterou poskytuje Java, namísto zapouzdřovaní této funkcionality do frameworku. k tomuto zapouzdření vedlo několik praktických důvodů.

Prvním z nich bylo rozhraní poskytované jazykem Java, to pracuje s objektovou reprezentací souborů, zatímco uživatel definuje v rámci konfigurace a argumentů příkazu fragmenty ze kterých se výsledné cesty a jména souborů skládají. Rozhodl jsem se tedy vytvořit dle návrhových vzorů Fasádu, která by poskytla funkcionalitu žádanou uživatelem ve formě, která bude pro uživatele ideálně znamenat volání jedné funkce pro každou požadovanou operaci.

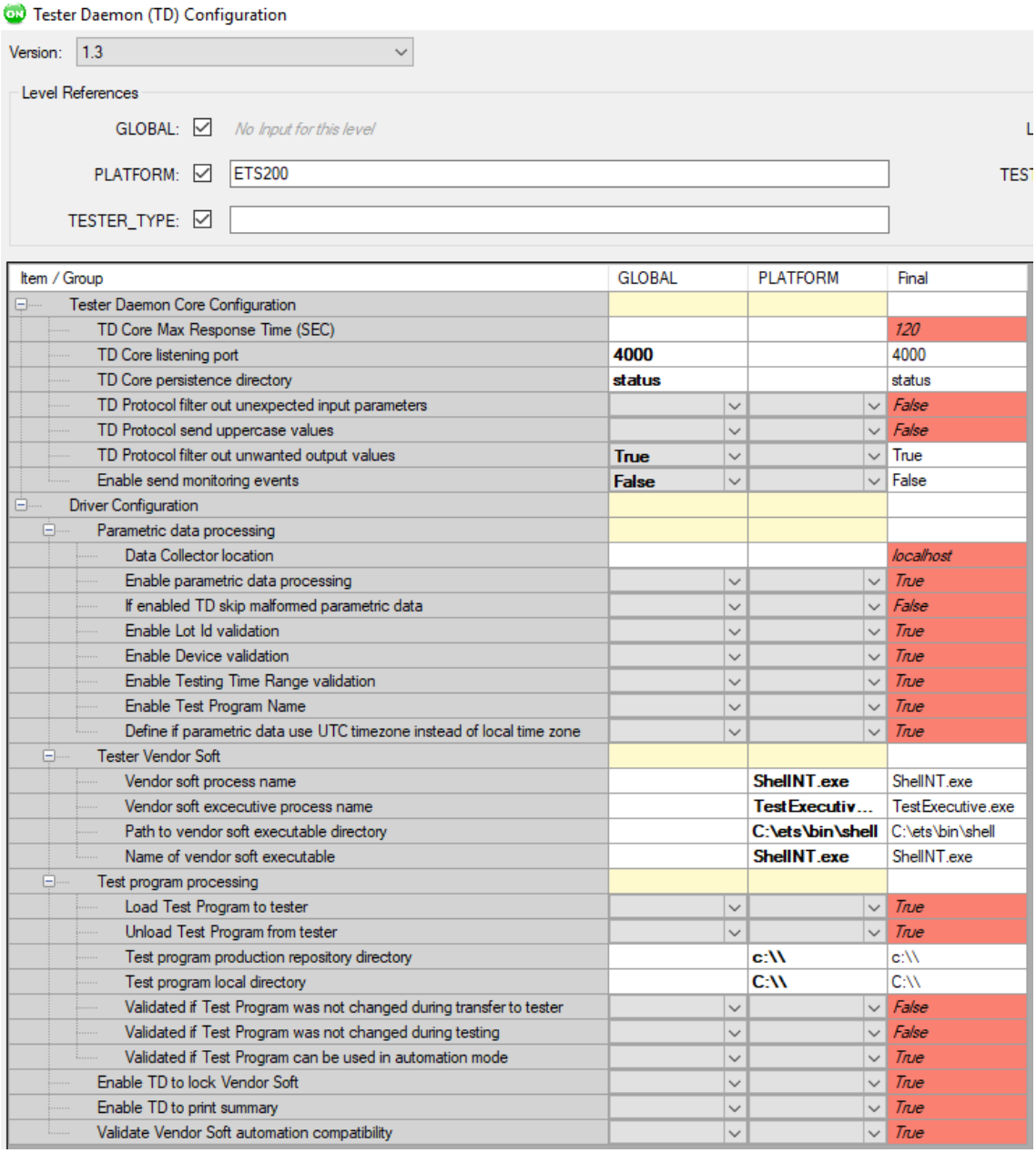
Druhým důvodem byla kontrola vstupních dat a přístupových práv. Zvláště pak kontrola práv se ukázala jako jedna z nejdůležitějších funkcí. Java na systému Windows přístupová práva příliš neřeší a pokud se pokusíte s takovým souborem pracovat začne se chovat neočekávaně. Proto bylo potřeba vždy explicitně provést kontrolu práv. Díky tomu jsme schopni uživatele o takové situaci přesně informovat.

Další částí práce s daty jsou validace dat vyprodukovaných samotným Testerem. Z pohledu výroby je snaha provést kontrolu co nejdříve. Pokud je problém detekován později je obvykle nutné provést celé měření znovu, včetně plánování. Zatímco pokud je problém detekován okamžitě je možné jej mnohdy vyřešit jednoduše.

3.8 Konfigurace aplikace

Konfigurace aplikace prošla pozvolným procesem přeměny. Když jsem začal pracovat, přistupoval jsem k práci podobně jako ke školním projektům, které nikdy nebylo třeba udržovat rozšiřovat, nebo přenastavit. Současně jsem celkem mylně předpokládal vyšší úroveň znalostí týmu, který bude aplikaci poskytovat online podporu. Postupem času jsem pochopil způsob jakým tyto týmy pracují a rozsah jejich nasazení. Díky tomu jsem začal více chápat že úspěch aplikace z velké části zaručí i jednoduchost její podpory a nastavení. Proto jsme v průběhu vývoje přešli z lokálního nastavení v rámci IOC kontextu (xml) na konfigurační soubory a poslední iterací je globální nastavení skrze webovou službu.

Pro potřeby realizace bylo potřeba vyřešit situace, kdy není připojená síť. Automatizace výroby musí fungovat, byť v omezené míře i v případě že není navázané síťové spojení. v rámci frameworku proto bylo potřeba doplnit tuto funkcionalitu o lokální zálohu, která může v případně výpadku nahradit data poskytované webovou službou.



Item / Group	GLOBAL	PLATFORM	Final
Tester Daemon Core Configuration			
TD Core Max Response Time (SEC)			*120*
TD Core listening port	**4000**		4000
TD Core persistence directory	**status**		status
TD Protocol filter out unexpected input parameters			*False*
TD Protocol send uppercase values			*False*
TD Protocol filter out unwanted output values	**True**		True
Enable send monitoring events	**False**		False
Driver Configuration			
Parametric data processing			
Data Collector location			*localhost*
Enable parametric data processing			*True*
If enabled TD skip malformed parametric data			*False*
Enable Lot Id validation			*True*
Enable Device validation			*True*
Enable Testing Time Range validation			*True*
Enable Test Program Name			*True*
Define if parametric data use UTC timezone instead of local time zone			*True*
Tester Vendor Soft			
Vendor soft process name		**ShellNT.exe**	ShellNT.exe
Vendor soft excecutive process name		**TestExecutiv...**	TestExecutive.exe
Path to vendor soft executable directory		**C:\ets\bin\shell**	C:\ets\bin\shell
Name of vendor soft executable		**ShellNT.exe**	ShellNT.exe
Test program processing			
Load Test Program to tester			*True*
Unload Test Program from tester			*True*
Test program production repository directory		**c:**	c:\\
Test program local directory		**C:**	C:\\
Validated if Test Program was not changed during transfer to tester			*False*
Validated if Test Program was not changed during testing			*False*
Validated if Test Program can be used in automation mode			*True*
Enable TD to lock Vendor Soft			*True*
Enable TD to print summary			*True*
Validate Vendor Soft automation compatibility			*True*

Obrázek 1.7: *Ukázka rozhraní pro konfiguraci Tester Driveru*

Konfigurace samotná je organizovaná po úrovních kdy začínáme úrovní Global která je společná pro všechny stroje v rámci továrny a postupně s rostoucí prioritou se blížíme stále více k jedinému stroji. Jako výsledná hodnota parametru se poté vezme hodnota z úrovně s nejvyšší

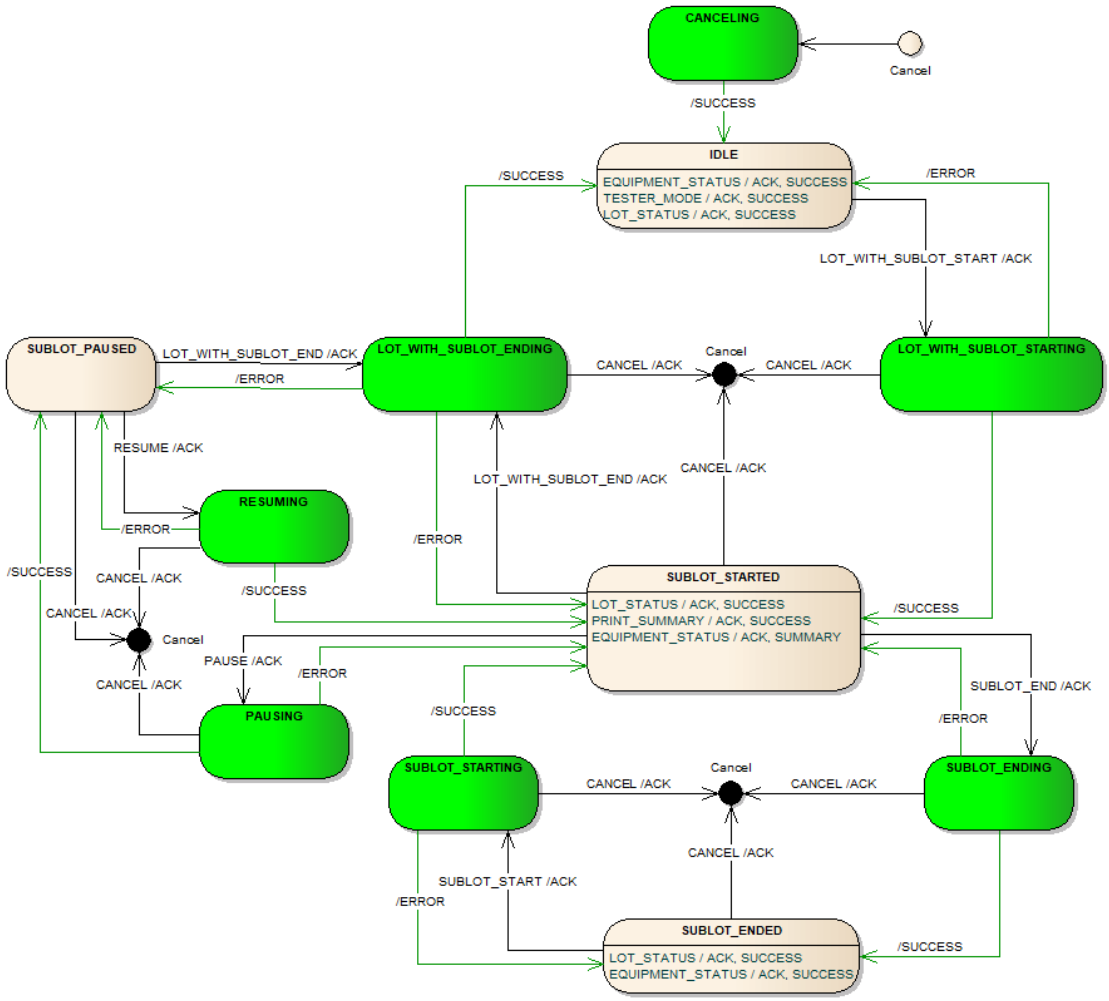
prioritou, případně defaultní hodnota. Díky tomu není potřeba opakovat konfiguraci neustále dokola pokud je stejná pro určitou podskupinu testerů, například pro všechny testery na jedné výrobní lince.

3.9 Jádro aplikace

Poslední velkou části frameworku je jádro aplikace připravené pro Tester Driver. Oproti ostatním částem frameworku je přímo určen pro Tester Drivery, zatímco ostatní části frameworku byly vyvíjeny s ohledem na použití i v jiných typech aplikací.

Předtím než představím jeho obsah a funkcionalitu je potřeba uvést co přesně Tester Driver představuje. Jedná se o aplikaci která poskytuje skrze jednotné rozhraní funkcionalitu která nahrazuje interakci s testerem kterou původně prováděl operátor výroby. Rozhraní společně s příkazy definuje také kdy je možné jednotlivé příkazy volat.

Pro příjem příkazů a odesílání odpovědí jsou použité Konektory a Protocol Handler které jsou součástí frameworku.



Obrázek 1.8: *Schéma platných příkazů pro Final Test tester*

Díky tomuto zadání se jako rozumné zdálo použít stavový stroj pro popis aktuálního stavu a určení které příkazy jsou v danou chvíli platné. Oproti předchozímu využití stavového

stroje v rámci frameworku jsem musel do stavového stroje přidat i tranzitní stavy (zelené na obrázku 1.5). Samotné příkazy lze rozdělit do dvou kategorií.

Nestavové příkazy které je možné volat periodicky a nemění stav aplikace. Jedná se o příkazy pro poskytnutí informací o aktuálním stavu, hear beat a online monitoring. Ty lze volat nezávisle na sobě a Tester Driver je musí být schopný provádět paralelně. Na obrázku jsou zobrazeny uvnitř jednotlivých stavů.

Druhým typem jsou stavové příkazy, ty naopak mění vnitřní stav Tester Driveru a je nutné je provádět sekvenčně, pokud tedy dojde k připojení příkazu před skončením předchozího je potřeba nově příchozí příkaz zkontrolovat a zařadit do fronty. Samotné příkazy mohou v závislosti na datech trvat i hodiny a díky přidání tranzitních stavů bylo jednoduché rozšíření pro identifikaci běhu příkazu. Důvod pro tento postup byla snaha, aby pokud je zavolán stavový příkaz před koncem předchozího příkazu, byl nově příchozí odmítnut. Tento návrh měl ale zásadní chybu. Stavové příkazy používaly pro své předzpracování a kontrolou stejné zdroje jako nestavové příkazy, mohly tedy tyto kroky provést paralelně. Pokud byly příkazy volány současně tak díky paralelnímu průběhu ověření mohly oba projít validací a následně být provedeny i když by mělo dojít k selhání jednoho z nich. Díky tomuto problému jsem musel kontrolu, zda je možné příkaz provést přenést až do samotného startu příkazu.

Samotný stavový stroj byl v původní verzi součástí zdrojového kódu, to bylo dostatečné pro původní verze Tester Driverů, které byly vyvíjeny pouze pro Final Test a obsahovaly prakticky pouze čtyři příkazy. Společně se začátkem vývoje frameworku došlo k podstatnému rozšíření očekávané funkcionality. Byly přidané nové příkazy, některé z nich pak měly být podporované pouze pro Final Test, nebo Probe. s tímto rozšířením byl stavový stroj potřebný pro realizaci na hraně čitelnosti, proto jsem se rozhodl využít zkušeností nabytých při tvorbě protokolů a stavový stoj definující možné volání stavových příkazů v konfiguraci.

```
public class StateMachineTransition {
      private TesterDaemonState sourceState;
      private CommandType command;
      private TesterDaemonState transitionState;
      private TesterDaemonState successState;
      private TesterDaemonState errorState;
}
```

Díky tomuto zápisu není potřeba měnit jádro pokud je potřeba změnit stavy kde se dají příkazy volat. Za předpokladu že jejich implementace bude mimo jádro.

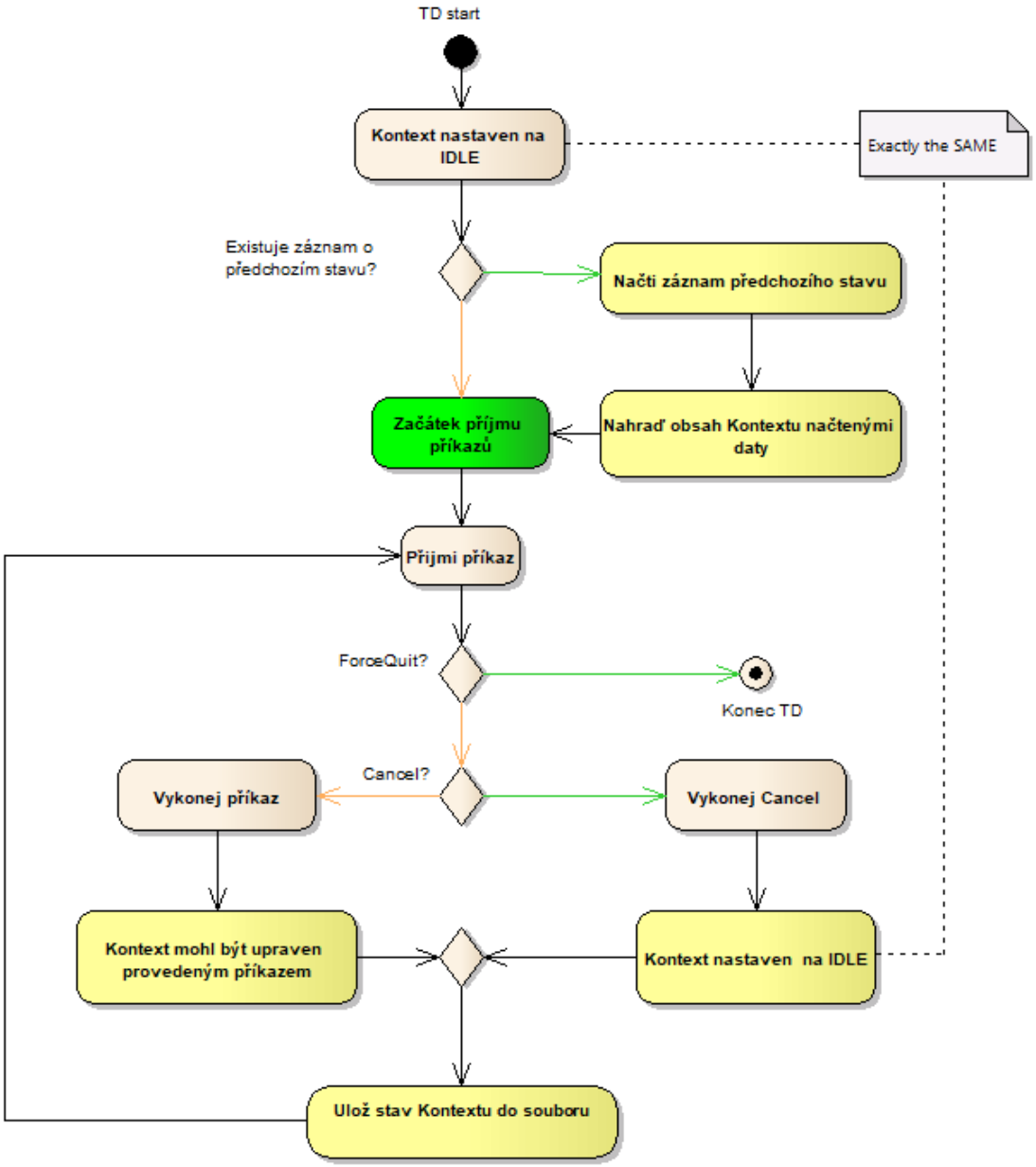
3.10 Persistence

Persistence je jedna z posledních částí frameworku která byla přidána. Pokud nenastane žádná chyba během měření tak není vůbec potřeba. Důvody pro její uvedení byly dva. První z nich je čistě uživatelská nešikovnost, pokud totiž uživatel omylem vypnul Tester Driver, nebylo možné, kvůli limitacím testeru samotného, jej po startu synchronizovat se stavem testeru, který mohl zůstat nedotčen. Druhým pak byly situace, kdy došlo k chybě na straně Tester Driveru, například díky špatné konfiguraci. Oprava takové chyby vyžaduje s současné verzi restart. Díky čemu dojde stejně jako v předchozím případě ke ztrátě kontextových dat. Uživatelé v takovém případě byli nuceni provést celý proces měření znovu, vzhledem k tomu že proces může trvat hodiny, až dny jednalo se o nedostatek, který bylo potřeba odstranit.

Řešením tohoto problému bylo uvedení persistence do Tester Driveru. Prvotním krokem bylo rozdělení kontextových informací na obnovitelné a neobnovitelné. Kdy obnovitelné jsme schopni reprodukovat na základně konfigurace, je jím například TCP/IP spojení s testerem. Naopak neobnovitelné jsou data poskytnutá uživatelem, nebo testerem na které není možné se opětovně doptat. Při použití frameworku existují tři místa, kde se neobnovitelné kontextové informace objevují.

Kontext jádra frameworku držící informace o aktuálním stavu stavového stroje. Kontext Lotu, který představuje obecné nastavení pro měření a Sublot/Wafer kontext který představuje specifické nastavení pro Sublot, nebo Wafer. Pro uchování těchto dat jsem se rozhodl použít soubory typu xml, které budou obsahovat poslední známý stav. Pro xml jsem se rozhodl díky tomu, že je velice snadné v jazyce Java převádět objekty na XML a naopak díky podpoře, kterou jazyk poskytuje.

V případě Tester Driverů bylo cílem udržet persistentní stav mezi jednotlivými příkazy. O rozšíření funkcionality i na průběh jednotlivých příkazů zatím neuvažuji, protože s tímto scénářem zatím neumí pracovat řídící software stanice. Správa persistence je součástí frameworku a z pohledu uživatele se jedná o funkcionalitu, která funguje automaticky v každém Tester Driveru.



Obrázek 1.9: *Schéma udržení persistence Tester Driveru*

3.11 Simulátor

Během vývoje frameworku se začaly ozývat požadavky na tvorbu simulátoru Tester Driverů. v mnoha případech probíhal vývoj řídicího systému stanice současně s vývojem Tester Driverů pro danou část výroby. Pokud se během vývoje driveru vyskytly problémy vedoucí ke zpoždění dodacích termínu, nebo pokud neměl tým realizující implementaci řídicího systému zkušenost s použitím Tester Driveru, docházelo k problémům, které mnohdy vedly k neplánovaným změnám v kódu aplikací, protože aplikace spolu poprvé komunikovaly až během testování s reálnými stroji v produkci.

Rozhodl jsem se proto do frameworku přidat i simulátor, který se z pohledu uživatele chová stejně jako skutečný Tester Driver. Navíc je možné nadefinovat, jak se má chovat pro každý definovaný příkaz. Kdy pro každý z nich je možné nadefinovat více reakcí a simulátor si mezi nimi poté vybere dle vstupních parametrů.

Samotné nastavení simulátoru je provedeno v rámci konfigurace a ne zdrojových kódů. Díky tomu je možné, aby si uživatel definoval chování na míru, nebo jej vytvořit společně během fáze analýzy a tým odpovědný za Tester Driver jej může použít pro kontrolu implementovaného chování a tým odpovědný za systém řízení stanice jej má k dispozici pro vlastní vývoj.

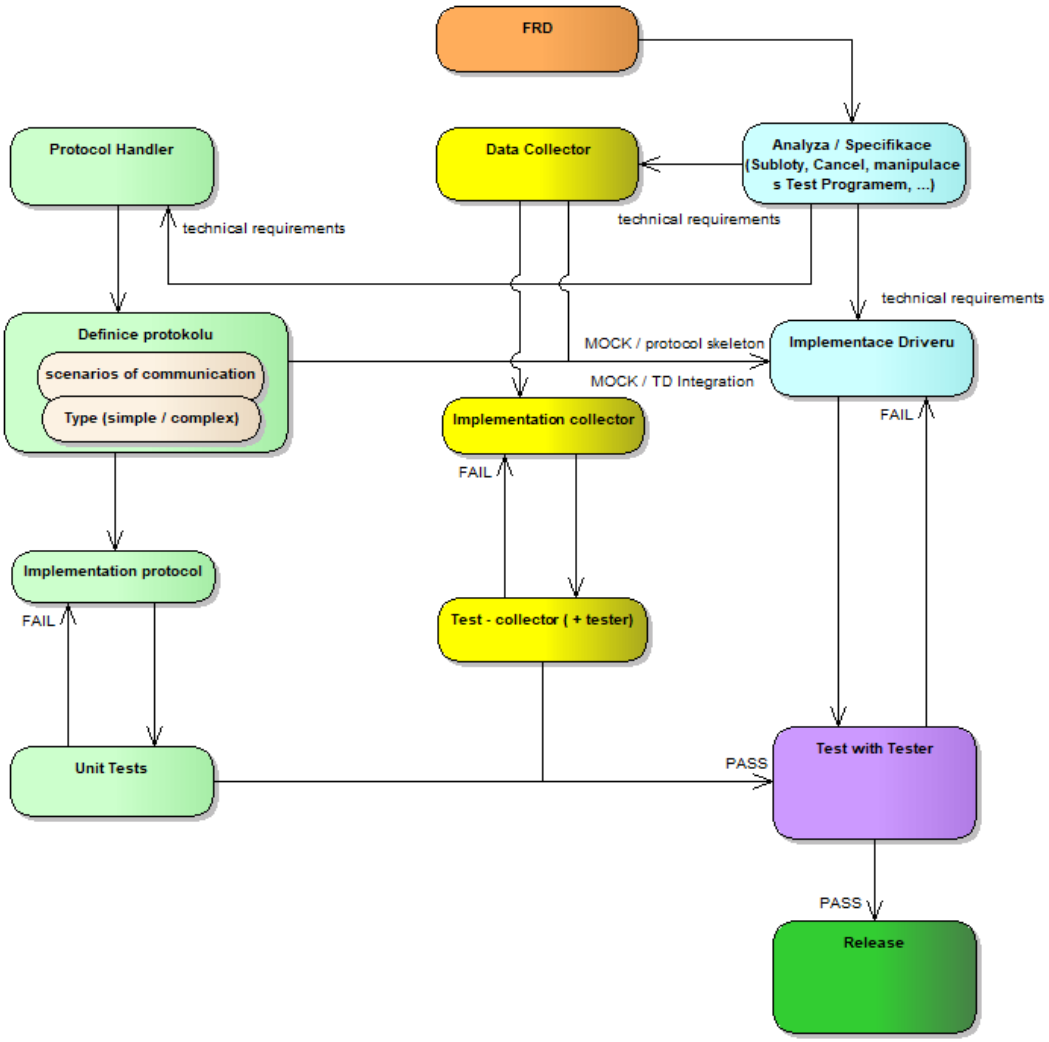
Díky tomu došlo ke zlepšení vztahů mezi týmy a větší popularizaci Tester Driverů, díky jejich jednoduché prezentaci a vyzkoušení skrze simulátor.

4 Využití frameworku v praxi

V této kapitole si ukážeme příklad využití frameworku na příkladu z praxe pro blíže nespecifikovaný tester. Pro přehlednost budou demonstrovány jednotlivé kroky vývoje, tak jak odpovídají běžnému vývoji v rámci firmy.

4.1 Analýza požadavků a možností jak je realizovat

Prvním krokem vývoje je určení kde bude Tester Driver nasazen a jaké jsou požadavky na implementované příkazy. Zde rozlišujeme hlavně mezi tím zdali je tester určen pro Probe, nebo Final Test. Používají totiž jinou úroveň kontroly testeru z pohledu systému řízení testovací stanice. Dále budeme pokračovat s příkladem pro Final Test, protože využívá více komponent frameworku.



Obrázek 1.10: *Vývoj Tester Driveru*

Vývoj driveru jak můžeme vidět z obrázku můžeme rozdělit na tři základní části. Těmi je samotná implementace Tester Driveru, implementace potřebných komunikačních protokolů a aplikace pro sběr dat. Ta se nazývá Data Kolektor a původně tato funkcionalita byla součástí Tester Driveru. Jedná se ale o funkcionalitu, která musí být extrémně robustní, rychlá a použitelná i v případě že je potřeba tester ovládat manuálně. Proto byla vyčleněna do vlastní aplikace.

Data Kolektor stejně jako Tester Driver používá komponenty z frameworku v rámci vlastního jádra aplikace a funkcionality. v rámci ukázky na něm bude demonstrováno využití komponent frameworku mimo Tester Drivery.

Poznámka - v následujících příkladech budou uváděna jména potřebných tříd ve formátu "knihovna ve které je implementován : jméno třídy bez balíku ve kterém je umístěna". Rozhodl jsem se tak kvůli přehlednosti. Jména tříd jsou unikátní a nebude problém je dohledat.

Poznámka 2 - Framework byl vyvíjen pro použití v mezinárodní fimě, proto jsou všechny ukázky v angličtině. Raději jsem chtěl ukázat fragmenty kódu které jsou reálně používané , než vytvářet umělé příklady. v popisech se budu dál držet českého názvosloví.

Poznámka 3 - Příklady zdrojových kódů jsou v této kapitole jako obrázky, protože při zachování předepsaného formátování textu by byly nepřehledné

4.2 Implementace "simple" protokolu

Simple protokol představuje libovolný protokol který není potřeba implementovat stavovým strojem. Pro realizaci budeme potřebovat:

- tester-daemon-model : DataPattern pro definici vzorů příkazů a odpovědí
- tester-daemon-model : PatternCommand je rozhraní pro definici příkazů protokolu
- tester-daemon-model : ProtocolHandler je rozhraní pro implementaci překladu příkazů a odpovědí z/do transportního formátu.

Začneme definicí příkazů. Zde se mě osvědčilo využít možnosti implementovat rozhraní ve výčtovém typu. Díky tomu získáme výhody příjemné práce s výčtovým typem a současně jej můžeme kdykoliv vyměnit díky tomu že s ním budeme dále pracovat skrze rozhraní PatternCommand.

V následujícím příkladu je protokol využívaný pro asynchronní sdílení dat mezi Tester Driverem a Data Kolektorem. Popis protokolu obsahuje všechny příkazy které protokol využívá nezávisle na tom kdo je posílá. Dva z nich pro žádost a uvolnění dat jsou posílány Tester Driverem, poslední z nich poskytuje data a posílá jej Data Kolektor.

V rámci definice příkazů jsou použity vzory. Na příkladu PROVIDE_DATA příkazu je demonstrován zápis volitelného parametru ve vzoru.

```
public enum TesterDaemonProviderCommand implements PatternCommand {

    REQUEST_DATA(new DataPattern(
            new PatternConstant( value: "REQUEST_DATA DATALOG_LOT:"), new PatternProperty(DriverParameter.DATALOG_LOT),
            new PatternCondition(DriverParameter.WAFER).isDefined(
                    new PatternConstant( value: " WAFER:"), new PatternProperty(DriverParameter.WAFER)),
            new PatternConstant( value: " REQUESTER:"), new PatternProperty(DriverParameter.ID),
            new PatternCondition(DriverParameter.TEST_MODE).isDefined(
                    new PatternConstant( value: " TEST_MODE:"), new PatternProperty(DriverParameter.TEST_MODE)),
            new PatternConstant( value: "\r\n").writeOnly())),

    RELEASE_DATA(new DataPattern(
            new PatternConstant( value: "RELEASE_DATA REQUESTER:"), new PatternProperty(DriverParameter.ID),
            new PatternConstant( value: "\r\n").writeOnly())),

    PROVIDE_DATA(new DataPattern(
            new PatternConstant( value: "PROVIDE_DATA PATH:"), new PatternProperty(DriverParameter.DATALOG_FILE_PATH),
            new PatternCondition(DriverParameter.PARAMETRIC_DATA_ID).isDefined(
                    new PatternConstant( value: " PARAMETRIC_DATA_ID:"),
                    new PatternProperty(DriverParameter.PARAMETRIC_DATA_ID)),
            new PatternConstant( value: "\r\n").writeOnly()));

    private DataPattern commandPattern;

    TesterDaemonProviderCommand(DataPattern commandPattern) { this.commandPattern = commandPattern; }

    @Override
    public DataPattern getCommandPattern() { return commandPattern; }

    @Override
    public DataPattern getResponsePattern() { return null; }
}
```

Obrázek 1.11: *Implementace Protokolu bez stavového stroje*

Jakmile máme protokol definován je potřeba vytvořit jeho převodník do transportního formátu. k tomuto účelu naimplementujeme třídu ProtocolHandleru. Jedná se o generické rozhraní jehož typy jsou transportní, vstupní a výstupní formát. Protože chceme použít RemoteCommandExecutor pro provádění příkazů je potřeba použít jako vstup a výstup třídy definované ve frameworku.

Samotný ProtocolHandler má dvě sady metod jednu pro překlad požadavků a druhou pro překlad odpovědí. v tomto případě je ProtokolHandler používán oběma aplikacemi, proto je potřeba implementovat všechny čtyři metody. Pokud je ProtocolHandler využíván jen jednou stranou stačí implementace používaných metod.

```
public class TesterDaemonProviderProtocolHandler
        implements ProtocolHandler<String, CommonRemoteRequest, CommonRemoteResponse> {

    private static final Logger log = LoggerFactory.getLogger(TesterDaemonProviderProtocolHandler.class);

    private String ACK_MESSAGE = "ACK";
    private String ERROR_MESSAGE = "ERROR";
    private String ERROR_MESSAGE_SEPARATOR = ":";

    @Override
    public CommonRemoteRequest decodeRequest(String inbound) {
        log.info("Decoding request: {}", inbound);
        CommonRemoteRequest result = null;
        for (TesterDaemonProviderCommand command : TesterDaemonProviderCommand.values()) {
            if (inbound.startsWith(command.name())) {

                Parameters requestParameters = Patterns.getProperties(inbound.trim(), command.getCommandPattern());
                result = new CommonRemoteRequest(command, requestParameters);
                break;
            }
        }
        return result;
    }

    @Override
    public String encodeRequest(CommonRemoteRequest outbound) {
        log.info("Encoding request: {}", outbound);
        String result = Patterns.setPropertiesInPattern(outbound.getCommand().getCommandPattern(),
                outbound.getParameters(), ignoreMissingProperties: false);
        log.info("Request encoded as: {}", result);
        return result;
    }
```

Obrázek 1.12: *Příklad Protokol Handleru pro protokol bez stavového stroje*

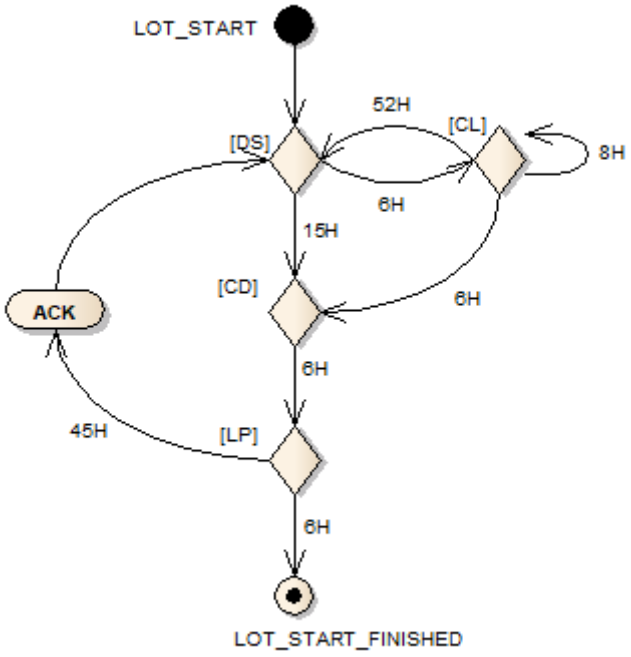
V ukázce můžete vidět způsob jakým jsou využité vzory v rámci převodu mezi formáty. v případně tohoto protokolu jsou jako odpovědi využité jednoduché potvrzovací zprávy definované na začátku třídy, v případě složitějších odpovědí bychom postupovali stejně jak při převodu příkazů.

4.3 Implementace "complex" protokolu

Complex protokol představuje libovolný protokol, který je možné implementovat stavovým strojem. Pro realizaci budeme potřebovat:

- tester-daemon-model : DataPattern pro definici vzorů příkazů a odpovědí
- tester-daemon-model : AbstractComplexProtocolHandler je předpřipravený protokol handler který umí pracovat se stavovým strojem
- tester-daemon-model : ComplexProtocolHandler je rozhraní pro implementaci překladu příkazů a odpovědí z/do transportního formátu.¨
- tester-daemon-model : ComplexProtocolHandlerState je rozhraní pro definici stavů a přechodů stavového stroje

Stejně jako v případě protokolu bez stavového stroje je potřeba začít definicí protokolu. Zde je vhodné začít definicí jednotlivých příkazů a vytvořit si pro každý z nich nákres stavového stroje, který jej reprezentuje. Poté můžeme přistoupit k popisu samotného stavového stroje.



Obrázek 1.13: *Ukázka stavového stroje pro specifický příkaz*

Stejně jako v předchozím případě pro popis stavů a přechodů využijeme výčtový datový typ díky jednoduché práci s ním. Je ovšem potřeba obejít limitaci jazyku Java který pro překlad výčtů používá jen jeden průchod, pokud tedy chceme realizovat křížové odkazy je nutné rozdělit deklaraci a definici výčtu.

```
public enum RgHostLinkCommandState implements ComplexProtocolCommandState {

    // SHARED STATES
    RCV_FINAL_ACK,
    RCV_FINAL_EC,

    // HANDSHAKE
    HANDSHAKE_SEND_COMMAND,
    HANDSHAKE_WAIT_FOR_RESPONSE,

    // LOAD TEST PROGRAM
    LTP_DS_SEND_COMMAND,
    LTP_DS_WAIT_FOR_RESPONSE,
    LTP_CL_SEND_COMMAND,
    LTP_CL_WAIT_FOR_RESPONSE,
    LTP_CD_SEND_COMMAND,
    LTP_CD_WAIT_FOR_RESPONSE,
    LTP_LP_SEND_COMMAND,
    LTP_LP_WAIT_FOR_RESPONSE,
```

Obrázek 1.14: *Deklarace stavů stroje pro protokol*

```
private static void initialize(RgHostLinkCommandState target, RgHostLinkCommandState defaultNextState,
                               DataPattern sendPattern) {
    target.nextStep = ComplexProtocolStepType.SEND_MESSAGE;
    target.defaultNextState = defaultNextState;
    target.sendPattern = sendPattern;
}

private static void initialize(RgHostLinkCommandState target, RgHostLinkCommandState defaultNextState,
                               Pair<HostLinkResponseCode, RgHostLinkCommandState>... transitions) {
    target.nextStep = ComplexProtocolStepType.WAIT_FOR_REPLY;
    target.defaultNextState = defaultNextState;
    for (Pair<HostLinkResponseCode, RgHostLinkCommandState> transition : transitions) {
        target.transitions.add(new Pair<String, ComplexProtocolCommandState>(
                transition.getKey().getCode(), transition.getValue()));
    }
}

private static void initialize(RgHostLinkCommandState target, RgHostLinkCommandState defaultNextState,
                               RgHostLinkCommandState timeoutNextState,
                               Pair<HostLinkResponseCode, RgHostLinkCommandState>... transitions) {
    target.nextStep = ComplexProtocolStepType.WAIT_FOR_REPLY;
    target.defaultNextState = defaultNextState;
    target.timeoutNextState = timeoutNextState;
    for (Pair<HostLinkResponseCode, RgHostLinkCommandState> transition : transitions) {
        target.transitions.add(new Pair<String, ComplexProtocolCommandState>(
                transition.getKey().getCode(), transition.getValue()));
    }
}
```

Obrázek 1.15: *Metody pro snadnou definici přechodů stroje*

```
static {
    initialize(RCV_FINAL_ACK, ComplexProtocolStepType.COLLECT_RESULT);
    initialize(RCV_FINAL_EC, ComplexProtocolStepType.COLLECT_RESULT);

    initialize(HANDSHAKE_SEND_COMMAND, HANDSHAKE_WAIT_FOR_RESPONSE,
            new DataPattern(new PatternConstant( value: "[" + String.valueOf('\u0005') + "]")));
    initialize(HANDSHAKE_WAIT_FOR_RESPONSE, ComplexProtocolStepType.WAIT_FOR_REPLY); // custom implementation

    //LOAD TEST PROGRAM
    initialize(LTP_DS_SEND_COMMAND, LTP_DS_WAIT_FOR_RESPONSE,
            new DataPattern(new PatternConstant( value: "[DS]")));
    initialize(LTP_DS_WAIT_FOR_RESPONSE, RCV_FINAL_EC,
            new Pair<>(HostLinkResponseCode.NAK, LTP_CL_SEND_COMMAND),
            new Pair<>(HostLinkResponseCode.ACK, LTP_CL_SEND_COMMAND));

    initialize(LTP_CL_SEND_COMMAND, LTP_CL_WAIT_FOR_RESPONSE,
            new DataPattern(new PatternConstant( value: "[CL]")));
    initialize(LTP_CL_WAIT_FOR_RESPONSE, RCV_FINAL_EC,
            new Pair<>(HostLinkResponseCode.NAK, LTP_CD_SEND_COMMAND),
            new Pair<>(HostLinkResponseCode.ACK, LTP_CD_SEND_COMMAND),
            new Pair<>(HostLinkResponseCode.EC_NOT_LOGGING, LTP_CD_SEND_COMMAND));
```

Obrázek 1.16: *Definice přechodů stavového stroje*

Definice přechodů je prováděna vždy pro jeden stav. Pomineme-li terminální stavy, jedná se prakticky o stavy reprezentující odesílání, nebo příjem dat. v případně odesílání je stav inicializován pomocí cílového stavu a dat která se mají poslat. v případně příjmu dat je možné popsat cílové stavy pomocí očekávaných odpovědí, případně bude v dalším kroku možné definovat přechod v rámci kódu protokol handleru. Tuto možnost jsem přidal, aby bylo možné pře použít části stavového stroje, které jsou společné, v příkladu je to Handshake, který se provádí pokaždé. Případně pro případ že by popis přechodů byl nedostatečný. v praxi zatím aplikujeme pouze první případ.

V ukázce implementace protocol handleru je demonstrována možnost realizovat přechod mezi stavy v rámci samotného protokol handleru. v tomto případě se jedná o pře použití stavové stroje pro Handshake a následně protokol handler pokračuje stavovým strojem pro samotný příkaz.

U této implementace je potřeba myslet na fakt že je třeba pro každý stav implementovat pouze metodu odpovídající prováděné akci. Mali daný stav posílat data stačí naimplementovat metodu pro získání odesílané zprávy.

```
public class RgHostLinkComplexProtocolHandler implements
        ComplexProtocolHandler<String, RgHostLinkCommand, CommonRemoteResponse> {

    protected RgHostLinkCommand command;
    protected Parameters parameters;
    protected ComplexProtocolCommandState currentState;
    protected String errorMessage;

    public RgHostLinkComplexProtocolHandler(RgHostLinkCommand command, Parameters parameters) {...}

    @Override
    public RgHostLinkCommand getCommand() { return command; }

    @Override
    public ComplexProtocolStepType getNextStep() { return currentState.getNextStep(); }

    @Override
    public String getNextMessage() {

        String result = null;
        ComplexProtocolCommandState oldState = currentState;

        if (currentState.getSendPattern() != null) {
            log.debug("Data pattern defined in state: {}", currentState.getSendPattern());
            result = Patterns.setPropertiesInPattern(currentState.getSendPattern(),
                    parameters, ignoreMissingProperties: false);
            currentState = currentState.getDefaultNextState();
        } else {
            log.error("State: {} don't perform step: {}", currentState, ComplexProtocolStepType.SEND_MESSAGE);
            throw new IllegalStateException("State: " + currentState
                    + " don't perform step: " + ComplexProtocolStepType.SEND_MESSAGE);
        }

        log.info("Protocol Handler state changed from: {} to: {} with next message: {}",
                oldState, currentState, result);
        return result;
    }
```

Obrázek 1.17: *Complex protocol handler příprava dat pro odeslání*

```
@Override
public void processReply(String reply) {
    log.info("Processing HostLink reply: {}", reply);
    ComplexProtocolCommandState oldState = currentState;

    HostLinkResponseCode responseCode = HostLinkResponseCode.parse(reply);
    if (!currentState.getNextStateTransitions().isEmpty()) {
        processHostLinkResponseCode(responseCode, currentState.getDefaultNextState(),
                currentState.getNextStateTransitions());
    } else {
        if (RgHostLinkCommandState.HANDSHAKE_WAIT_FOR_RESPONSE.equals(currentState)) {
            switch (responseCode) {
                case ACK:
                    switch (command) {
                        case CHECK_CONNECTION:
                            currentState = RgHostLinkCommandState.RCV_FINAL_ACK;
                            break;
                        case LOAD_TEST_PROGRAM:
                            currentState = RgHostLinkCommandState.LTP_DS_SEND_COMMAND;
                            break;
```

Obrázek 1.18: *Complex protocol handler zpracování odpovědi*

4.4 Vytvoření projektu pro Tester Driver

Pro potřeby jednoduché práce s frameworkem doporučuji používat Maven(3) pro který je v rámci projektu připravené vše potřebné pro vytvoření spustitelné aplikace. Na straně Tester Driveru je třeba definovat v minimální implementaci dva soubory. Samotnou třídu driveru která bude implementovat potřebný výběr příkazů z Tester Driver rozhraní a definici pro sestavení aplikace v IoC.

4.4.1 Nastavení životního cyklu

Pro sestavení aplikace je použitý Maven který se stará primárně o sestavení aplikace a správu závislostí, definice projektu a jeho životního cyklu je v souboru pom.xml. Jednou z výhod Mavenu je možnost definovat proces sestavení hierarchicky. Díky tomu je pro Tester Drivery připravena obecný popis životního cyklu a sestavení který je součástí projektu "tester-daemon-parent". Ten obsahuje primárně dodatečné kroky sestavení aplikace, které nejsou zahrnuty z defaultním životním cyklu Mavenu(4).

Je zde využit "maven-assembly-plugin" pro sestavení spustitelné aplikace. Podklady pro sestavení jsou součástí projektu "tester-daemon" který představuje část frameworku určenou pro testery. Díky tomu jsou vždy aktuální pro danou verzi a existují vždy pouze v jedné kopii. Aby je bylo možné použít bylo potřeba doplnit životní cyklus aplikace o "maven-dependency-plugin" který zpřístupní potřebné soubory během sestavení. Samotné sestavení má tři varianty podle toho kolik instancí se má najednou vytvořit. Obvyklý scénář je jeden tester driver na jeden tester, existují ale i případy kdy tester fyzicky obsahuje více separátních jednotek každá z nich pak má svou vlastní instanci driveru.

Druhým rozšířením životního cyklu je kontrola kompatibility s cílovou verzí jazyka Java. Tato dodatečná kontrola byla přidána od frameworku dodatečně protože jsem předpokládal, že nastavením správné verze zdrojových kódu a překladu ošetřím případnou nekompatibilitu s verzí Javy která je dostupná na počítači, kam bude driver nasazen. Dlouho tato kontrola fungovala dobře a nebyl žádný problém s kompatibilitou. Při vývoji jednoho driveru se během testování stávalo že aplikace spadla bez zjevného důvodu. Po bližší analýze problému jsem zjistil že Java má podstatný nedostatek při kontrole kompatibility. Pokud ve zdrojovém kódu používáte funkcionalitu dodanou samotnou Javou je při kompilaci provedena kontrola, zda metoda existuje vůči verzi kterou jste použili pro kompilaci a ne oproti verzi pro kterou má být vaše výsledná aplikace. Prakticky tam můžete vytvořit aplikaci která bude například pro Javu 6 ale bude volat metodu která byla do Javy přidána až později. Jedná se o velice specifickou chybu přesto díky tomu že na projektu pracuje více lidí a podporovaná verze Javy se pohybuje aktuálně mezi Java 5 až Java 8. Bylo potřeba přidat kontrolu která by zajistila kompatibilitu nezávisle na vývojovém prostředí.

Pro tuto potřebu existuje pro Maven Animal Sniffer Maven Plugin(5) jehož součástí jsou signatury všech verzi Javy, dokonce i pro upravené verze Javy vytvořené firmami jako Apple, IBM, ap.

4.4.2 Spring IoC

S výhodami aplikací založených na využití principu "IoC - Inversion of Control" jsem se setkal poprvé až v praxi. Pro malé projekty nemá smysl přidávat další komplexitu. Pokud ale dodržujete principy OOP tak se u komplexních aplikací jedná o neocenitelného pomocníka.

Princip IoC je sestavení aplikace až při spuštění z jednotlivých komponent dle poskytnuté definice. Díky tomu nemusíte ve zdrojovém kódu provádět konstrukci aplikace, která jej akorát znepřehlední. Díky tomu je možné jednoduše provádět změny v rámci aplikace bez zásahu do zdrojového kódu.

Pro samotné sestavení je potřeba definovat strukturu aplikace v rámci firmy používáme řešení IoC poskytované frameworkem Spring(6). Framework na něm není závislý, ale jeho použití výrazně zjednodušuje.

Spring poskytuje dvě možnosti jak aplikaci sestavit. První je možnost využít anotací v kódu, toto řešení je vhodné pokud aplikace nepotřebuje časté změny. Druhou variantou kterou používáme ve firmě je definice pomocí Aplikačního Kontextu v souboru xml.

Princip sestavení spočívá nejprve ve vytvoření instancí definovaných komponent, následně jsou nalezeny v kódech těchto komponent všechny spojení, která mají být provedena a IoC následně propojí komponenty na základě typů tříd, nebo podle definovaných identifikátorů.

4.4.3 Implementace příkazů pro tester

Pro implementaci příkazů Tester Driveru je potřeba pouze vytvořit třídu, která dědí z "AbstactDriver" třídy. Toto je jeden z mála případů dědičnosti, která je v rámci projektu použita a to kvůli nemožnosti definovat v jazyce Java defaultní implementaci metod rozhraní do verze 8. Díky využití abstraktní třídy stačí naimplementovat pouze příkazy, které jsou potřeba pro daný tester.

```
public class Ets200Driver extends AbstractDriver {

    private static final Logger log = LoggerFactory.getLogger(Ets200Driver.class);

    private LotRuntimeContext lotRuntimeContext;

    @Override
    public void initialize() {
        log.info("Initializing ETS200 driver ...");
        super.initialize();
        log.info("Initializing completed ...");
    }

    public LoadFromXml getRuntimeContext() {
        if (lotRuntimeContext == null) {
            lotRuntimeContext = LotRuntimeContext.newContext();
        }
        return lotRuntimeContext;
    }
```

Obrázek 1.19: *Inicializace implementace Tester Driveru*

Tato úvodní část bude stejná pro všechny Tester Drivery. Obsahuje deklaraci kontextových dat pro driver. Díky metodě "getRuntimeContext" k nim dokáže přistoupit jádro a uložit, nebo obnovit uvnitř uložené hodnoty dle potřeb implementace persistence. Tento kontext je také použit jako zdroj defaultních hodnot pro příkazy takže je potřeba k němu přistupovat přímo pouze pro zápis. v rámci inicializační metody je třeba zavolat inicializaci s AbstractDriveru, která spustí online monitoring komponent využívaných Tester Driverem, těmi můžou být aplikace nebo tester samotný.

Následně implementujeme příkazy, které jsou potřeba dle zadání. Za minimální implementaci se považuje sada příkazů LotStart, LotEnd a Cancel.

```
@Override
public Parameters lotStart(Parameters p, TesterDriverStatusListener statusListener) throws InterruptedException {
    log.info("Lot start in process with parameters: {} and lotRunTimeContext: {}.", p, lotRuntimeContext);
    lotRuntimeContext = LotRuntimeContext.newContext(p);

    if(p.getFirst(DriverParameter.LOAD_TEST_PROGRAM, mandatory: true, Boolean.class)) {
        if(WindowsProcesses.getProcessMetadataByProcessName(
                p.getFirst(DriverParameter.VENDOR_SOFT_PROCESS_NAME, mandatory: true, String.class)) != null) {
            log.error("Vendor soft shell is running. Terminate it before Lot Start");
            throw DaemonExceptions.getTesterVendorSoftRunningException("Terminate it before Lot Start");
        }

        Long requiredDiskSpaceKB = p.getFirst(DriverParameter.FREE_DISK_SPACE_REQUIRED_KB,
                mandatory: false, Long.class);
        if(requiredDiskSpaceKB != null) {
            long freeDiskSpace = FileHandler.getFreeDiskSpaceKB(
                    p.getFirst(DriverParameter.VENDOR_SOFT_EXECUTABLE_PATH, mandatory: true, String.class));
            if(requiredDiskSpaceKB >= freeDiskSpace) {
                log.error("There is not enough free space on tester. " +
                        "Required disk space: {}KB, actual disk space: {}KB", requiredDiskSpaceKB, freeDiskSpace);
                throw DaemonExceptions.getTesterValidationsFailedException("FREE_DISK_SPACE_VALIDATION",
                        "Required disk space: "+requiredDiskSpaceKB
                                +"KB is more than actual disk space: "+freeDiskSpace+"KB");
            }
        }

        String testProgramPath = FileHandler.composePath(
                p.getFirst(DriverParameter.TEST_PROGRAM_REPOSITORY_ROOT_PATH, mandatory: true, String.class),
                p.getFirst(DriverParameter.TEST_PROGRAM_PATH, mandatory: true, String.class));
        String testProgramNameWithExtension = p.getFirst(DriverParameter.TEST_PROGRAM_NAME, mandatory: true, String.class)
                + "." + p.getFirst(DriverParameter.TEST_PROGRAM_EXTENSION, mandatory: true, String.class);
        List<File> testProgram = FileHandler.getEmptyListNotNull(
                FileHandler.getFilesByNameWithExtension(testProgramPath, testProgramNameWithExtension));
        if(testProgram.isEmpty()) {
            log.error("Test program: {} is not present in defined directory: {}",
                    testProgramPath, testProgramNameWithExtension);
            throw DaemonExceptions.getFileNotFoundByName(
                    testProgramPath, testProgramNameWithExtension, fileType: "TEST_PROGRAM");
        }
```

Obrázek 1.20: *Ukázka implementace příkazu Lot Start - část 1.*

V první části příkazu LotStart jsou prováděny následující kroky:

- Aktualizace Lot kontextu parametry příkazu. Tento krok je potřeba, aby je mohl Tester Driver použít v následných příkazech. Tento krok je prováděn ze dvou důvodů. Není třeba posílat v dalších příkazech parametry, které jsou v rámci kontextu stále platné. Druhým důvodem je poté jistota, že parametr nebude během své platnosti změněn. To je důležité zvláště pro validace.
- Další kroky jsou volitelné, pokud se má Tester Driver načíst testovací program na tester. Tato funkce je volitelná, protože pro neprodukční použití může být testovací program nahrán ručně. Příkladem může být jeho vývoj.
- Kontrola zda běží aplikace testeru, která slouží pro jeho řízení a jako uživatelské rozhraní pro manuální práci s testerem. v rámci zadání bylo specifikováno, že pokud tomu tak je signalizuje to chybu, nebo rozpracovanou práci v manuálním módu. Pro oba případy má příkaz vrátit chybu, kterou musí operátor výroby odstranit, než bude moc pokračovat.
- Kontrola volného místa na disku, aby byl zajištěn dostatek prostoru pro data vzniklá při měření.
- Kontrola zda testovací program existuje, případně že je dostupný z daného testeru. Test programy se nacházejí v centrálním repositáři který je z testeru přístupný pro čtení.

```
    WindowsProcesses.startProcess(
            FileHandler.composePath(
                    p.getFirst(DriverParameter.VENDOR_SOFT_EXECUTABLE_PATH,  mandatory: true, String.class),
                    p.getFirst(DriverParameter.VENDOR_SOFT_EXECUTABLE_NAME,  mandatory: true, String.class)),
             ...params: p.getFirst(DriverParameter.TEST_PROGRAM_NAME,  mandatory: true, String.class),
            "-at", "-b", "-x0");


    boolean vendorSoftStarted = false;
    for (int i = 0; i < 24; i++) {
        Thread.sleep( millis: 5000);
        SystemProcessMetadata vendorSoftProcessParameters = WindowsProcesses.getProcessMetadataByProcessName(
                p.getFirst(DriverParameter.VENDOR_SOFT_PROCESS_NAME,  mandatory: true, String.class));
        SystemProcessMetadata vendorSoftExecutiveParameters = WindowsProcesses.getProcessMetadataByProcessName(
                p.getFirst(DriverParameter.VENDOR_SOFT_PROCESS_EXECUTIVE_NAME,  mandatory: true, String.class));

        if (vendorSoftProcessParameters != null && vendorSoftExecutiveParameters != null) {
            vendorSoftStarted = true;
            break;
        }

    }

    if(!vendorSoftStarted) {
        log.error("Vendor soft didn't start with following parameters: {}", p);
        throw DaemonExceptions.getTesterVendorSoftNotRunningException(
                "Vendor soft didn't start correctly, please check vendor soft for errors");
    }
}

log.info("Lot start completed with response: {}.", p);
return p;
}
```

Obrázek 1.21: *Ukázka implementace příkazu Lot Start - část 2.*

V druhé části příkazu LotStart jsou prováděné následující kroky:

- Načtení testovacího programu skrze aplikaci testeru v automatizovaném módu. Tento krok je zcela závislý na možnostech daného testeru a vychází z jeho dokumentace.
- Kontrola zda během načítání nedošlo k chybě, to je pro daný tester dáno tím zdali se spustí oba procesy aplikace testeru.

Kroky demonstrované v této části jsou mnohem častěji realizované skrze protokol poskytnutý výrobcem testeru. Jeho implementace byla nastíněná v kapitolách 4.2 a 4.3.

4.4.4 Tvorba Aplikačního Kontextu pro IoC

Aplikační kontext Spring IoC driveru definuje všechny části aplikace, aby byl tento proces zjednodušen, obsahuje framework předpřipravené nejpoužívanější fragmenty a ty je možné využít v rámci kontextu. Dále je třeba definovat komponenty unikátní pro daný tester. Jeden z nejjednodušších možných kontextů je na následujícím obrázku.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"

       xsi:schemaLocation="http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
          http://www.springframework.org/schema/context
          http://www.springframework.org/schema/context/spring-context-3.0.xsd
">
    <context:annotation-config/>

    <import resource="classpath:TesterDaemonCoreApplicationContext.xml"/>
    <import resource="classpath:DataCollectorProviderClientApplicationContext.xml"/>
    <import resource="classpath:DataCollectorControlClientApplicationContext.xml"/>
    <import resource="classpath:DataCollectorOnlineMonitoringApplicationContext.xml"/>

    <!-- Driver -->
    <bean id="driver" class="com.onsemi.cim.apps.testerdaemon.testerdriver.Ets200Driver"/>
</beans>
```

Obrázek 1.22: *Ukázka Aplikačního kontextu pro jednoduchý driver*

Na tomto příkladu bych rád ukázal jak snaha předpřipravit v rámci frameworku co nejvíce z výsledné funkcionality. Použité importy v kontextu redukovaly kontext z původních přibližně čtyř set řádků na méně než deset. Navíc díky použití ověřeného importu nemůže dojít ke konfigurační chybě.

První import obsahuje nastavení jádra, včetně konektoru pro příjem příkazů. Následující importy fungují jako spuštění dodatečné funkcionality, která by pokud by nebyly vloženy byla vypnutá. Jedná se o poskytovatele výsledků měření, externí signalizaci pro sběr dat a zahrnutí Data Kolektoru (aplikace pro sběr dat) do Online statusu driveru.

Následuje definice komponent potřebných pro driver samotný. v tomto případě používá driver pouze funkcí poskytnutých přímo frameworkem, včetně konfigurace. Proto není potřeba žádné další nastavení.

Další příklad demonstruje již driver který komunikuje jak s testerem tak handlerem a má konfigurační položky specifické pro driver.

```
<bean id="testerConnector" class=
"com.onsemi.cim.apps.testerdaemon.connectivity.connector.adapter.conversion.StringToBytesConversionConnectorAdapter">
    <constructor-arg value="UTF-8"/>
    <property name="instanceName" value="ISO"/>
    <property name="sourceConnector" ref="testerConnectorByte"/>
</bean>

<bean id="testerConnectorByte" class=
        "com.onsemi.cim.apps.testerdaemon.connectivity.connector.asynchronous.serialline.SerialLineConnector">
    <property name="instanceName" value="ISO"/>
    <property name="serialPortName" value="${tester.com.port}"/>
    <property name="baudrate" value="9600"/>
    <property name="databits" value="8"/>
    <property name="stopbits" value="1"/>
    <property name="parity" value="0"/>
    <property name="flowControl" value=
            "#{T(jssc.SerialPort).FLOWCONTROL_XONXOFF_IN + T(jssc.SerialPort).FLOWCONTROL_XONXOFF_OUT}"/>
</bean>


<!-- Handler Connection -->
<bean id="handlerConnector" class=
        "com.onsemi.cim.apps.testerdaemon.connectivity.connector.synchronous.PassiveWaitingSynchronousConnector">
    <constructor-arg value="java.lang.String"/>
    <property name="instanceName" value="OSAI"/>
    <property name="asyncConnector" ref="handlerConnectorWrapper"/>
    <property name="waitTimeoutMilisec" value="10000"/>
</bean>

<bean id="handlerConnectorWrapper" class=
"com.onsemi.cim.apps.testerdaemon.connectivity.connector.adapter.conversion.StringToBytesConversionConnectorAdapter">
    <constructor-arg value="UTF-8"/>
    <property name="instanceName" value="OSAI"/>
    <property name="sourceConnector" ref="asyncHandlerConnector"/>
</bean>

<bean id="asyncHandlerConnector" class=
        "com.onsemi.cim.apps.testerdaemon.connectivity.connector.asynchronous.tcpip.TcpIpServerConnector">
    <property name="instanceName" value="OSAI"/>
    <property name="maximalNoClients" value="2"/>
    <property name="portNumber" value="${handler.port}"/>
    <property name="inputMessageReader" ref="tdBufferMessageReader"/>
    <property name="outputMessageWriter" ref="tdBufferMessageWriter"/>
</bean>
```

Obrázek 1.23: *Ukázka konfigurace konektorů pro Sériovou linku a TCP/IP*

Na obrázku jsou ukázané dva konektory používané v rámci driveru. v prvním případě byl mimo samotného konektoru, použit pouze adaptér pro transformaci textových řetězců na pole bytů, zbytek funkcionality komunikace je v rámci driveru. Jedná se o starší driver, realizovaný v době před "complex" protokoly který nebyl zatím nebyl aktualizován pro vyšší integraci s frameworkem. Druhým spojením je TCP/IP které využívá i synchronní konektor který využívá adapter pro převod textových dat a asynchronní TCP/IP konektor v roli serveru.

```
<!-- Implementation of tester driver -->
<bean id="testerDriver" class="com.onsemi.cim.apps.testerdaemon.testerdriver.IsoDriver"/>

<bean id="driverConfiguration"
      class="com.onsemi.cim.apps.testerdaemon.testerdriver.model.configuration.DriverConfiguration">
    <property name="platform" value="ISO"/>
    <property name="stepName" value="${step.name}"/> <!-- old identifier -->

    <property name="testProgramUploadRetryLimit" value="5"/>
    <property name="testProgramExtension" value="TST"/>

    <property name="parametricDataFileExtension" value="stdf"/>

    <property name="testerDaemonTempPath" value="${driver.test.program.local.dir-path}"/>

    <property name="skipOsaiCommands" value="${skip.osai.commands}"/>
</bean>
```

Obrázek 1.24: *Driver s konfigurací*

Na obrázku je ukázán driver s dodatečnou konfigurací. Konfigurace je použita jako jeden ze zdrojů defaultních hodnot pro prováděné příkazy.

Kompletní příklady jsou součástí přiložených zdrojových kódů.

Závěr

Během práce na diplomové práci jsem docílil značného zjednodušení implementace aplikací pro automatizaci testerů v produkci. Docílil jsem toho analýzou existujících řešení. Na jejich základně jsem dále postupně vytvářel jednotlivé komponenty, které ve výsledku vytvořily celý framework.

Když si dnes vzpomenu, že celá práce začala mým odmítnutím překopírovat fragment zdrojového kódu mezi dvěma aplikacemi a namísto toho jsem se zamyslel, zdali by nešlo zdrojový kód pře použít ze sdílené knihovny. Ušel jsem dlouhou cestu vývojem, na které jsem se sám mnohé naučil. Kdybych měl vypíchnout jedinou věc, kterou jsem se naučil, byla by to schopnost mnohem lépe analyzovat problémy a určit takový rozsah implementace, která je dostatečná pro danou chvíli, ale současně neomezuje budoucí rozšiřitelnost. Díky těmto zkušenostem a jejich aplikací do praxe se z frameworku nestal projekt do šuplíku, ale firemní standard pro automatizaci testerů ve výrobě.

Proto má práce na frameworku určitě nekončí. v posledních měsících už přerostly schopnosti frameworku rozsah původních automatizačních řešení. Společně s mým nadřízeným jsme přišli s následujícími rozšířeními. Nejbližším plánem pro rozvoj je přidání možnosti znovu načíst konfiguraci aplikace za běhu. Tato funkce odstraní potřebu restartu při změně konfigurace a současně umožní dosáhnout prvního velkého cíle. Tímto cílem je možnost online aktualizací. Počet strojů využívajících Tester Drivery se neustále rozšiřuje a proto chceme poskytnout jednoduchý nástroj jak aktualizovat vybrané testery z pohodlí kanceláře. s tím je spojený i vývoj dashboardu případně integrace s řešením třetí strany. Výraznou plánovanou změnou bude také rozšíření funkcionality konektorů o události a implementace protokolu SECS/GEM. SECS/GEM protokol je celosvětově uznávaným standardem a za možnost jej vůbec požít si výrobci zařízení nechávají dobře zaplatit. Jakmile budeme mít možnost jej použít tak zrychlíme vývoj pro stroje, které jej používají. Současně jím můžeme nahradit i protokol, který používají Tester Drivery nyní, čímž celé aplikaci skokově vzroste renomé.

Použitá literatura

[1] GAMMA, Erich. Design patterns: elements of reusable object-oriented software. Boston: Addison-Wesley, 1995. ISBN 0201633612.

[2] MCCONNELL, Steve. Code complete. 2nd ed. Washington: Microsoft Press, c2004. ISBN 978-0-7356-1967-8.

[3] Maven [online]. Apache [cit. 2020-05-14]. Dostupné z: https://maven.apache.org/

[4] Introduction to the Build Lifecycle. Https://maven.apache.org/ [online]. [cit. 2020-05-15]. Dostupné z: https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html

[5] Animal Sniffer Maven Plugin. MojoHaus [online]. [cit. 2020-05-15]. Dostupné z: https://www.mojohaus.org/animal-sniffer/animal-sniffer-maven-plugin/

[6] Spring [online]. [cit. 2020-05-15]. Dostupné z: https://spring.io/

Seznam příloh

Součástí DP jsou přílohy odevzdané v digitální formě. Adresářová struktura příloh:

- README.txt - návod na práci s přílohami
- Libs - složka s použitými knihovnami
- ukazky
  - ets200-tester-driver-1.1.0.local - driver pro www.teradyne.com/products/ets-200t/
    - dcfg-configuration-backup - složka obsahující zálohu konfigurace
    - ets200-tester-driver-1.1.0.local - složka obsahující samotný driver a jeho závislosti
    - configuration.properties - lokální konfigurační soubor pro driver
    - set-td-dc-env.bat - skript pro nastavení verze Javy která se má použít
    - start-td-only.bat - skript pro spuštění driveru
  - tester-daemon-6.2.0.local-simulator
    - configuration-backup - složka obsahující zálohu konfigurace
    - lib - složka se závislostmi simulátoru
    - resources - složka s nastavením simulátoru
    - run-simulator-and-reset-to-idle-shell.bat - skript který spustí simulátor bez persistence
    - run-simulator-shell.bat - skript který pustí simulátor se zapnutou persistencí
    - ukazka-volani-prikazu.txt - ukazky kominikace s driverem
- zdrojove-kody
  - ets200-tester-driver
    - src - složka se zdrojovými kódy projektu
    - pom.xml - konfigurační soubor Mavenu
  - on-global-parent
    - pom.xml - konfigurační soubor Mavenu
  - tester-daemon
    - src - složka se zdrojovými kódy projektu
    - pom.xml - konfigurační soubor Mavenu
  - tester-daemon-model
    - src - složka se zdrojovými kódy projektu
    - pom.xml - konfigurační soubor Mavenu
  - tester-daemon-parent
    - pom.xml - konfigurační soubor Mavenu