VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

AGILNÝ OBJEKTOVO-ORIENTOVANÝ VÝVOJ SOFTWARU V ABAP

AGILE OBJECT-ORIENTED SOFTWARE DEVELOPMENT IN ABAP

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — LUCIA BAGINOVÁ
AUTHOR

VEDOUCÍ PRÁCE — RNDr. MAREK RYCHLÝ, Ph.D.
SUPERVISOR

BRNO 2022

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2021/2022

Zadání bakalářské práce

Studentka: **Baginová Lucia**
Program: Informační technologie
Název: **Agilní objektově-orientovaný vývoj softwaru v ABAP**
Agile Object-Oriented Software Development in ABAP
Kategorie: Softwarové inženýrství
Zadání:

1. Seznamte se s IS SAP a s programovacím jazykem ABAP a jeho vývojovým prostředím. Seznamte se s principy agilního vývoje software, zejména s programováním řízeném testy (Test-Driven Development, TDD) a využitím refaktorizace kódu při vývoji.
2. Prostudujte a popište možnosti aplikování TDD vč. refaktorizace při vývoji objektově-orientovaného software v ABAP.
3. Po konzultaci s vedoucím zvolte vhodný praktický příklad pro SAP/ABAP a na tomto výše uvedený přístup demonstrujte, tj. software navrhněte a v několika iteracích implementujte s využitím TDD a refaktorizace.
4. Vyhodnoťte a diskutujte výsledky, zejména ve srovnání s neagilním a neobjektovým vývojem v ABAP.

Literatura:

- Bandari, K.: Complete ABAP. SAP Press, 2016. ISBN: 978-1-4932-1272-9.
- Zaidi R.: ABAP Unit Test-Driven Development. In: SAP ABAP Objects. Apress, Berkeley, CA, 2019. ISBN 978-1-4842-4964-2. Dostupné z: [https://doi.org/10.1007/978-1-4842-4964-2_7]
- McDonough J.E.: Test-Driven Development. In: Automated Unit Testing with ABAP. Apress, Berkeley, CA, 2021. ISBN 978-1-4842-6951-0. Dostupné z: [https://doi.org/10.1007/978-1-4842-6951-0_11]
- Beck, K.: Programování řízené testy. Grada, Praha, 2004. ISBN 80-247-0901-5

Pro udělení zápočtu za první semestr je požadováno:

- Body 1, 2 a rozpracovaný bod 3.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Rychlý Marek, RNDr., Ph.D.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2021
Datum odevzdání: 11. května 2022
Datum schválení: 11. října 2021

Abstrakt

Táto práca predstavuje agilné metodiky vývoja softwaru, ich vlastnosti a aj využitie refaktorizácie pri vývoji. Popisuje tiež nástroje, ktoré sú v spoločnosti SAP využívané pre vývoj softwaru v ich vlastnom programovacom jazyku ABAP. Cieľom práce je s využitím agilných prístupov implementovať objektovo-orientovaný program v jazyku ABAP. Program umožňujúci manuálne zadanie určitých hodnôt výkazu DPH je implementovaný s využitím refaktorizácie, testovaný a dodávaný zákazníkom spoločnosti SAP. Metodikou *Test Driven Development* je implementované rozšírenie, ktoré umožňuje export zadaných hodnôt vo formáte CSV.

Abstract

This thesis introduces agile software development, characteristics of the methodologies, and usage of refactoring as part of the development. Various tools used by SAP for developing software in their proprietary programming language ABAP are presented. The thesis aims to implement an object-oriented program in ABAP using agile principles. A program that enables users to manually enter specific values of VAT report is implemented using refactoring, tested, and delivered to SAP customers. Additional functionality that allows exporting the values to CSV format is implemented using *Test Driven Development*.

Kľúčové slová

agilné metodiky, refaktorizácia, vývoj riadený testami, jednotkové testovanie v jazyku ABAP, SAP, ABAP report, dynpro

Keywords

agile metodologies, refactoring, Test Driven Development, unit testing in ABAP, SAP, ABAP report, dynpro

Citácia

BAGINOVÁ, Lucia. *Agilný objektovo-orientovaný vývoj softwaru v ABAP*. Brno, 2022. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce RNDr. Marek Rychlý, Ph.D.

Agilný objektovo-orientovaný vývoj softwaru v ABAP

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracovala samostatne pod vedením pána RNDr. Marka Rychlého, Ph.D. Uviedla som všetky literárne pramene, publikácie a ďalšie zdroje, z ktorých som čerpala.

. . . .
Lucia Baginová
11. mája

Poďakovanie

Rada by som poďakovala RNDr. Marku Rychlému, Ph.D. za vedenie tejto práce, poskytnuté rady a spätnú väzbu. Poďakovať chcem aj svojim kolegom Mgr. Stanislavovi Šuhájkovi a Mgr. Marii Żyśko, ktorí boli ochotní mi kedykoľvek poradiť a zdieľať svoje skúsenosti. Veľká vďaka patrí tiež Mgr. Petrovi Matouškovi za poskytnuté poradenstvo v období písania tejto práce.

Obsah

1 Úvod **3**

2 Agilné metodiky vývoja softwaru a využitie refaktorizácie **4**
2.1 Princípy a vlastnosti agilných metodík . . . 4
2.2 Extreme programming . . . 6
2.3 SCRUM . . . 7
2.4 Test Driven Development . . . 9
2.5 Ďalšie agilné metodiky . . . 10
2.6 Refaktorizácia ako súčasť agilného vývoja . . . 11
2.6.1 Čo refaktorizácia prináša . . . 11
2.6.2 Ako refaktorovať . . . 11
2.6.3 Kedy je refaktorizácia (ne)vhodná . . . 12

3 Vývoj softwaru a jednotkové testovanie v jazyku ABAP **13**
3.1 Nástroje používané pre vývoj v SAP . . . 13
3.2 Interakcia s užívateľom pomocou dynpra . . . 14
3.3 Jednotkové testovanie v jazyku ABAP . . . 16
3.3.1 Umožnenie testovateľnosti odstránením závislostí v programe . . . 16
3.3.2 Vytváranie a implementácia testovacích tried . . . 17
3.3.3 Spúšťanie testov, reprezentácia výsledkov . . . 20

4 ABAP report pre manuálne zadávanie hodnôt výkazu DPH **22**
4.1 Návrh obrazoviek a tried . . . 23
4.2 Implementácia ABAP reportu . . . 24
4.2.1 Spôsob uloženia dát v databáze . . . 25
4.2.2 Triedy zabezpečujúce zobrazenie dát na dynpre . . . 26
4.2.3 Logika toku programu . . . 28
4.2.4 Priebeh implementácie s využitím refaktorizácie . . . 31
4.3 Priebeh testovania . . . 33
4.3.1 Unit testy . . . 33
4.3.2 Testy FIT a SAT . . . 34

5 Exportovanie tabuliek do formátu CSV **35**
5.1 Implementácia CSV exportu . . . 35
5.2 Test Driven Development a jeho výsledky . . . 37

6 Zhodnotenie vývoja s využitím objektovej orientácie a agilných prístupov **39**

7 Záver 41

Literatúra 42

A Diagramy tried 45

B Dynpro 100 ABAP reportu GLO_FIN_HU_VAT_DECL 48

C Dynpro 200 pre CSV export 51

D Obsah priloženého pamäťového média 52

Kapitola 1

Úvod

V snahe držať krok s rýchlo rozvíjajúcim sa trhom a neustálymi zmenami, ktoré dnešná doba prináša, je pre IT spoločnosti využitie agilných metodík dôležitejšie než kedykoľvek predtým. V porovnaní s tradičnými prístupmi poskytujú tieto metodiky vývojovým tímom väčšiu flexibilitu a umožňujú implementovať software rýchlejšie, jednoduchšie a v lepšej kvalite. Refaktorizácia pri vývoji zvyšuje schopnosť vyhovieť meniacim sa požiadavkám a teda umožňuje dodávať zákazníkom softwarové produkty, ktoré im prinášajú skutočnú hodnotu a pomáhajú ich spoločnostiam uspieť.

Cieľom práce je predstaviť najvýznamnejšie agilné metodiky a refaktorizáciu a s ich využitím implementovať objektovo-orientovaný sofware v jazyku ABAP. Pre spoločnosť SAP je implementovaná požiadavka, ktorá vychádza zo zmeny v maďarskej daňovej legislatíve, a ktorá vyžaduje do výkazu DPH zahrnúť nové položky, ktorých hodnoty nie je možné automaticky získať zo systému. V tejto práci je popísaný vývoj softwaru, ktorý umožňuje manuálne zadanie týchto hodnôt a funkcionalita programu je rozšírená o možnosť ich exportovania vo formáte CSV.

Kapitola 2 je zameraná na agilné metodiky a ich vlastnosti. Vysvetľuje dôvody k ich vzniku, aké benefity sú spojené s ich využívaním a tie najvýznamnejšie metodiky sú podrobnejšie predstavené. Samostatná podkapitola je venovaná refaktorizácii, jej prínosom a tiež kedy a ako ju aplikovať.

V kapitole 3 je predstavené vývojové prostredie spoločnosti SAP, jeho nástroje a tiež ich vlastný programovací jazyk ABAP spolu s jeho špecifickými prvkami. Jednotkovému testovaniu v jazyku ABAP, ktoré je kľúčové pri vývoji riadenom testami, je venovaná podkapitola vysvetľujúca nielen čo pri tomto type testovania dodržiavať, ale aj ako pracovať s *ABAP Unit Framework*, ktorý unit testovanie v danom jazyku umožňuje.

Programu v jazyku ABAP, ktorý umožní manuálne zadanie hodnôt výkazu DPH, je venovaná kapitola 4. Popisuje objektovo-orientovaný návrh programu, akým spôsobom je zabezpečené zobrazenie a editácia relevantných dát, ako sú jednotlivé triedy implementované s využitím refaktorizácie a akým spôsobom bola funkcionalita programu testovaná.

Kapitola 5 je venovaná rozšíreniu funkcionality o exportovanie zadaných hodnôt vo formáte CSV, ktoré je implementované metodikou *Test Driven Development*.

V kapitole 6 sú zhodnotené výhody objektovo-orientovaného modelu programovania v jazyku ABAP v porovnaní s procedurálnym a tiež využitie agilných prístupov pri vývoji SAP softwarov. Zhodnotený je vývoj implementovaného programu, čo využitie agilných princípov prinieslo a aké sú potenciálne možnosti na vylepšenie daného programu.

Kapitola 2

Agilné metodiky vývoja softwaru a využitie refaktorizácie

Agilné metodiky začali vznikať v dôsledku toho, že si viaceré významné kapacity v obore softwarového inžinierstva uvedomili, že celá rada klasických techník, ku ktorým radíme napríklad *vodopádový* alebo *špirálový model*, postupne prestávali svojou formálnosťou a robustnosťou vyhovovať požiadavkám. Tie sa rapídne zvýšili najme v ohľade na rýchlosť dodania softwarovej aplikácie a tiež flexibilite pri jej vývoji. Agilné metodiky odpovedajú práve tým, že za jedinú cestu, vedúcu k správnosti navrhovaného systému, považujú čo najrýchlejšiu implementáciu a následne postupné upravovanie na základe spätnej väzby od zákazníka [18].

V tejto kapitole bude popísaný historický vznik agilných metodík, akými princípmi sa riadia a akými vlastnosťami vynikajú. V následných podkapitolách budú špecifikovaní najvýznamnejší zástupcovia, medzi ktorých patria *Extrémne programovanie*, *metodika Scrum* a *Test Driven Development*, teda vývoj riadený testami, a stručne tiež ďalšie agilné metodiky. Ďalej sa táto kapitola venuje dôležitosti refaktorizácie pri agilnom vývoji, aké so sebou prináša výhody a ako ju praktikovať.

2.1 Princípy a vlastnosti agilných metodík

Spojenie slova agilný spolu s prístupom k tvorbe softwaru vzniklo v roku 2001, kedy sa v americkom štáte Utah stretlo 17 významných predstaviteľov nových prístupov k tvorbe softwaru. Medzi najvýznamnejšie postavy tzv. *Aliancie pre agilný vývoj* [4] patrili napríklad Kent Beck, zakladateľ konceptu extrémneho programovania [2], Martin Fowler, ktorý zaviedol termín refaktorovania [8], Alistar Cockburn, Ward Cunningham, Ken Schwaber a mnohí iní.

Podarilo sa im vtedy sformulovať *Manifest agilného vývoja softwaru*, ktorý vychádza z dvoch základných téz: [18]

1. *Prijať a umožniť zmenu je omnoho efektívnejšie, ako pokúšať sa jej zabrániť.*
2. *Je potrebné byť pripravený reagovať na nepredvídateľné udalosti, lebo tie bezpochyby nastanú.*

Na základe týchto téz dospeli tiež k formulácií štyroch hodnôt, tvoriacich myšlienkový základ agilného prístupu: [5]

- ***Jednotlivci a interakcia*** *pred procesmi a nástrojmi*
- ***Fungujúci software*** *pred vyčerpávajúcou dokumentáciou*
- ***Spolupráca so zákazníkom*** *pred vyjednávaním o zmluve*
- ***Reagovanie na zmeny*** *pred dodržiavaním plánu*

Ako súčasť agilného manifestu zadefinovali jeho autori tiež 12 téz, ktorými sa agilný vývoj riadi. Tieto princípy úzko súvisia a formujú vlastnosti agilných metodík.

Kľúčovým rysom, ktorým sa agilný prístup odlišuje od toho tradičného, je komunikácia so zákazníkmi a spracovávanie ich spätnej väzby priebežne počas implementácie systému. Nie len na začiatku pri zahájení a potom až pri akceptačnom testovaní a integrácii systému, ako je tomu pri lineárne-sekvenčnom *Vodopádovom modele*, ktorý je najstarším modelom životného cyklu softwaru [36]. Všetky agilné metodiky odporúčajú, ak nie priam vyžadujú, aby bol niekto od zákazníka súčasťou vývojového tímu a spolupracoval na návrhu, či spolurozhodoval o testoch. O tom tiež vypovedá jedna z téz: „*Lidé z byznysu a vývoje musí spolupracovat denně po celou dobu projektu.*“ [6]

Komunikácia zo strany zákazníka je nevyhnutná aj vzhľadom na ďalší z princípov, ktorý hovorí o tom, že zmeny v požiadavkách sú vítané počas celej doby vývoja, aj v neskorších fázach, pretože vďaka určitej novej vlastnosti systému môže klient získať konkurenčnú výhodu. Taktiež vedie k výraznému zníženiu rizika dodania produktu, ktorý nakoniec nespĺňa požiadavky zadávateľa, či už z dôvodu nejasnej a neúplnej špecifikácii alebo jej nepochopenia. Čas strávený prácou na zle definovanom systéme môže bezpochyby viesť ku katastrofálnym ekonomickým dopadom, či k úplnému neúspechu projektu. [22] Spoločným znakom agilných prístupov je teda schopnosť rýchlej reakcie na zmeny, a práve z tohto dôvodu sú najvhodnejšie pre projekty, ktoré majú na začiatku neisté a nejasné zadanie, prípadne ak dopredu predpokladáme často meniace sa požiadavky [6].

Ďalšie kľúčové princípy definované *Manifestom agilného vývoja softwaru* sa týkajú dodávok fungujúceho softwaru, a teda že ich priebežnosť by mala byť najvyššou prioritou a tiež, že by vývojový tím mal dodávať v intervaloch týždňov až mesiacov, ideálne v čo najkratších periódach [21].

Dôležitosť kvalitnej komunikácie už bola spomenutá vo vzťahu k zákazníkovi. Osobná konverzácia je však z pohľadu agilných metodík považovaná aj za najúčinnejšiu a najefektívnejšiu metódu, ako zdeľovať informácie vývojovému tímu zvonka, ale aj vzájomne medzi vývojármi v rámci tímu. Agilné prístupy sa na dokumentáciu pozerajú ako na nástroj porozumenia problému. Zároveň však tvrdia, že k tomuto porozumeniu sa najjednoduchšie dospeje práve osobnou komunikáciu a nie čítaním obsiahlych dokumentácií [18].

Ďalšia z téz hovorí: „*Budujeme projekty okolo motivovaných jednotlivcov. Vytvárame im prostredie, podporujeme ich potreby a dôverujeme, že odvedú dobrú prácu.*“ [6] To úzko súvisí s princípom podpory udržateľného vývoja, ktorý sa vylučuje s dlhodobým preťažovaním pracovníkov nadčasmi či prácou v noci. Z krátkodobého hľadiska sa to môže zdať ako východisko z nepriaznivej situácie, ale dlhodobo vedie k nízkej efektivite pri práci [18].

Princíp jednoduchosti agilných metodík sa dá najlepšie vystihnúť princípom *Occamovej britvy.* Ten hovorí, že pre vysvetlenie určitého javu nemá byť použitých viac entít, než je

skutočne nutné a všetky zbytočnosti treba eliminovať [15]. Navrhujeme a implementujeme teda len nevyhnutné minimum, ktoré zabezpečí požadovanú funkcionalitu.

Posledný princíp, ktorý spomeniem, no nie menej dôležitý, sa týka návrhu architektúry. Agilné prístupy zdôrazňujú, že zmeny v návrhu nevypovedajú o jeho predchádzajúcej nízkej kvalite. Ako už bolo spomenuté, zmeny sú prirodzenou a nevyhnutnou súčasťou vývoja. Preto je návrh, na rozdiel od tradičných životných cyklov vývoja softwaru, každodennou súčasťou vo všetkých fázach projektu [18].

Úzko prepojený termín, ktorý je nutné spomenúť, je *agilné testovanie*. To je nevyhnutné pre dosiahnutie kvality softwaru, ktorá je jednou z priorít agilného tímu. Jedná sa o prepojenú metodiku vývoja, keďže testovanie prebieha paralelne s implementáciou a nepretržite. Poskytuje tak kľúčovú spätnú väzbu a vďaka priebežným opravám je udržovaný čistejší kód. Do tohto procesu je zapojený nie len testovací tím, ale aj vývojári a zákazníci [20].

2.2 Extreme programming

Extrémne programovanie (XP) je najznámejšou z agilných metodík, využíva obecne známe princípy a postupy, ale ako už vypovedá názov, dovádza ich do extrémov. Zakladateľ tejto metodiky, Kent Beck, o nej tvrdí, že je metodikou účinnou, flexibilnou, predvídateľnou a tiež zábavnou [2].

A ako konkrétne extremizuje doterajšie postupy? Ak sa osvedčila revízia kódu, tak sa bude revidovať neustále. To sa dá dosiahnuť napríklad využitím techniky *pair programming*, párové programovanie, pri ktorom spolupracujú dvaja vývojári pri jednej pracovnej stanici. Jeden programátor píše zdrojový kód a druhý dohliada na kvalitu kódu, premýšľa nad inými, efektívnejšími spôsobmi programovania aktuálne písanej funkcionality a rozmýšľa nad problémami, ktoré sa môžu vyskytnúť a bude ich teda treba ošetriť. Zároveň spolu diskutujú o svojich nápadoch a dospievajú tak k lepším riešeniam. Výsledky nie sú na prvý pohľad očividné, ale z dlhodobého hľadiska prispieva párové programovanie k vysoko kvalitnému softwaru [9].

Ďalší prvok, ktorý je dovedený do extrému, je testovanie. Pri využívaní XP sa testuje neustále. Testujú vývojári, pomocou jednotkových testov, ktorými sa snažia otestovať každý možný prípad, ale tiež zákazníci, ktorí overujú validitu softwaru, a teda či software naozaj vykonáva to, čo od neho zákazník očakáva [11].

Pri tradičných vývojových technikách bolo jednou z najväčších nevýhod uvoľňovanie funkčných verzií vo veľmi dlhých časových intervaloch. Agilné prístupy dodávajú v kratších intervaloch a metodika XP dovádza aj tento aspekt opäť do krajnosti a dodávkové cykly sa namiesto týždňov a mesiacov pohybujú rádovo v hodinách či dokonca minútach. Takéto rýchle dodávky sú možné aj vďaka tomu, že metodika XP sa orientuje na zdrojový kód, ktorý považuje za uchovávateľa informácií a uprednostňuje ho pred rozsiahlymi dokumentáciami [18].

Ďalšou dôležitou úvahou je jednoduchosť. Cieľom je udržiavať systém v najjednoduchšej možnej podobe a teda implementovať iba tie vlastnosti, ktoré sú v danej chvíli nutné pre požadovanú funkcionalitu. Na zdokonaľovaní návrhu a architektúry sa pri využívaní XP pracuje priebežne a podieľajú sa na tom všetci zúčastnení. Refaktorizácia 2.6 je toho nevyhnutnou súčasťou [19].

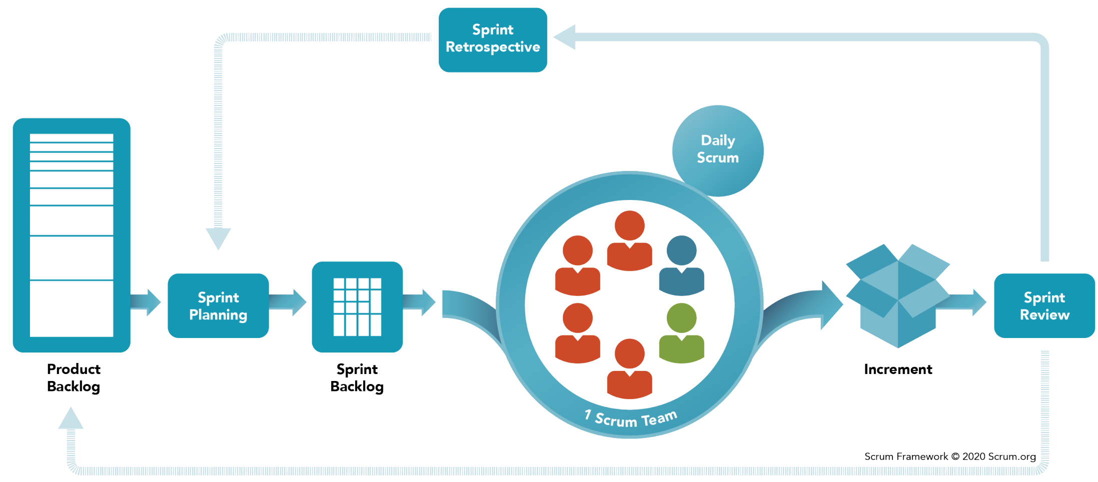
Metodika extrémneho programovania je vhodná pre menšie tímy, typicky dvaja až desať programátorov, a pre projekty, pri ktorých sa predpokladá častá zmena zadania či vopred nejasne špecifikované požiadavky. Z toho dôvodu tiež nehľadí príliš do budúcnosti a nevytvára zbytočne robustnú architektúru. Kladie dôraz na inkrementálny vývoj, krátke

iterácie a neodmysliteľnou súčasťou je úzky kontakt so zadávateľmi projektu. Významnú úlohu pri tomto spôsobe vývoja zohráva odvaha. Vedenie aj programátori musia byť pripravení zahodiť už dokončené časti kódu, ak sa ukáže, že existujúca architektúra nie je vhodná pre zapracovanie novej požiadavky. Avšak schopnosť prispôsobovať sa meniacim sa požiadavkám od zákazníka vedie nie len k lepším a trvalejším vzťahom, ale tiež k vyššej kvalite softwaru [18].

2.3 SCRUM

Metodika Scrum patrí jednoznačne medzi agilné metodiky aj vďaka tomu, že sa vyznačuje inkrementálnym a iteratívnym prístupom a jej cieľom je zvýšiť efektivitu tímu a celkovú hodnotu výsledného produktu [18].

Vývoj pomocou tejto metodiky prebieha podľa nasledujúcej schémy 2.1:



Obr. 2.1: Grafické znázornenie priebehu metodiky Scrum. Prevzaté z [32]

Keďže táto metodika zavádza veľa nových pojmov, princíp metodiky SCRUM bude popísaný práve cez vysvetlenie týchto pojmov.

Sprint

Tvorí esenciálnu súčasť Scrumu, trvá vopred stanovenú časovú periódu, typicky jeden týždeň až jeden mesiac, a počas neho tím vyvíja funkcionalitu, ktorú sa zaviazal počas neho dosiahnuť. Jednotlivé sprinty nadväzujú jeden na druhý a teda rozdeľujú celý proces vývoja do cyklov. Časová perióda musí byť dodržaná, ak sa však nestihol všetok vývoj, môže byť prenesený do ďalšieho sprintu. Či sa tak stane, záleží na rozhodnutí zákazníka. Tento koncept umožňuje pomerne rýchlu reakciu na meniace sa požiadavky a tiež redukuje plytvanie finančných zdrojov v prípade, že dôjde k nepochopeniu požiadaviek [35].

Scrum tím

Scrum tím tvoria tri kľúčové role: *Product Owner*, *Scrum Master* a *Scrum Team Member*, ktorých býva viacero. Každá z rolí má nezastúpiteľnú úlohu a nedoporučuje sa, aby jeden človek zastával viac rolí.

Product Owner, teda vlastník produktu, je za produkt zodpovedný. Je v zastúpení zákazníka a obhajuje jeho záujmy, definuje funkcionalitu produktu a rozhoduje o tom, na čom sa bude pracovať a je tým pádom zodpovedný za vytváranie *Backlogu* 2.3. Človek v tejto roli teda úzko komunikuje so zákazníkom, definuje užívateľské príbehy a tiež prioritu jednotlivých úloh. Leží na ňom aj zodpovednosť za pochopenie požiadaviek členmi vývojového tímu. Tento človek musí teda disponovať výbornými komunikačnými schopnosťami [21].

Scrum Master je človek, ktorý tím motivuje k lepším výsledkom, pomáha im pri dosahovaní cieľov a uisťuje sa, že všetci rozumejú agilným princípom vývoja. Odstraňuje vzniknuté problémy a tiež sa snaží o zachovanie príjemnej atmosféry v tíme a minimalizovanie vzájomných sporov. *Scrum Master* nefiguruje v tradičnej úlohe šéfa, keďže pri tejto metodike sa od členov očakáva samoorganizácie [21].

Scrum Team Member je člen tímu, ktorý sa riadi metodikou Scrum. Oproti iným metodikám nie sú pre členov definované tradičné role ako vývojár, tester, analytik či iné. Naopak v takomto tíme sa predpokladá, že členovia sú navzájom zastúpiteľní a sú schopní zastávať všetky role, aby nedochádzalo k tomu, že v určitej časti vývoja sa nemajú niektorí členovia na čom podieľať [35].

Artefakty

Medzi tri najhlavnejšie artefakty patrí *Product Backlog*, *Sprint Backlog* a *Increment*. Slúžia k uchopeniu požiadaviek na systém a k ich správnemu pochopeniu všetkými zainteresovanými. *Product Backlog* je *Product ownerom* upravovaný počas celého vývoja, zachytáva všetky funkčné požiadavky a reflektuje ich do užívateľských príbehov. Položky sú priebežne zoraďované podľa priority. *Sprint Backlog* je určitou podmnožinou *Product Backlogu*. Je vytvorený na začiatku každého *Sprintu* a jeho obsah je vyberaný na základe priorít užívateľských príbehov. *Sprint Backlog* by mal byť viditeľný všetkým členom buď pomocou online nástroja alebo vo fyzickej podobe, kde každý môže upravovať položkám ich status podľa toho, či sa na nich už začalo pracovať alebo ak sú už hotové. *Increment* je do slovenčiny preložený ako novo nasaditeľný produkt a predstavuje všetky implementované a už dokončené položky z minulých a aj aktuálneho *Sprintu* [35].

Pravidelné stretnutia

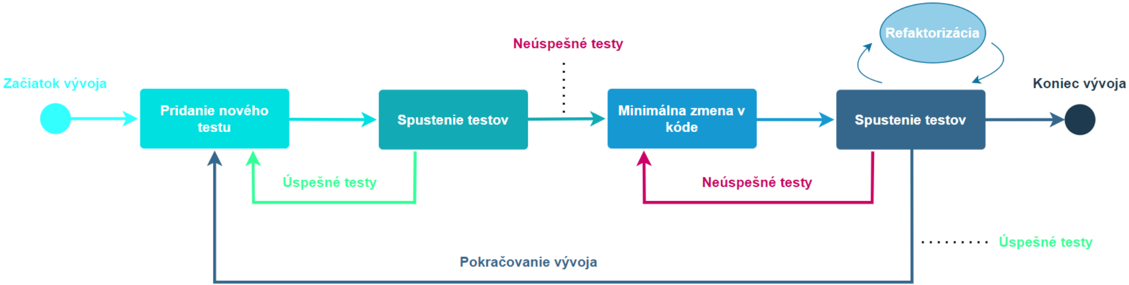
Scrum metodika predpisuje viacero typov stretnutí. Na plánovaní *Sprintu* prebieha vyberanie novo pridávanej funkcionality pre nasledujúci *Sprint*. *Sprint Review*, teda *Demo Sprintu* slúži na prezentáciu novej funkcionality zákazníkovi, ktorý poskytuje spätnú väzbu. Zúčastňujú sa na ňom teda všetky zainteresované strany. *Retrospektíva* sa odohráva na konci každého *Sprintu* a je určená pre vývojový tím a *Scrum Mastera*. Hodnotia spolu posledný *Sprint*, s akým problémov vývojári čelili a ako sa im ich podarilo zdolať, čo prebiehalo dobre, čo nie a ako by sa nasledujúci *Sprint* mohol vylepšiť [35].

2.4 Test Driven Development

To, že žiadny vývojový proces sa nemôže zaobísť bez fázy dôkladného testovania, je zrejmé. Dôsledkom by bolo až testovanie zákazníkmi, čo z finančných dôvodov a tiež v rámci zachovania vzťahov neprichádza do úvahy. Pri vodopádovom modeli patrilo testovanie až medzi posledné fázy vývoja a väčšina agilných metodík prevádza testovanie priebežne počas celého vývojového procesu. No metodika *Test Driven Development* (TDD), preložiteľná ako *Programovanie riadené testami*, je na testovaní priamo postavená.

Základnou myšlienkou pri vývoji riadenom testami je, že najskôr musí programátor vytvoriť jednotkový testovací prípad, až potom napíše zdrojový kód pokrývajúci tento test a napokon refaktorizuje 2.6.

Obrázok 2.2 zobrazuje rozhodovací proces pri vývoji riadenom testami.



Obr. 2.2: Grafické znázornenie metodiky Test Driven Development. Prevzaté z [18]

1. Prvým krokom pri riadení sa metodikou TDD je vytvorenie nového testovacieho prípadu. Programátor vytvára test s predpokladom, že má test zlyhať.

2. Nasleduje spustenie všetkých testov, prípadne ich určitej podmnožiny pri rozsiahlych projektoch. Dôležité je, aby novo pridaný test neprešiel úspešne. Pokiaľ všetky testy prebehnú v poriadku, je nutné vrátiť sa o krok späť a vymyslieť iný, zlyhávajúci test.

3. Ak však test zlyhal, programátor sa môže pustiť do implementácie tej časti kódu, ktorá zabezpečí úspešnosť novo pridaného testu. Zásadou je pridať do zdrojového kódu minimálne množstvo novej funkcionality, teda len tej, ktorá ovplyvňuje nový testovací prípad.

4. Keď sa nové zmeny v kóde zdajú dostačujúce, nasleduje fáza, v ktorej sa opäť spúšťajú všetky testy, prípadne ich podmnožina. Tieto dve fázy, zmeny v kóde a spustenie testov, je nutné opakovať až do chvíle, kým všetky testy neprejdú úspešne. Môže totiž nastať aj situácia, že nové zmeny zanesú chybu do kódu a prestanú fungovať už predchádzajúce testy.

5. Keď sú všetky testy úspešné, nasleduje fáza refaktorizácie kódu 2.6 a odstraňovania vzniknutých duplicít.

6. Ak vývoj ešte nie je úplne dokončený, programátor sa vracia na začiatok do prvej fázy a teda na vytvorenie nového testovacieho prípadu.

Jednotlivé cykly prebiehajú na úrovni konkrétnych funkcií, prípadne modulov či subsystémov a jeden takýto cyklus trvá pomerne krátku dobu, zväčša len niekoľko minút [18].

Takýto štýl programovania kladie však aj určité podmienky pre jeho úspešné aplikovanie. Vzhľadom na dĺžku trvania jedného cyklu je nutné, aby si testy písal programátor sám a taktiež vývojové prostredie, v ktorom pracuje, musí byť schopné rýchlo reagovať na zmeny. Je nutné, aby testy boli navzájom nezávislé, boli prevediteľné v krátkom čase a ideálne aby ich vstupmi boli reálne dáta. Pre uľahčenie testovania by mali návrh a architektúra systému pozostávať z voľne spojených softwarových komponent [3].

Pre programátorov býva pomerne náročné, najmä zo začiatku, dodržiavať pravidlá tejto techniky. Je veľmi jednoduché skĺznuť k napísaniu viac zdrojového kódu, než je nutné pre konkrétny test. Pri takto riadených projektoch je teda vhodný prísnejší prístup zo strany manažmentu alebo kombinácia s technikou *párového programovania*, podrobnejšie popísaná v podkapitole 2.2, kedy programátori môžu navzájom dohliadať jeden na druhého [18].

Čo však prísne dodržiavanie vývoja riadeného testami prináša? Keďže táto metodika sa sústreďuje na testovanie každej funkcionality hneď zo začiatku, výsledné pokrytie kódu testami je pri tejto metodike výrazne vyššie v porovnaní s inými metodikami vývoja [37]. Ďalšou výraznou výhodou v porovnaní s inými metodikami je nepretržitá spätná väzba, vďaka ktorej sú chyby opravované ihneď po ich vytvorení a v čase, kedy sa programátor zaoberá miestom ich výskytu. To súvisí tiež s tým, že pri využití TDD sa prakticky vývoj nedostáva do fáze *debuggingu*, teda hľadania chýb.

Vďaka tejto metodike je vývoj predvídateľný a vedenie projektu môže požiadavky vyhlásiť za naplnené bez toho, aby sa museli zaoberať zdĺhavým hľadaním chýb. Ak sa dá dostatočne znížiť počet chýb a zaistiť kvalita týmto spôsobom, môže potom tím pracovať na novom vývoji a neplytvať energiou reakciami na chyby. Menší počet chýb a nepríjemných prekvapení umožňuje manažérom projektu lepšie prerozdelenie práce, rozvrhnutie v čase a dodávanie funkčných programov načas, čo vedie k budovaniu lepších partnerských vzťahov [3].

Prístup TDD teda nielen napomáha zmenšovať prekážky brániace kvalite a pravidelnému dodávaniu softwaru, ale podporuje aj tvorenie optimalizovaného kódu, zvyšuje produktivitu vývojového tímu a pomáha vývojárom lepšie analyzovať a pochopiť požiadavky zákazníka. Vďaka ľahko udržateľnej a flexibilnej kódovej základni navyše uľahčuje budúci vývoj a údržbu [37].

2.5 Ďalšie agilné metodiky

Do rodiny agilných metodík patrí tiež *Lean development*. Táto metodika nebola priamo vyvinutá pre vývoj softwaru, je prevzatá z iných inžinierskych odvetví. Nepopisuje presne, akým spôsobom vyvíjať, ale združuje niekoľko kľúčových pravidiel a princípov, ktoré by mali viesť k optimálnemu, efektívnemu a kvalitnému vývojovému procesu. Medzi tieto pravidlá patrí odstrániť všetko, čo je zbytočné, teda minimalizovať zásoby a medziprodukty, nesnažiť sa neustále optimalizovať a skrátiť teda čas potrebný pre vývoj na minimum. Pravidlá *Lean developmentu* nechávajú rozhodovanie na poslednú chvíľu a umožňujú, aby prebiehalo aj na najnižších úrovniach. Ďalšie princípy sú spojené s vybudovaním istej kultúry pre možnosť zlepšovania, vybudovania lepších partnerstiev s dodávateľmi aj zákazníkmi, uspokojovať ich požiadavky a zaviesť spätnú väzbu, teda nebáť sa zmeniť už vykonané rozhodnutia [18].

Feature Driven Development (FDD), teda vývoj riadený vlastnosťami, je ďalšou z agilných metodík, ktorá kladie silný dôraz na výsledný produkt. Vývoj sa delí na päť fáz, pričom prvé tri sú sekvenčné a sústreďujú sa na vytvorenie doménového modelu, ktorý popisuje celý

systém. Model sa následne prevedie do zoznamu vlastností – funkcionalít, ktoré prinášajú užívateľom hodnotu. Posledné dve fázy prebiehajú iteratívne a implementujú sa počas nich konkrétne vlastnosti systému [21].

Ako poslednú spomeniem ešte rodinu adaptabilných metodík *Crystal*, pri ktorej autor Alistair Cockburn vychádza z myšlienky, že každý vývojový proces je jedinečný, vždy aspoň mierne odlišný od iných predchádzajúcich, a veci, ktoré raz fungovali, nemusia fungovať stále. Niektoré metodiky sú viac zamerané technicky, projektovo, nástrojovo, no metodiky *Crystal* naopak kladú dôraz na ľudský faktor a vyzdvihujú vzájomnú, priebežnú a otvorenú komunikáciu. Snažia sa o vhodné zloženie a vedenie vývojového tímu, teda prácu s ľudskými zdrojmi, a znižujú objem byrokracie, papierovania a dokumentácie na najnižšiu možnú hodnotu, ale zároveň tak, aby bol projekt ešte stále úspešný a hlavne dokončený [18].

2.6 Refaktorizácia ako súčasť agilného vývoja

Refaktorizácia je proces, pri ktorom programátor prevádza zmeny vo vnútornej štruktúre kódu, pričom ale tieto zmeny nemajú žiadny vplyv na vonkajšiu funkcionalitu softwaru. Zmeny sú prevádzané postupne po malých krokoch, avšak ich kumulatívny efekt podstatne zlepšuje návrh celého systému a jeho kvalitu [8].

Refaktorizácia ide ruku v ruke s agilným vývojom, ktorého cieľom je podporovať experimentovanie a hľadanie jednoduchých a efektívnych riešení. Agilné metodiky sa vyhýbajú rozsiahlemu, dlhodobému plánovaniu a práve refaktorizácia umožňuje ľahšiu adaptabilitu na zmeny, ktoré s určitosťou budú prichádzať. Taktiež vďaka kontinuálnemu vylepšovaniu kódu po malých krokoch pomáha vývojovému tímu udržať vývoj komplexného softwaru pod kontrolou [7].

2.6.1 Čo refaktorizácia prináša

Meniť kód tak, aby sa nezmenilo jeho chovanie, sa dá rôznymi spôsobmi. Pri refaktorizácii sa však prevádzajú práve tie zmeny, ktoré vedú ku kódu jednoduchšiemu na pochopenie a ktorý sa teda jednoduchšie udržuje a rozširuje. Hlavným cieľom je odstraňovanie duplicít, ktoré môžu viesť k nekonzistencii, teda ak je chyba opravená na jednom mieste, no zabudne sa na inú časť kódu, ktorá vykonáva takmer totožnú činnosť. Ďalej je tiež nutné premiestňovať tie časti kódu, ktoré sú na nesprávnych miestach, napríklad metódy, ktoré by sa lepšie hodili do inej triedy. Niekedy je na mieste odstrániť či pridať celú triedu alebo rozčleniť kód na iné metódy, ktorých názvy lepšie vystihujú ich význam.

Aj keď refaktorizácia zaberie určitý čas, počas ktorého sa nepridáva žiadna nová funkcionalita, tak lepšia štruktúra kódu, jeho väčšia zrozumiteľnosť a celkovo lepší návrh vedú k výrazným úsporám času pri rozširovaní systému a pridávaní nových zmien a teda celkovo prispieva k rýchlejšiemu vývoju. Naviac refaktorovanie nielenže vyžaduje pochopenie kódu, čo samotné môže viesť k odhaleniu chýb, ale vo všeobecnosti čím je kód jednoduchší na pochopenie, tým ľahšie sa v ňom prípadné chyby hľadajú [8].

2.6.2 Ako refaktorovať

Zakladateľ extrémneho programovania, Kent Beck, navyše odporúča programátorom riadiť sa princípom *dvoch klobúkov*, pokiaľ pri vývoji softwaru pravidelne využívajú aj refaktorizáciu. To znamená vedieť rozlíšiť medzi činnosťou, kedy pridávajú novú funkcionalitu, prípadne nové testy, a počas tohto času nijak nezasahovať do už existujúceho kódu, iba

pridávať nový. Naopak, keď zistia, že ďalší vývoj by sa im implementoval jednoduchšie pri zmene štruktúry, mali by si nasadiť druhý pomyslený klobúk a začať refaktorovať, teda meniť existujúci kód, ale nepridávať žiadnu novú funkcionalitu a ani nové testy [8].

Aby mal vývojový tím pri refaktorizácii istotu, že sa nemení správanie softwaru navonok, je najlepším riešením mať pri refaktorizácii k dispozícii sadu testov, ktoré je možné pravidelne prevádzať, podobne ako pri metodike *Test driven development* 2.4. Testy poskytujú programátorom istotu, ktorá im umožňuje refaktorovať odvážnejšie a zároveň je pri chybe možné sa vždy vrátiť do stavu, kedy testy naposledy fungovali [7].

Refaktorovanie môže byť rýchlym a jednoduchým procesom, pokiaľ sa človek riadi osvedčenými postupmi, ktoré sú systematické, čistia kód cielene a efektívne a minimalizujú riziko zavedenia nových chýb. Postupy, ako správne refaktorovať, sú popísané a vysvetlené na príkladoch v knihe *Refaktoring: zlepšení existujícího kódu* od Martina Fowlera [8] a využívam ich pri implementácii ABAP reportu, ktorému je venovaná kapitola 4.

2.6.3 Kedy je refaktorizácia (ne)vhodná

Naskytuje sa prirodzená otázka, kedy sa refaktorovať oplatí, a kedy nie. Ako už bolo spomenuté, vhodným časom kedy uvažovať o refaktorizácii, je práve pred pridávaním novej funkcionality, ktorú proces refaktorizácie môže uľahčiť a je tiež jedným zo spôsobov, ako program rýchlo pochopiť, ak je pre vývojára kód celkom neznámy alebo sa k nemu vracia po dlhšom čase. Ďalším dobrým dôvodom je, keď sa objaví chyba, ktorú nie je možné jednoducho odhaliť. Keďže refaktorizácia vedie k lepšiemu porozumeniu a čitateľnosti kódu, je možné, že chybu programátor nájde prirodzene počas tohto procesu. Niektoré organizácie navyše pravidelne revidujú kód, čo tiež poskytuje vhodný čas na refaktorizáciu a zároveň sa pri tomto procese predávajú a rozširujú znalosti celého vývojového tímu [8].

A kedy naopak nie je vhodné refaktorizáciu využiť? Je dôležité podotknúť, že refaktorizácia by sa mala prevádzať iba vtedy, ak software viac-menej funguje a väčšina testov je prevádzanú úspešne. Pokiaľ to tak nie je, tak by proces refaktorizácie pravdepodobne nebol príliš efektívny a je oveľa vhodnejšie začať celý software implementovať od začiatku. Ďalšia situácia, pri ktorej by refaktorovanie mohlo spôsobiť viac škody než úžitku, je tesne pred termínom dokončenia projektu. Keď sa tento proces začne, môže byť neočakávane časovo náročný a znamenať pre klienta dodatočné navýšenie finančných nákladov [34]. Ak sa ale vývoj nenachádza v úplne finálnej fáze, nie je vhodné sa refaktorizácii vyhýbať z dôvodu nedostatku času. Práve naopak, keďže refaktorizácia prináša vyššiu produktivitu a rýchlejší vývoj, tak nedostatok času je známka toho, že je potrebné refaktorovať [8].

Pri refaktorizácii je tiež dôležité mať na pamäti, že sa vlastne nikdy nedá považovať za ukončenú. Každý kód bude napokon považovaný za zastaralý alebo jednoducho nebude vhodný pre implementovanie nových požiadaviek a bude teda vyžadovať ďalšie refaktorovanie. Kľúčom je sústrediť sa na pokrok, nie na úplnú dokonalosť a rozpoznať situácie, kedy už refaktorovanie prestáva byť efektívne [34].

Kapitola 3

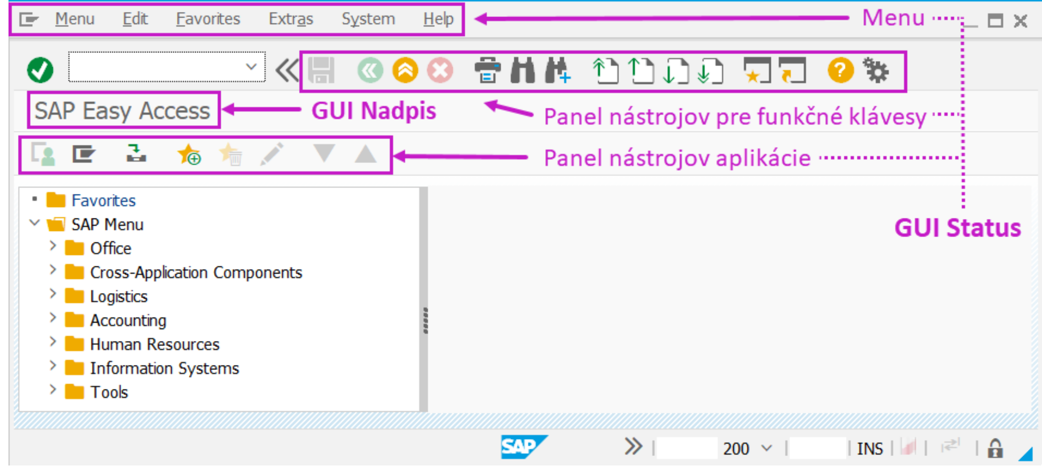
Vývoj softwaru a jednotkové testovanie v jazyku ABAP

V tejto kapitole je predstavené prostredie a nástroje používané pre vývoj softwaru spoločnosti SAP a špeciálne typy obrazoviek, ktoré sa využívajú pri implementácii takzvaných *ABAP reportov*. Hlavnou úlohou takýchto programov je organizovaným spôsobom prezentovať užívateľovi vybrané dáta z databázy. Podkapitola 3.3 je venovaná jednotkovému testovaniu a akým spôsobom sa v jazyku ABAP implementuje.

3.1 Nástroje používané pre vývoj v SAP

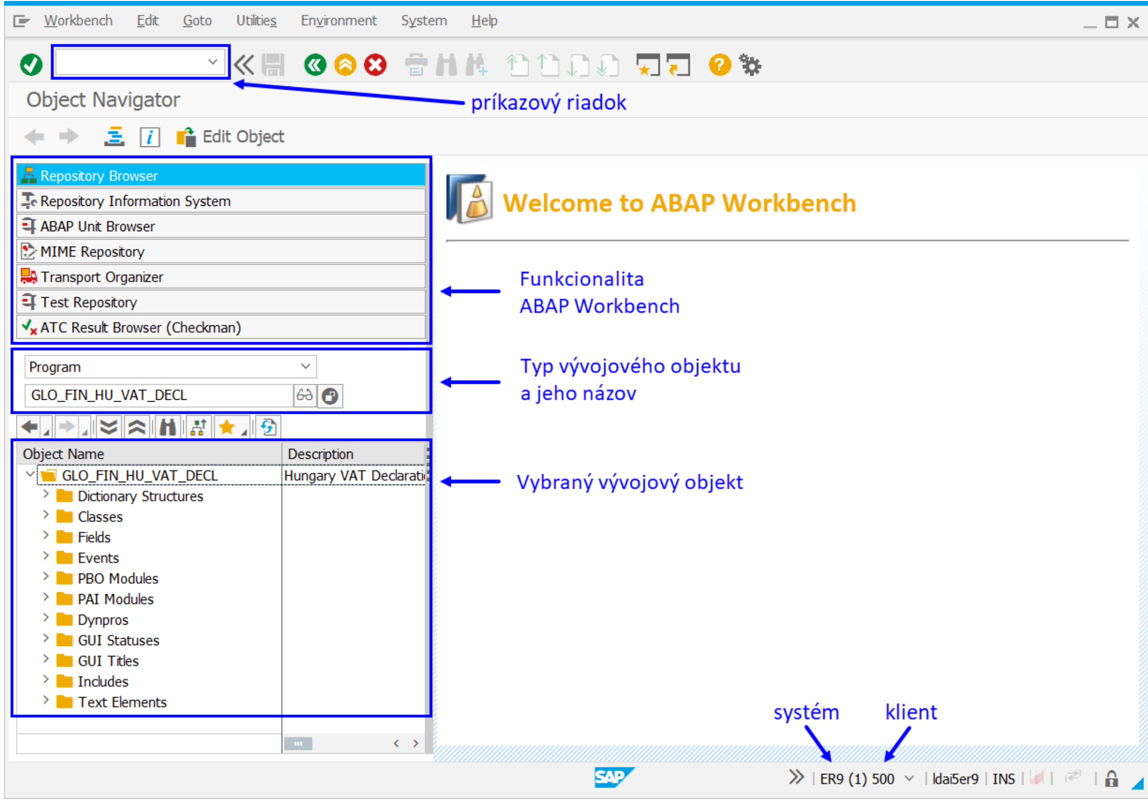
V sedemdesiatych rokoch vyvinula spoločnosť SAP proprietárny programovací jazyk ABAP za účelom vývoja podnikových informačných systémov – ERP (enterprise resource planning). Tieto systémy umožňujú zákazníkom riadiť ich obchodné procesy zahŕňajúc účtovníctvo, výrobu, marketing a predaj či riadenie ľudských zdrojov [10].

Pre prístup do SAP systému sa štandardne využíva SAP GUI, obrázok 3.1, ktoré po inštalácii na Windows poskytuje všetky funkcie pre vývoj a spúšťanie aplikácií. Inštalácia na operačné systémy Mac alebo Linux, či dokonca prístup cez internetový prehliadač, sú tiež možné, no spájajú sa s určitými obmedzeniami a nie sú preto až tak populárne [1].



Obr. 3.1: SAP GUI

Prístup k existujúcim aplikáciám a samotný vývoj v jazyku ABAP je umožnený vďaka transakciám, ktoré sa zadávajú do príkazového riadka. Na základe zadanej transakcie sa dajú spustiť rôzne nástroje, ktoré poskytuje *ABAP Workbench*, čo sa dá preložiť ako ABAP Pracovná plocha. Medzi najdôležitejšie nástroje patria *ABAP Editor*, *Function Builder*, *Class Builder*, *Screen Painter*, *Menu Painter*, či *ABAP Data Dictionary*. Pri vývoji novej aplikácie je však potrebné využívať viaceré nástroje naraz a preto existuje *Object Navigator*, obrázok 3.2, prístupný z transakcie SE80. *Object Navigator* zabezpečuje centrálny prístup ku kľúčovým nástrojom *ABAP Workbench* a umožňuje tak jednoducho organizovať, vytvárať a meniť súvisiace objekty aplikácie z jedného miesta [1].



Obr. 3.2: Object Navigator

3.2 Interakcia s užívateľom pomocou dynpra

V prostredí SAP GUI sa na interakciu s užívateľom využíva tzv. *dynpro (dynamic program)*, ktoré je vždy komponentou konkrétneho programu. Samotné dynpro pozostáva z obrazovky, zoznamu elementov a tzv. *flow* logiky.

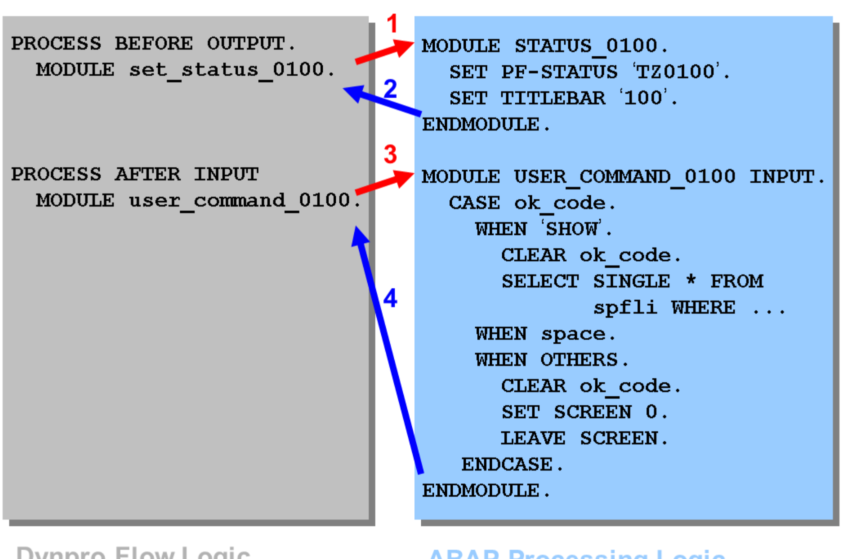
Obrazovky sa delia na tri typy – *selection screen*, *list screen* a *general screen*. Pre vytvorenie *list screen* stačí, aby sa v časti globálnej definície programu nachádzal príkaz `WRITE` a pri aktivácii programu sa automaticky vygeneruje relevantné dynpro. Rovnakým spôsobom je možné vytvoriť aj základnú *selection screen*, jej generovanie je naviazané na kľúčové slová `SELECT-OPTION` alebo `PARAMETERS`, ktoré slúžia na špecifikovanie výberových kritérií užívateľom. Na základe týchto kritérií sú potom užívateľovi na výstupe z databázy repor-

tované iba relevantné dáta. Pomocou príkazu `SELECTION-SCREEN` a číselného označenia sa dajú vytvárať ďalšie takéto obrazovky.

Na vytvorenie *general screen* je nutné využiť nástroj *Screen Painter*, ktorý na Windowse poskytuje aj grafický editor rozloženia, ktorý tvorbu obrazovky ešte viac zjednodušuje. Na tento typ obrazovky je okrem základných elementov možné vkladať aj vlastné ovládacie prvky, ako je napríklad často využívaný *container*, vďaka ktorému je následne možné využívať knižnicu *SAP List Viewer (ALV)*. Táto knižnica automaticky spravuje výstup reportu a zabezpečuje ďalšie užitočné operácie, ktoré je možné nad dátami prevádzať pre ich lepšiu organizáciu a čitateľnosť na výstupe [1].

Všetky obrazovky majú 4-číselný identifikátor a zdieľajú v rámci programu jeden menný priestor, čo umožňuje, aby každý program obsahoval viacero dynpier. Tie je potom možné volať z rôznych častí programu pre lepšiu organizáciu výstupu pre užívateľa.

Ďalšou súčasťou dynpra je *flow logic*, teda logika toku, ktorá pozostáva z určitých blokov udalostí a tie umožňujú ovplyvňovať správanie obrazovky a tiež reagovať na akcie vykonané užívateľom. PBO – *Process Before Output* je blok udalostí, ktoré sa vykonajú vždy ešte predtým, než je obrazovka zobrazená užívateľovi a využíva sa na pripravenie obrazovky, teda napríklad priradenie predvolených hodnôt, či dynamicky zobraziť/skryť určité políčka. Udalosti, ktoré obsahuje blok PAI – *Process After Input*, sú naopak vyvolané až po nejakej akcii vykonanej užívateľom a umožňujú teda reagovať na ňu. Ako je vidno na obrázku 3.3, do týchto blokov dynpra sa vkladajú volania *dialógových modulov*. Implementácie týchto modulov sa nachádzajú v rámci tela hlavného programu, čím je zabezpečená logika spracovania celého programu.



Obr. 3.3: Volanie dialógových modulov dynpra z hlavného programu. Prevzaté z [31]

V zozname elementov dynpra sa môžu nachádzať texty, parametre, *select-options*, ktoré umožňujú zadanie rozsahu, zaškrtávacie políčka, prepínače, rozbalovacie zoznamy a ďalšie. Keď sa v globálnej definícii programu nachádzajú dátové objekty, ktoré majú zhodný názov s elementami dynpra, tak v PAI udalosti sa ich hodnoty z obrazovky automaticky prenášajú do programu a naopak v PBO udalosti sa hodnoty z programu prenášajú do príslušných elementov dynpra.

Ďalšími dôležitými komponentami ABAP programu, ktoré sú úzko prepojené s obrazovkami, sú GUI Status a GUI Nadpis, viď 3.1. Tieto komponenty sa nastavujú zvlášť pre každú obrazovku v PBO moduloch. GUI Status sa vytvára pomocou nástroja *Menu Painter* a zabezpečuje, že po vykonaní užívateľskej akcie, sa do programu prenesie funkčný kód toho prvku, na ktorý bolo kliknuté.

3.3 Jednotkové testovanie v jazyku ABAP

Existuje veľa úrovní, na ktorých sa dá software testovať a tou najnižšou úrovňou je testovanie jednotiek, teda *unit testing*. Pri jednotkovom testovaní sa overuje korektná funkcionalita najmenších častí aplikácií, ako sú triedy či metódy, ktoré je možné pri testovaní izolovať od ostatných častí kódu. Unit testovanie je najčastejšie využívané na aplikačnej vrstve, kde sa overujú výsledky výpočtov, spracovania dát a celkovo business logiky. Unit testy sú tiež základným stavebným kameňom pri využívaní metodiky *Test Driven Development* 2.4 a je možné ich spúšťať manuálne aj automaticky [38].

Vytváranie jednotkových testov spadá pod zodpovednosti programátora, nie testera. Písanie testovacích prípadov pre všetky metódy môže byť časovo náročné, no tento spôsob testovania je napriek tomu považovaný za najlacnejší, keďže prebieha zároveň s vytváraním kódu a chyby teda môžu byť opravené prakticky okamžite. Navyše vyžaduje od vývojárov písať kód takým spôsobom, aby bol dobre testovateľný, čo vedie k vytváraniu nie veľmi rozsiahlych metód, ktoré sú čo najmenej závislé od ostatného kódu, a to je v súlade s doporučeniami pre zvyšovanie kvality kódu [12].

Funkčná sada unit testov ale tiež zabezpečuje obrovskú výhodu do budúcnosti. Ich opakované spúšťanie priebežne počas implementácie poskytuje istotu, že nový vývoj neovplyvňuje už existujúcu funkcionalitu, že sa novými časťami kódu nezaniesli do softwaru neočakávané chyby [1].

V jazyku ABAP je zakotvený *unit test framework*, ktorý poskytuje automatizovaný mechanizmus pre aplikovanie TDD prístupu. *ABAP Unit Framework* umožňuje vývojárom testovať jednotky kódu nezávisle v izolovanom prostredí a bez obáv o ovplyvnenie celého riešenia. Testovanie týchto jednotiek zabezpečuje, že keď sú spojené všetky dohromady do väčšieho celku, budú fungovať bez chýb [38].

3.3.1 Umožnenie testovateľnosti odstránením závislostí v programe

Pre správne fungovanie unit testov je nutné ich odizolovať od reálnych databázových tabuliek, aby vykonávanie testov nebolo závislé na dátach, ktoré v databáze nachádzajú v čase vykonávania testov. Pri testovaní ABAP programov, ktoré čítajú či zapisujú do určitých dátových zdrojov, existuje *ABAP SQL Test Double Framework*, ktorý zabezpečuje odstránenie závislostí a nahradenie týchto zdrojov takými, ktoré sú vytvorené špeciálne pre testovacie prípady [1]. Tento *framework* sa využíva nasledovne [24]:

Pri vytváraní testovacej triedy je nutné do nej pridať statický atribút:

```
CLASS-DATA: environment TYPE REF TO if_osql_test_environment.
```

Potom je nutné vytvoriť list závislostí, k čomu slúži metóda `create`, a je volaná vždy len raz pre každú testovaciu triedu.

```
cl_osql_test_environment=>create( i_dependency_list = '<dependency_list>' )
```

Testovacie dáta sa následne vkladajú volaním metódy `insert_test_data` nad triednym atribútom `environment`, teda takto:

```
environment->insert_test_data( <test_data> )
```

Zmazanie dát sa prevádza metódou `clear_doubles` a pomocou metódy `destroy` sa ruší celé toto testovacie prostredie vrátane vymazania vložených dát.

Odstránenie závislostí pomocou rozhraní

Ďalší spôsob, ktorý sa využíva pre programovanie kódu s čo najmenšími závislosťami je využitie rozhraní – *interface*. Rozhrania sa dajú považovať za abstraktnú definíciu funkcionality. Slúžia k špecifikácii, ako s nejakou entitou komunikovať, čo všetko dokáže a poskytuje. Rozhrania však neposkytujú implementáciu, teda akým spôsobom je dané správanie zabezpečené [23].

Snaha o obmedzenie závislostí na minimum je dôležitá preto, že čím menej závislostí kód obsahuje, o to menej je pravdepodobné, že by zmena v jednej časti kódu mohla neželane ovplyvniť aj niektorú inú časť. Triedy by teda mali byť navrhované takým spôsobom, aby neboli závislé na konkrétnej implementácii iných tried. Využitie rozhraní umožňuje oddeliť kód od akejkoľvek konkrétnej implementácie a v prípade, že sa objaví nejaká lepšia, nahradiť ju bez obáv z narušenia celkovej funkcionality. Dá sa teda povedať, že kód má čo najmenšie možné množstvo závislostí vtedy, keď jediné na čom je závislý, sú práve rozhrania [17].

Využitie rozhraní navyše umožňuje veľmi jednoduché rozširovanie funkcionality. Ak sa po čase objaví požiadavka na implementáciu nového typu objektu, ktorý je veľmi podobný iným objektom tried implementujúcich určité rozhranie, dá sa veľmi jednoducho pre nový objekt vytvoriť nová trieda, ktorá bude tiež implementovať to isté rozhranie. Týmto spôsobom nie je vyžadovaná žiadna dodatočná zmena v tých častiach kódu, ktoré pracujú s objektami implementujúcimi dané rozhranie [16].

Ďalším praktickým dôvodom k využitiu rozhraní je, že robia kód jednoducho testovateľným. Akákoľvek implementácia je totiž vďaka nim jednoducho nahraditeľná. To je vhodné napríklad vtedy, keď účelom testovania je overenie správnosti výpočtov prevádzaných nad dátami a nie ich samotné získanie z databázy. Vtedy je možné nahradiť implementácie metód pracujúcich nad databázou falošnými implementáciami a teda poskytnúť pre testovanie akékoľvek vzorové dáta bez dlhého prevádzania databázových operácií, ktoré je v danom prípade nežiadúce [16].

3.3.2 Vytváranie a implementácia testovacích tried

Pred implementáciou unit testov je nutné najskôr vytvoriť testovacie triedy. Testovacie triedy pre daný program sa vždy definujú v *include* daného programu, a to pridaním kľúčového slova `FOR TESTING` hneď za príkazom definujúcim triedu. Pri definícii je tiež treba špecifikovať vlastnosti ako `RISK LEVEL` (úroveň rizika) a `DURATION` (trvanie). Na základe úrovne rizika systém spustí len tie testy, ktoré vyhovujú nastaveniam klienta. Testy, ktoré majú vyššie riziko, než je aktuálne požadované, sa nevykonajú. Testy, ktorých prevádzanie trvá dlhšie, než špecifikuje ich trvanie, skončia neúspešne [38].

Risk level môže nadobúdať nasledujúce hodnoty:

- **Harmless** – *neškodné* – testy nespôsobujú zmeny v databáze
- **Dangerous/alarming** – *nebezpečné/alarmujúce* – testy zapisujú do databáze

- **Critical** – *kritické* – testy robia zmeny v prispôsobení a perzistentných dátach

Duration určuje dobu trvania testu, ktorá sa pohybuje v nasledujúcich hraniciach:

- **Short** – *krátke* – testy, ktoré prebehnú veľmi rýchlo, typicky za menej ako 60 sekúnd
- **Medium** – *stredne dlhé* – testy, trvajúce v rozmedzí 60 a 600 sekúnd
- **Long** – *dlhé* – testy trvajúce značnú dobu, viac ako 600 sekúnd

Podobne ako pri testovacích triedach, aj pri testovacích metódach je nutné za ich definíciu pridať kľúčové slovo `FOR TESTING`. Práve tento dodatok ich totiž jednoznačne odlišuje od produkčného kódu.

Definícia testovacej triedy môže teda vyzerať nasledovne:

```
CLASS abap_unit_testclass DEFINITION FOR TESTING
     DURATION SHORT
     RISK LEVEL HARMLESS.

  PUBLIC SECTION.
    METHODS unit_test_method FOR TESTING.

ENDCLASS.
```

Test story

Každý testovací prípad by mal rozprávať určitý príbeh, vďaka ktorému nezainteresovaný čitateľ testu vie, čo sa programátor snažil testom dosiahnuť. Je možné si to predstaviť nejak takto: "Je daný určitý kontext, keď spustím testovaný kód, očakávam tento výsledok." Dobrou praktikou je teda rozdeliť test na tri časti [14]:

1. *given* – na začiatku testovacieho prípadu je vytvorený objekt triedy, ktorú treba otestovať a jej referencia je uložená do premennej *cut* – class under test.
2. *when* – v tejto časti sú volané metódy testovanej triedy a ich výsledky sú uložené do premenných
3. *then* – posledná časť slúži na porovnanie skutočného výsledku s tým očakávaným, čo rozhodne o úspešnosti testovacieho prípadu

Metódy SETUP a TEARDOWN

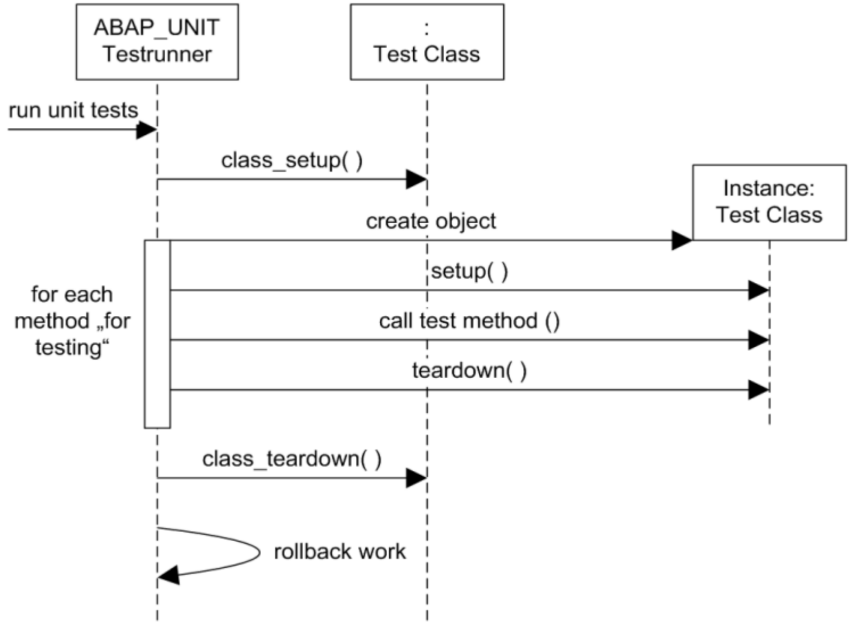
Keď je týmto spôsobom napísaných viac testov, je možné si všimnúť, že prvá časť *given* sa v testoch opakuje. Na odstránenie týchto duplicít poskytuje *ABAP Unit* preddefinované metódy, ktoré majú špeciálnu funkciu v určitých špeciálnych miestach. Zabezpečujú, aby každý test bol prevedený v novom kontexte. Použitie týchto metód je dobrovoľné a sú nimi tieto štyri [38]:

- `SETUP( )` – inštančná metóda, ktorá je volaná pred každou jednou testovacou metódou danej triedy. Využíva sa pre prípravu kontextu pre testovací prípad, teda pre vytvorenie testovacích dát.

- `TEARDOWN( )` – inštančná metóda, ktorá je volaná po vykonaní každej testovacej metódy danej triedy. Využíva sa pre vyčistenie kontextu po každom teste.
- `CLASS_SETUP( )` – statická metóda volaná len raz, a to skôr než je vykonaný akýkoľvek test danej triedy. Využíva sa pre inicializáciu premenných, ktoré budú v testoch používané.
- `CLASS_TEARDOWN( )` – statická metóda, ktorá je volaná až po dokončení všetkých testov danej testovacej triedy. Využíva sa pre vyčistenie využitých dátových premenných.

Metóda *teardown* môže byť obzvlášť dôležitá. Človeku sa môže zdať, že kód, ktorý obsahuje, by sa dal vložiť na koniec testovacích metód. Nie je tomu však tak. Ak totiž testovacia metóda odhalí chybu, jej vykonávanie sa v danom mieste zastaví a zvyšný kód v tele metódy sa už nevykoná. V tej chvíli sa spustí metóda *teardown*, ktorá je vykonaná vždy po ukončení testovacej metódy, bez ohľadu na jej výsledok [14].

Na nasledujúcom obrázku 3.4 je zobrazené, akým spôsobom vykonávanie unit testov prebieha. Je možné vidieť, že pre každý test sa vytvorí nová inštancia testovacej triedy. Nutné je ešte dodať, že testy sú prevádzané nezávisle jeden na druhom a vo vopred nedefinovanom poradí [14].



Obr. 3.4: Priebeh programu vykonávajúceho unit testy. Prevzaté z [14]

Assertion metódy

V poslednej časti testovacieho príbehu, v *then* časti, sa využívajú metódy triedy `CL_ABAP_UNIT_ASSERT`, ktoré poskytuje *ABAP Unit*. Vďaka týmto metódam, ktoré sa nazývajú *assertion methods*, môžme porovnávať konkrétne hodnoty či celé tabuľky, ktoré po vykonaní produkčnej metódy očakávame. Medzi najpoužívanejšie z týchto metód patria [38]:

- `ASSER_EQUALS` – skontroluje, či sú dva dátové objekty rovnaké

- `ASSERT_DIFFERS` – skontroluje, či sú dva dátové objekty rozdielne
- `ASSERT_BOUND` – kontroluje, či premenná odkazuje na validnú referenciu
- `ASSERT_NOT_BOUND` – vyžaduje, aby referencia premennej bola nevalidná
- `ASSERT_INITIAL`, `ASSERT_NOT_INITIAL` – kontroluje, či dátový objekt obsahuje hodnotu *initial*/hodnotu rôznu od *initial*
- `ASSERT_SUBRC` – vyžaduje konkrétnu hodnotu návratového kódu `SY-SUBRC`
- `ASSERT_TRUE`, `ASSERT_FALSE` – hodnota premennej musí byť true/false
- `ASSERT_TABLE_CONTAINS`, `ASSERT_TABLE_NOT_CONTAINS` – vyžadujú, aby nejaký záznam internej tabuľky obsahoval/neobsahoval konkrétne dáta

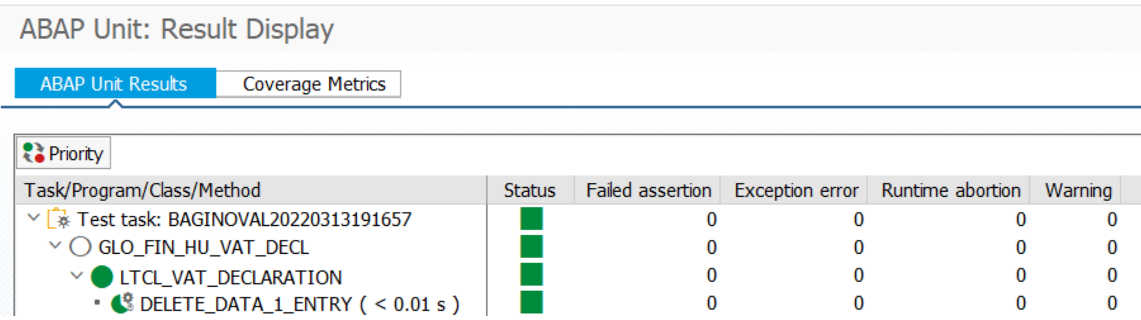
3.3.3 Spúšťanie testov, reprezentácia výsledkov

Najjednoduchší spôsob, ako spustiť unit testy pre daný program, je pravým kliknutím na program → **Execute** → **Unit tests**. Pokiaľ všetky testovacie triedy a metódy prebehnú úspešne, zobrazí sa v dolnej lište *status message* so štatistikou o ich počte (viď obr. 3.5).

Processed: 1 programs, 6 test classes, 27 test methods

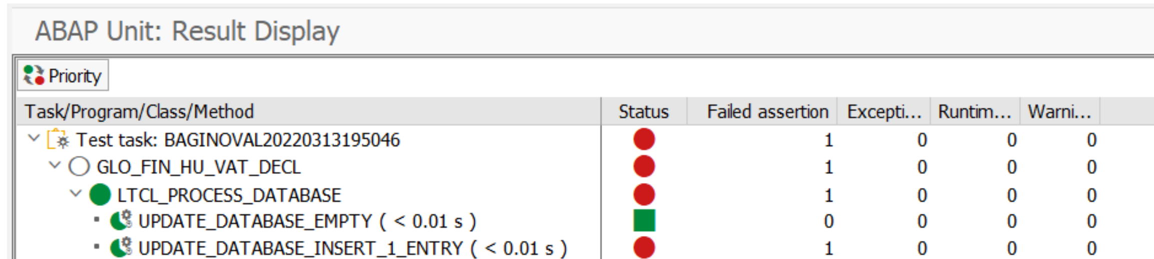
Obr. 3.5: Informácia o úspešnom výsledku testov

Ďalšou možnosťou spustenia testov je vybrať **Execute** → **Unit Tests With** → **Coverage**, čo zobrazí detajlnejšie informácie o testovacích triedach a metódach (viď obr 3.6). Na záložke *Coverage Metrics* sa nachádzajú pokročilé metriky s absolútnymi aj percentuálnymi štatistikami o pokrytí vetiev, procedúr a príkazov.



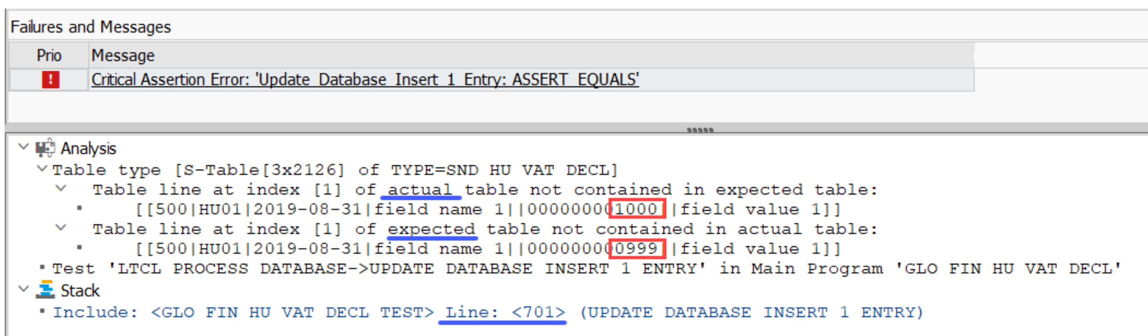
Obr. 3.6: Podrobné výsledky unit testov

Pokiaľ niektorý z unit testov zlyhá, *ABAP Unit* automaticky zobrazí celú stromovú štruktúru, ktorá obsahuje informácie, pre ktorý program chyba nastala, v ktorej testovacej triede a ktorá metóda bola neúspešná a prečo (viď obr. 3.7).



Obr. 3.7: Zobrazenie zlyhávajúcich metód

Pre ešte jednoduchšiu identifikáciu chyby je možné vidieť, aká hodnota bola očakávaná a aká je v skutočnom výsledku. Pre rýchlejšiu navigáciu sa dá kliknúť na číslo riadku, čo nás prenesie do presného miesta, v ktorom testovacia metóda zlyhala (viď obr. 3.8).



Obr. 3.8: Podrobnosti o očakávanej a skutočnej hodnote

Kapitola 4

ABAP report pre manuálne zadávanie hodnôt výkazu DPH

Táto kapitola je venovaná programu implementovanom v jazyku ABAP, ktorý umožní zadávať hodnoty určitých položiek výkazu DPH manuálne. Implementácia tejto novej požiadavky spadá pod zodpovednosti tímu, v ktorom pre SAP pracujem. Nasledujúce podkapitoly vysvetľujú funkčné požiadavky, návrh a implementáciu s využitím refaktorizácie a tiež spôsob testovania daného programu.

Motivácia a požadovaná funkcionalita

Formálne požiadavky, ktoré musí spĺňať výkaz DPH, sú v každej krajine trochu odlišné a postupom času sa menia. Podľa nových zmien v maďarských daňových zákonoch, musia daňové subjekty do výkazu DPH zahrnúť aj určité hodnoty, ktoré nie je možné získať automaticky zo systému, ale musia byť zadané ručne.

V porovnaní s požadovanými formami reportovania DPH v iných krajinách, nie je táto požiadavka úplne netradičná. Implementáciu podobnej funkcionality je možné nájsť aj v iných ABAP reportoch, ktoré spoločnosť SAP svojim zákazníkom poskytuje. V prípade Bulharska sa tieto špeciálne hodnoty zadávajú ako parametre na *selection screen* programu, ktorý je zodpovedný za vytvorenie celého výkazu DPH. S takýmto riešením sa však spája určitá nevýhoda. Pokiaľ totiž nejaké položky vo výslednom výkaze nesedia, robia sa rôzne korekcie v dokladoch a proces vytvárania výkazu DPH prebieha odznova. Celý proces sa môže opakovať aj niekoľkokrát, až kým nie sú opravené všetky chyby a je teda možné výkaz DPH odovzdať príslušným daňovým autoritám. Určite sa nájdu aj typy daňových subjektov, pre ktoré tieto špeciálne položky nie sú relevantné a teda ich nevypĺňajú. V ostatných prípadoch však pri tomto riešení, kedy sú hodnoty zadávané ako parametre *selection screen*, je nutné manuálne zadávať dané hodnoty pri každom spustení programu odznova.

Ďalšie možné riešenie poskytuje poľský report, pri ktorom je zadávanie týchto hodnôt vyňaté ako samostatná aktivita a manuálne zadané hodnoty sú perzistentne uložené do databázy, aby mohli byť načítané v prípade opätovného prevádzania celého procesu. Náš *Product Owner* 2.3 učinil rozhodnutie, že práve túto možnosť budeme maďarským zákazníkom poskytovať. Problémom však pri tomto konkrétnom riešení je, že daný report bol vyvinutý relatívne dávno a nespĺňa kvalitatívne požiadavky, ktoré sa v dnešnej dobe snaží spoločnosť SAP dosahovať.

Pri implementácii nového ABAP reportu, budem teda vychádzať z už existujúceho poľského riešenia, pričom jeho funkcionalitu bude treba upraviť tak, aby vyhovovala maďar-

ským požiadavkám. Tento starší poľský report je navyše napísaný procedurálne a v súčasnej dobe je prakticky všetok nový vývoj v SAP objektovo orientovaný. Preto budem kompletne meniť štruktúru programu na objektovo orientovanú a kód budem ešte postupne refaktorovať pre zvýšenie jeho kvality, čitateľnosti a zjednodušeniu údržby v budúcnosti.

4.1 Návrh obrazoviek a tried

Selection screen

Pri vytváraní ABAP reportov je možné využívať dynprá *selection screen* 3.2, ktoré slúžia na špecifikáciu parametrov, s ktorými bude program pracovať. V tomto prípade využijem *selection screen* na to, aby som od užívateľa získala informácie, pre aký *company code*[1], a pre aké reportovacie obdobie chce program spustiť. Ďalej potrebujem zaškrtávacie políčko *delete data*, ktoré zabezpečuje špecifickú funkcionalitu. Ak je toto pole zaškrtnuté, tak po spustení reportu sa nebudú manuálne zadávať hodnoty, ale budú zmazané všetky záznamy z databázovej tabuľky, ktoré vyhovujú zadaným výberovým kritériám.

O vizuálne rozloženie na *selection screen* sa programátor nemusí vôbec starať. Elementy dynpra sú totiž automaticky generované podľa ich definície v zdrojovom kóde, teda podľa parametrov *selection screen* a ich príslušných typov, v poradí ich definície a prípadne sa dajú zhlukovať do blokov a priraďovať k nim vysvetľujúce texty.

Hlavné dynpro

Po zadaní výberových kritérií, a ak pole *delete data* nie je zaškrtnuté, sa tok programu prenesie na hlavnú obrazovku, ktorá bude slúžiť na samotné manuálne zadávanie hodnôt. Keďže moja implementácia bude vychádzať z už existujúceho poľského ABAP reportu, využijem pre návrh jeho hlavnú obrazovku, ktorej rozmiestnenie elementov je rozumné a dostatočne intuitívne vzhľadom k funkcionalite reportu. Tento program využíva dynpro *general screen* 3.2, aby sa pre zobrazenie všetkých relevantných záznamov z databázy dal využiť špeciálny ovládací prvok *container*. Rozmiestnenie týchto kontajnerov, definované pomocou nástroja *Screen painter*, je vidno na obrázku B.1.

Kontajner v ľavej časti slúži k zadávaniu číselných hodnôt. Napravo hore je kontajner, ktorý pozostáva zo záznamov so zaškrtávacími políčkami a ten pod ním obsahuje polia, v ktorých sa očakáva textová hodnota. Do ďalšieho kontajnera je možné napísať vysvetlenie k zadaným hodnotám a posledný slúži k zobrazeniu poučenia a podmienok. Pod ním sa nachádza zaškrtávacie políčko, ktorým sa potvrdzuje súhlas s danými podmienkami.

Diagram tried

Predtým než začnem so samotnou implementáciou, vytvorím si najskôr návrh tried a ich metód. Existujúci poľský program, z ktorého vychádzam, nie je objektovo-orientovaný. Jeho zdrojový kód pozostáva z *formov*, čo sú pomenované a opätovne volateľné podprogramy, no tento element jazyka ABAP je už zastaralý a pre nový vývoj ho nie je možné použiť [27]. Pre účely návrhu však využijem, čo mi tieto *formy* poskytujú a to je logické členenie programu. Navrhované metódy teda vychádzajú z názvov pôvodných *formov* a podľa ich významu a funkcionality ich zoskupujem do konkrétnych tried. Návrh tried v podobe UML diagramu je možné vidieť na obrázku A.1.

[1] Company code – kód firmy je definovaný v účtovníctve a predstavuje právne nezávislú firmu so samostatným účtovníctvom. Je reprezentovaný štvormiestnym alfanumerickým kódom [25].

4.2 Implementácia ABAP reportu

Na implementáciu ABAP reportu využívam SAP GUI na počítači so systémom Windows. Vzhľadom na to, že sa jedná o nový vývoj, ktorý vyžaduje vytváranie rôznych elementov, využívam *Object Navigator* – transakcia *SE80*, ktorý sprístupňuje všetky potrebné nástroje *ABAP Workbench*. V príslušnom balíku pre maďarský vývoj vytváram samostatný ABAP report `GLO_FIN_HU_VAT_DECL`, ktorý tvorí jadro implementácie. Keďže viem, že budem tento report jednotkovo testovať a budem využívať *selection screen*, pre oba tieto účely vytváram samostatný *include*, ktoré sa využívajú pre lepšie členenie zdrojového kódu na logické celky.

Selection screen 1000

Prvá obrazovka, ktorá sa zobrazí hneď po spustení reportu, teda štandardné *selection dynpro* s číslom 1000, slúži k špecifikácii kritérií, podľa ktorých sú užívateľovi zobrazené relevantné záznamy z databázy pre manuálne vyplnenie hodnôt výkazu DPH. Je vygenerovaná automaticky na základe kľúčových slov v *include* `GLO_FIN_HU_VAT_DECL_SCREEN`.

Všetky elementy obrazovky sú obalené do bloku, aby spolu ako celok mohli obsahovať nadpis *Selection Criteria*. Tento nadpis, ako aj všetky ďalšie využívané texty, sa nachádza v *Text Elementoch* daného reportu, aby bolo možné texty zobrazované užívateľovi prekladať do iných jazykov. Prvé dva elementy úvodnej obrazovky sú povinné, čo zabezpečí kľúčové slovo `OBLIGATORY` na konci ich definície.

Prvým parametrom je *Company Code* dátového typu `bukrs`, ktorý predstavuje kód firmy, a pomocou `MEMORY ID` je na tento parameter naviazaný zoznam všetkých dostupných *company kódov*, ktoré sú užívateľovi zobrazené po vyžiadaní nápovedy klávesou F4.

Druhým elementom je `SELECT-OPTION` typu `vatdate`, čo je dátový typ využívaný pre dátum pri vykazovaní daní z tabuľky `BKPF`, ktorá uchováva dátové typy pre položky hlavičiek účtovných dokladov. Použitie `SELECT-OPTION` umožňuje zadávať rôzne kombinácie rozmedzí dátumov, no vďaka vlastnosti `NO-EXTENSION` sa dá obmedziť tak, aby bolo možné zadať jediný dátumový interval, čo je práve v tejto situácii žiaduce.

Posledný element *selection screen* je parameter `AS CHECKBOX`, teda zaškrtávacie políčko, ktoré užívateľ zaškrtne vtedy, pokiaľ chce iba zmazať relevantné záznamy z databázy.

Všetky elementy *selection screen* sú definované ako globálne a je teda možné k ich hodnotám pristupovať z akejkoľvek časti hlavného reportu. Rozloženie elementov na *selection screen* je vidno obrázku 4.1.

Obr. 4.1: Selection screen programu GLO_FIN_HU_VAT_DECL

4.2.1 Spôsob uloženia dát v databáze

Po zadaní výberových kritérií na *selection screen* už môže program vyfiltrovať relevantné záznamy. Pred popisom samotnej implementácie fungovania reportu je nutné objasniť, akým spôsobom sú dáta ukladané.

Štruktúra databázových tabuliek, ktoré budú uchovávať všetky dáta zobrazované a upravované pomocou reportu `GLO_FIN_HU_VAT_DECL`, zostáva v zásade identická s poľským riešením, z ktorého celá implementácia vychádza. Potrebné dátové elementy sú skopírované a ich popisy upravené tak, aby reflektovali vývoj v maďarskom balíku `GLO_FIN_IS_VAT_HU`. Ten obsahuje nasledujúce databázové tabuľky viažuce sa k danému reportu.

SND_HU_VAT_DECL

Táto aplikačná tabuľka slúži zákazníkom k udržiavaniu ich vlastných záznamov a konkrétnych hodnôt pre všetky reportovacie obdobia a *company kódy*. Po vyplnení kritérií na *selection screen* sú z nej vyberané relevantné záznamy.

- `MANDT` – číslo klienta v rámci daného systému
- `BUKRS` – *company kód*
- `DECLARATION_DATE` – dátum, ku ktorému sa hodnota vykazuje
- `FIELD_NAME` – vypovedá o presnom umiestnení hodnoty v daňovom výkaze a má špecifický zákonne definovaný formát
- `FIELD_AMOUNT` – zaznamenáva číselnú hodnotu
- `FIELD_CURRENCY` – viaže sa k poľu `FIELD_AMOUNT` a reprezentuje menu zadanej hodnoty
- `FIELD_FLAG` – uchováva položky, ktorých hodnota sa vyjadruje príznakom
- `FIELD_VALUE` – uchováva fulltextové hodnoty

Prvé štyri polia tvoria primárny kľúč tabuľky, ostatné sú nepovinné a udržujú vždy len tie hodnoty, ktoré sú relevantné pre daný typ manuálne zadávanej hodnoty.

Pre každé reportovacie obdobie existujú ešte dva špeciálne záznamy, ktoré sa odlišujú v poli `FIELD_NAME`. Prvým je záznam `JUSTIFICATION` slúžiaci k fulltextovému vysvetleniu k zadaným hodnotám. Druhý záznam `CONFIRMATION` uchováva hodnotu zaškrtávacieho políčka pre súhlas s podmienkami.

SNI_HU_VAT_FLD

Tabuľka `SNI_HU_VAT_FLD` je systémová, čo znamená, že záznamy v nej sú preddefinované a dodávané spoločnosťou SAP. Zákazníci nemajú možnosť tieto dáta upravovať [26]. Záznamy tejto tabuľky hovoria o tom, aké položky a akého typu sú úradmi vyžadované vypĺňať pre konkrétne reportovacie obdobia. Primárny kľúč tejto tabuľky tvoria jej prvé dve polia.

- `FIELD_NAME` – toto pole je zhodné názvom aj typom s poľom v tabuľke `SND_HU_VAT_DECL` a práve cez toto pole je následne možné záznamy spájať

- `BEGDA`, `ENDDA` – počiatočný a koncový dátum – ohraničujú interval, počas ktorého je zákonom vyžadované vyplnenie danej hodnoty vo výkaze
- `FIELD_TYPE` – obsahuje položku typu `char` dĺžky 1, ktorá môže nadobúdať hodnoty `A`, ak je položka číselná, `X` v prípade položky typu príznak a `T` pre fulltextovú položku.
- `IS_EDITABLE` – zaškrtnutie tohto poľa znamená, že je hodnota na výstupe editovateľná, v opačnom prípade je bunka deaktivovaná
- `IS_HIDDEN` – v prípade zaškrtnutia tohto políčka nebude záznam na výstupe zobrazený
- `TAX_ITEM_FIELD` – udržuje špeciálny a jednoznačný identifikátor položky výkazu DPH
- `IS_REVERSED` – určuje, či má byť hodnota započítaná s opačným znamienkom

SNI_HU_VAT_FLDT

`SNI_HU_VAT_FLDT` je špeciálna textová tabuľka pre `SNI_HU_VAT_FLD`, ktorá s ňou má rovnaký primárny kľúč, ktorý je ale rozšírený ešte o pole `LANGU`, aby mohli byť textové popisy, ktoré uchováva v poli `FIELD_DESCRIPT`, preložené do viacerých jazykov a teda zobrazené podľa užívateľských nastavení. Táto tabuľka slúži teda k uchovaniu širšieho vysvetľujúceho popisu k položkám daňového výkazu, ktoré je potrebné manuálne zadať.

4.2.2 Triedy zabezpečujúce zobrazenie dát na dynpre

Výpis dát z databázy a ich zobrazovanie na dynpre podľa užívateľských požiadaviek je bežným vývojom v jazyku ABAP. Pre zjednodušenie práce poskytuje SAP svojim programátorom vstavanú knižnicu *SAP List Viewer – ALV*, ktorá automaticky spracováva a formátuje výstup a umožňuje tiež pokročilú funkcionalitu pri práci nad dátami [1]. Pre objektovo-orientovaný vývoj sa využíva trieda `CL_GUI_ALV_GRID`, ktorej metódy sú použité aj pri mojej implementácii.

V tejto podkapitole je popísaný princíp fungovania metód tých tried, ktoré majú na zodpovednosť prezentovanie dát na dynpre. Využitie ostatných tried bude spomenuté v ďalších podkapitolách. UML diagram všetkých tried programu je na obrázku A.2.

Keďže sú triedy vytvárané lokálne, majú prefix `lcl` – `local class`. Pre každú triedu je definované rozhranie obsahujúce metódy, ktoré triedy implementujú vo svojej `public` sekcii. Tieto lokálne definované rozhrania majú prefix `lif` – `local interface`.

Trieda lcl_outtab_setup

Trieda zabezpečuje získanie relevantných záznamov z databázy, ktorých hodnoty majú byť manuálne zadané užívateľom. Implementuje metódy rozhrania `lif_outtab_setup`, ktoré okrem potrebných metód definuje aj dátové typy pre globálne tabuľky, ktoré nesú záznamy zobrazované užívateľovi na hlavnej obrazovke. Metódy tohto rozhrania sú:

- `create_outtab` – zapisuje do tabuľky `gt_outtab`, ktorá uchováva záznamy pre zadávanie číselných hodnôt
- `create_outtab_flag` – zapisuje do tabuľky `gt_outtab_flag` pre zaškrtávacie políčka
- `create_outtab_text` – zapisuje do tabuľky `gt_outtab_text` slúžiacej pre uchovanie fulltextových hodnôt

Všetky tri metódy majú ako vstupné parametre *company kód* a reportovacie obdobie zadané na *selection screen* a tieto parametre využívajú na získanie odpovedajúcich záznamov z tabuľky `SND_HU_VAT_DECL` v spojení so `SNI_HU_VAT_FLD` pomocou SQL ABAP príkazov. Dané tri metódy sa odlišujú iba v tom, ktoré polia z tabuliek vyberajú a to práve na základe hodnoty v poli `FIELD_TYPE` tabuľky `SNI_HU_VAT_FLD`, ktoré určuje typ manuálne zadávanej hodnoty na výstupe.

Trieda lcl_grid_setup

Táto trieda má na starosť formátovanie výstupu a implementuje rozhranie `lif_grid_setup`. Výstupné dáta sú ukladané v troch tabuľkách s prefixom `gt_outtab`, ktoré sú definované v rámci tela programu. Tieto tabuľky sú definované ako globálne z dôvodu, aby ich bolo možné upravovať užívateľom z hlavného dynpra, na ktorom sú zobrazené.

Tri metódy s prefixom `set_grid_fcat` volajú funkčný modul `LVC_FIELDCATALOG_MERGE`. Ako vstupný parameter mu predávajú očakávané štruktúry stĺpcov, ktoré sú uložené v konštantách tejto triedy s prefixom `report_fields_struc` – každá z troch funkcií využíva príslušnú konštantu s ohľadom na typ zadávanej hodnoty. Funkčný modul dostáva tiež ako `CHANGING` parameter príslušný triedny atribút s prefixom `mt_fcat` a po jeho dokončení teda dané atribúty obsahujú informáciu, aké stĺpce majú byť užívateľovi zobrazené.

Metódy s prefixom `set_grid_edit` upravujú záznamy v tabuľkách s prefixom `gt_outtab` a to konkrétne ich pole `HANDLE_STYLE`. To určuje pre všetky typy hodnôt maximálnu dĺžku, akú je užívateľovi umožnené zadať. Následne je nad objektami *ALV gridov* vyvolaná metóda `set_ready_for_input`, ktorá umožní do nich zapisovať.

Trieda lcl_prepare_container

Trieda `lcl_prepare_container`, ktorá implementuje rozhranie `lif_prepare_container`, má na zodpovednosť zobrazenie elementov hlavného dynpra typu *container*. Využíva k tomu metódy vyššie spomenutých tried `lcl_grid_setup` a `lcl_outtab_setup`. Mimo iných táto trieda implementuje aj nasledujúce metódy:

- `d0100_prepare_container_vat`
- `d0200_prepare_container_flags`
- `d0500_prepare_container_text`

Dané tri metódy pracujú takmer totožne, ale prihliadajú na to, pre aké typy hodnôt bude kontajner slúžiť a majú na starosť jeho vytvorenie. Do príslušného triedneho atribútu vytvoria objekt triedy `cl_gui_custom_container`. Ten je priradený ako rodičovský objekt ďalšiemu objektu triedy `cl_gui_alv_grid`, ktorý je tiež uchovaný v triednom atribúte a spolu teda zabezpečujú, aby záznamy na obrazovke boli správne zobrazené v kontajneroch v mriežkovanej štruktúre. Následne je zavolaná príslušná metóda triedy `lcl_outtab_setup` pre vytvorenie výstupov a tiež metódy triedy `lcl_grid_setup` pre nastavenie formátu výstupu. Nad triednym atribútom, ktorý obsahuje *ALV grid*, sú následne volané metódy triedy `cl_gui_alv_grid`, ktoré zabezpečujú nastavenie zobrazenia tabuľky pre užívateľa a zaregistrovanie udalostí reagujúcich na zmenu bunky či potvrdenie v danom gride. Do lokálnej premennej `event_receiver` je vytvorená inštancia triedy `lcl_grid_event_receiver` a pre tento objekt je príkazom `SET HANDLER` nastavené, že po užívateľskej akcii v danom gride je zavolaná triedna metóda `handle_data_changed`

triedy `lcl_grid_event_receiver`. Tá nastavuje potrebné parametre, aby sa z iných častí programu dalo na užívateľskú akciu reagovať.

Medzi ďalšie metódy, ktoré trieda `lcl_prepare_container` implementuje v rámci svojho rozhrania, patria:

- `d0300_prepare_container_txt` – po vytvorení potrebného kontajneru je tiež vytvorený objekt triedy `cl_gui_textedit` a sú nad ním volané základné metódy tak, aby bol pripravený pre zadanie užívateľského vstupu. Potom je vyvolaná metóda `read_database_text_editor` triedy `lcl_process_database`, ktorá vyhľadá záznam `JUSTIFICATION` z databázy pre zadané parametre na *selection screen*. Ak záznam existuje, zapíše ho do textového editoru na obrazovke.
- `d0400_prepare_container_txt_ro` – v tejto metóde je tiež vytvorený kontajner a textový editor, no tento je nastavený na mód, ktorý neumožňuje zápis a je v ňom zobrazený text so zákonnými podmienkami, ktoré treba potvrdiť v zaškrtávacom políčku pod týmto kontajnerom
- `set_layout` – metóda nastavuje položky triednej štruktúry typu `lvc_s_layo`, ktorá definuje rozloženie *ALV gridu*

4.2.3 Logika toku programu

Reporting Events, teda udalosti ABAP reportu, sú bloky udalostí, ktoré spúšťa *ABAP runtime environment*. Po spustení programu je volaná séria udalostí jedna po druhej a v programe sa teda dajú definovať bloky, ktoré na tieto udalosti reagujú [1]:

- `INITIALIZATION` - táto reportovacia udalosť je volaná ako úplne prvá spomedzi ostatných hneď pri spustení ABAP reportu. Pokiaľ má ale program definovanú *selection screen*, blok `INITIALIZATION` je vykonaný po spracovaní kódu, ktorý vytvára *selection screen*, ale ešte pred zobrazením tejto obrazovky užívateľovi. Preto sa tento blok využíva na priradenie počiatočných hodnôt parametrov na obrazovke. Avšak tento blok je vykonaný iba jeden krát za spustenie programu a pokiaľ sa z inej obrazovky vráti tok programu späť na *selection screen*, je treba mať na pamäti, že tento blok už nebude znova vykonaný [1]. V programe `GLO_FIN_HU_VAT_DECL` je využitá táto časť na vytvorenie inštancií tried, aby sa s danými objektami dalo pracovať v ďalších blokoch udalostí tohto reportu.
- `START-OF-SELECTION` - tento blok udalostí je volaný hneď po `INITIALIZATION`. Pokiaľ program obsahuje *selection screen*, práve tento kód je vykonaný po dokončení spracovávania udalostí na danej obrazovke. Vo všeobecnosti do tohto bloku patrí kód zabezpečujúci výber potrebných dát z databázy. Program `GLO_FIN_HU_VAT_DECL` v tomto bloku volá metódy triedy `lcl_vat_declaration` – metóda `auth_check` overuje autorizáciu pre zadaný *company kód* a metóda `lock_program` uzamkne prístup k databázovej tabuľke `SND_HU_VAT_DECL`, z ktorej program bude hodnoty nie len načítavať, ale aj do nej zapisovať. Potom sa tok programu rozdeľuje podľa zaškrtávacieho políčka pre vymazanie dát nasledovne:
  - Pokiaľ ho užívateľ zaškrtol, tak je volaná metóda `delete_data`, v ktorej sú pomocou ABAP SQL príkazu `DELETE` zmazané všetky záznamy z databázovej tabuľky `SND_HU_VAT_DECL`, ktoré sa zhodujú v *company kóde* a súčasne ich

`declaration date` sa zhoduje s hornou hranicou intervalu zadaného v parametroch na *selection screen.*

– Pokiaľ políčko zostane nezaškrtnuté, je vykonaný príkaz `CALL SCREEN 100`, ktorým je zavolané hlavné dynpro číslo 100.

Ako kontext napovedá, medzi blokmi udalostí `INITIALIZATION` a `START-OF-SELECTION` sú vykonané ešte udalosti *selection screen*, medzi ktoré patrí aj `AT SELECTION-SCREEN`. Bežne sa využíva na overenie užívateľských vstupov, keďže je volaná po ich zadaní. Práve táto udalosť je v programe využitá pre overenie platnosti intervalu zadaného užívateľom metódou `date_range_validation` a prípadne je vyvolaná výnimka, ktorá na neplatný interval dátumu upozorňuje. Ostatné polia na obrazovke sú v tom čase deaktivované, až kým užívateľ nezadá v danom parametri platnú hodnotu.

Po vykonaní všetkých udalostí ABAP reportu aj *selection screen* už plne preberá zodpovednosť za tok programu hlavné dynpro a riadi ho pomocou svojich dialógových modulov.

General dynpro 100

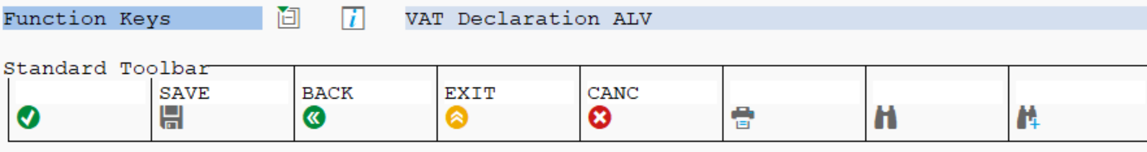
Tento typ obrazovky je manuálne vytvorený ako komponenta daného programu. Pomocou nástroja *Screen Painter* 3.2 je presne definované rozloženie elementov obrazovky na výstupe a je zachytené na obrázku B.1. Na obrazovku je vložených päť elementov *container* a jedno zaškrtávacie políčko `CHECK` s textovým popisom. Význam týchto elementov bol popísaný pri ich návrhu 4.1. Každý element má svoj špecifický názov, ktorý je jedinečný v rámci celého programu a to umožňuje komunikáciu a predávanie hodnôt medzi obrazovkou a zdrojovým kódom reportu.

V rámci nástroja *Screen painter* je tiež definovaná *Flow logika* dynpra, teda dialógové moduly v blokoch PBO a PAI. Dynpro 100 obsahuje nasledujúce:

PBO modul d0100_set_status

Modul `d0100_set_status` obsahuje iba 2 príkazy. Jedným z nich je `SET TITLEBAR 'D0100'`, pričom `D0100` je kompomentou programu typu `GUI Title` a jej popis *Hungary VAT Declaration Input* je nastavený ako nadpis daného dynpra, ktorý je zobrazený v ľavej hornej časti obrazovky.

Príkaz `SET PF-STATUS 'D0100'` nastavuje `GUI Status` dynpra, teda aké ikony sa zobrazia v menu v hornej časti. `GUI Status D0100` obsahuje 4 ikony a kliknutie na ne je potrebné zaznamenať. Sú im teda priradené jedinečné funkčné kódy, ktoré je vidno na obrázku 4.2.



Obr. 4.2: Priradenie funkčných kódov ikonám v menu pre prenesenie informácie o kliknutí na ne do tela programu.

Dynpro 100 obsahuje element `G_OKCODE` a v časti globálnej definície premenných programu sa nachádza premenná s identickým názvom, ktorá je typu `sy-ucomm`, teda reprezen-

tuje užívateľský príkaz. Po kliknutí na niektorú z ikon sa jej funkčný kód prenesie z dynpra do rovnomennej premennej v programe, ku ktorej je možné pristupovať z akejkoľvek časti programu a podľa jej aktuálnej hodnoty teda reagovať na užívateľské akcie.

PBO modul d0100_prepare_container

Ako každý PBO dialógový modul, aj tento je prevádzaný pred každým zobrazením dynpra. Hneď na začiatku je však pomocou `boolean` premennej zabezpečené, aby vykonávanie metód v tele modulu prebehlo iba raz, práve po spustení programu po zadaní parametrov na *selection screen*. V tomto module sú totiž volané metódy triedy `lcl_prepare_container` 4.2.2 a to všetkých päť metód, ktoré majú na zodpovednosť vytvorenie a pripravenie kontajnerov pre zobrazenie na dynpre.

PBO modul d0100_prepare_checkbox_terms

Tento modul volá metódu `read_database_checkbox_terms` triedy `lcl_process_database`, ktorá načíta z tabuľky `SND_HU_VAT_DECL` špeciálnu hodnotu `CONFIRMATION` pre zvolený *company kód* a reportovacie obdobie. Táto hodnota je uložená do globálnej premennej programu `check`, ktorá je súčasne zdieľaná s elementom dynpra 100 – zaškrtávacím políčkom pre potvrdenie podmienok. Tento záznam umožňuje, aby podmienky zostali potvrdené a nebolo nutné ich zaškrtávať pri každom spustení programu.

PAI modul d0100_fcode

Blok PAI modulov je volaný po zaznamenaní nejakej užívateľskej akcie. Dynpro 100 obsahuje jediný PAI modul `d0100_fcode`, ktorý volá metódu rovnakého názvu nad objektom triedy `lcl_user_action_handler`. V danej metóde sa tok delí na dve vetvy podľa hodnoty funkčného kódu v `G_OKCODE`. Je dôležité spomenúť, že vždy, keď sa pracuje s hodnotou funkčného kódu, je dobrým zvykom uložiť hodnotu do lokálnej premennej a hodnotu v globálnej `G_OKCODE` zmazať. Následne sa v kóde pristupuje k tejto lokálnej hodnote, čo slúži ako prevencia voči tomu, aby sa z inej časti kódu dalo získať už neaktuálnu hodnotu, ktorá by inak zostala uložená v `G_OKCODE` až do jej prepísania inou užívateľskou akciou.

Pokiaľ užívateľ stlačil jednu z ikon pre návrat späť, je najskôr z obrazovky načítaný text v textovom editore, ktorý uchováva záznam `JUSTIFICATION` a pokiaľ bol tento záznam užívateľom upravený, sú ešte nastavené premenné udržujúce informáciu o tom, že na obrazovke nastala zmena a treba aktualizovať databázu.

Potom je zavolaná metóda `d0100_exit`, ktorá overí, či databáza bola aktualizovaná a či sa na obrazovke medzitým neudiali neuložené zmeny v iných kontajneroch. Ak áno, tak metóda `confirm_on_exit` triedy `lcl_call_popup` zavolá vyskakovacie okno, v ktorom sa užívateľa opýta, či naozaj chce obrazovku opustiť. Ak zaklikne, že áno, alebo v prípade, že sa stav databázy zhoduje so stavom na obrazovke, tak pomocou metódy `leave_screen` sú uvoľnené všetky objekty, ktoré boli vytvorené pre toto dynpro. Vykonávanie programu sa vráti na *selection screen*, odkiaľ môže byť hlavné dynpro znova spustené s novými parametrami pre reportovacie obdobie a *company kód*.

Ak užívateľ klikol na ikonu pre uloženie zmien, sú zo všetkých troch gridov aj textového editoru pre `JUSTIFICATION` získané aktuálne dáta na dynpre. Ak nie je na obrazovke zaškrtnuté políčko `CONFIRMATION`, metóda `confirm_on_save` triedy `lcl_call_popup` sa vyvolaním vyskakovacieho okna postará o zistenie, či si užívateľ napriek tomu želá zmeny uložiť. Ak áno, alebo aj ak boli podmienky zaškrtnuté, je nad objektom triedy `lcl_process_database`

zavolaná metóda `update_database`. V nej sa pomocou ABAP SQL príkazov vytvorí lokálna tabuľka rovnakého typu ako je `SND_HU_VAT_DECL`, ktorej záznamy presne reflektujú hodnoty z dynpra – všetky číselné, príznakové a fulltextové záznamy, ale tiež `JUSTIFICATION` a `CONFIRMATION`.

Záznamy lokálnej tabuľky sú prenesené do databázovej tabuľky `SND_HU_VAT_DECL` pomocou príkazu `MODIFY`, ktorý pridáva nové záznamy a upravuje už existujúce, ak sa ich hodnoty zmenili. Zabezpečí však, že nedôjde k strate záznamov z iných reportovacích období a *company kódov*.

Hlavnú obrazovku programu `GLO_FIN_HU_VAT_DECL` aj so zadanými vzorovými hodnotami, je možné vidieť na obrázku B.2.

4.2.4 Priebeh implementácie s využitím refaktorizácie

Ako už bolo spomenuté, jadro tohto ABAP reportu vychádza zo staršej poľskej implementácie. V tomto prípade je jedným z hlavných dôvodov k čiastočnému využitiu už existujúceho riešenia práve práca s objektami kontajnerov a gridov. Pre správne nastavenie zobrazenia na dynpre je nutné po vytvorení objektov príslušných tried nad nimi volať pomerne veľké množstvo metód, aby boli na dynpre nielen zobrazené v požadovanom formáte, ale aby do zvolených polí bol umožnený aj zápis. Pokus o písanie podobného kódu nanovo by zabral programátorovi veľké množstvo času, no nepriniesol by žiadny nový výsledok. Kopírovanie kódu z iných reportov, ktoré pracujú s objektami tried knižnice ALV, je bežnou praktikou, ktorá šetrí čas a eliminuje množstvo chýb.

Pri začiatku implementácie bolo najdôležitejšie zbaviť sa zastaralých *formov* a vytvoriť objektovo orientovaný report, ktorý bude spustiteľný a funkčný. Z toho stavu sa už potom dá aplikovať refaktorizácia – meniť štruktúru návrhu, vytvárať pomocné metódy, či premiestňovať už existujúce do iných tried. Na začiatku som teda definovala triedy podľa prvotného návrhu A.1 a telá pôvodných *formov* som presunula do rovnomenných metód. Niektoré dáta, ktoré boli využívané iba v rámci jednotlivých *formov*, boli definované lokálne, ale všetky ostatné dáta využívané naprieč viacerými *formami* boli globálne definované na začiatku reportu, čo som najskôr tiež tak nechala.

V tej chvíli bol report spustiteľný a nastala fáza zmeny funkcionality, kedy bolo treba najmä identifikovať a odstrániť kód, ktorý zabezpečoval špecifickú poľskú funkcionalitu, ktorá pri vývoji pre Maďarsko nebola žiadúca. Už na prvý pohľad boli niektoré *formy* pre tento report irelevantné, a tie som nezahrnula ani do návrhu tried, no v iných *formoch* sa nachádzali podmienené vetvenia, ktoré bolo nutné prispôsobiť, prípadne odstrániť.

Po úprave kódu tak, aby jeho funkcionalita zodpovedala tej požadovanej, som začala s fázou skutočnej refaktorizácie, teda zmenou vnútornej štruktúry programu bez toho, aby sa zmenila funkcionalita navonok.

Ako je uvedené v sekcii *Ako refaktorovať* 2.6.2, pri refaktorizácii je potrebné mať nástroje na rýchlu a pravidelnú kontrolu, či sa najnovšími zmenami nezaviedli do kódu chyby. Jedným z takýchto nástrojov môže byť aj kontrola syntaxe, ktorá napríklad odhalí, ak pri presúvaní atribútu zostala nejaká metóda, ktorá k nemu už nemá prístup, no potrebuje ho. Pre kontrolu, či sa po zmenách nenachádzajú v programe chyby, ktoré by ovplyvňovali výsledky výpočtov, či záznamy v databáze, sú však najvhodnejšie unit testy. Pre dve triedy, ktoré v tomto programe pracujú nad databázou, som si teda už v tejto fáze vytvorila jednotkové testy, ktoré ale postupne tiež prešli úpravami popri tom, ako sa menil kód, ktorý testujú. Pre metódy, ktoré zabezpečujú zobrazenie na dynpre, ktorými je tento ABAP re-

port tvorený do veľkej časti, je účinnou a rýchlou metódou pre odhalenie novozavedených chýb obyčajné spustenie programu. Vďaka tomu, že na *selection screen* sa dajú uložiť zadávané parametre do variantov, je táto kontrola správnej manipulácie nad objektami dynpra naozaj rýchla.

Pri samotnej refaktorizácii som začala tým, že som dáta z globálnej definície programu premiestňovala do *public* sekcie tried, ku ktorým boli najviac relevantné. Následne som aplikovala viaceré refaktorizácie z knihy Refaktoring [8], napríklad ako zapúzdriť dátové položky, čím sa obmedzuje priama práca nad nimi – tá je umožnená výlučne pomocou prístupových metód. Niektoré dátové položky či metódy tried som premenovávala a presúvala do iných tried, ak sa časom ukázalo, že je to tak vhodnejšie. Lokálne premenné v určitých metódach som nahradzovala volaním vhodne pomenovaných metód, čo zjednodušuje čitateľnosť kódu a pri prechádzaní niektorých logických vetvení som podmienky upravila takým spôsobom, ktorý je na prvý pohľad jednoduchší na pochopenie.

Prínosy refaktorizácie

Keď porovnám prvotné riešenie, ktoré spĺňalo očakávané požiadavky, a ktoré som dosiahla prekopírovaním *formov* do príslušných metód tried, je vo výslednom refaktorizovanom riešení štruktúra lepšie členená a prehľadnejšia, čo znamená lepšiu čitateľnosť kódu a teda jednoduchšiu udržovateľnosť v budúcnosti.

Veľkým prínosom refaktorizácie v tomto prípade bolo tiež to, že vedie k podrobnej analýze a pochopeniu kódu. Vďaka tomu sa ukázalo, že aj po zmazaní špecifickej funkcionality pre Poľsko stále existovalo pomerne veľké množstvo mŕtveho kódu, teda miesta programu, ku ktorým sa vo vykonávaní prakticky nedá dostať. Vo viacerých *formoch* sa tiež na ich začiatku nachádzalo overovanie určitej podmienky, ktorá ale bola overená tesne pred volaním tohto *formu*, čo predstavovalo zbytočné duplicity v kóde a teda som ich odstránila. Niektoré zložené podmienky obsahovali tiež overenia, ktoré v každom prípade vyšli pravdivo a pôsobili teda zbytočne zavádzajúco. Okrem odstránenia viacerých pasáží kódu som tiež odstránila dva PAI moduly hlavného dynpra, ktorých funkciou bolo reagovať na užívateľské akcie, no to sa mi zdalo vhodnejšie spracovať priamo v metóde `d0100_fcode`. Pri prechádzaní zdrojových kódov bolo viditeľné, že jeho autor tiež využíval nejaké existujúce riešenie a nechal tam pasáže, ktoré nastavovali parametre aktuálne nevyužívanej funkcionality. Jedným z takých prípadov bolo použitie a nastavenie štruktúry `disvariant`, ktorá sa využíva pre uloženie rôznych variantov rozloženia ALV. To pri momentálne požadovanej funkcionalite reportu nie je využiteľné a teda ponechať v programe tieto časti kódu by išlo proti princípom agilného vývoja.

Pri procese refaktorizácie a súčasnom testovaní programu som tiež narazila na viaceré chyby, ktoré sa mi podarilo opraviť v mojej implementácii, a tiež som o nich informovala tím zodpovedný za poľský vývoj. Jednou z nich bola chýbajúca podmienka, ktorá overovala, či už bol kontajner na obrazovke raz vytvorený a to v určitých situáciach viedlo k opätovnému vytvoreniu objektu a teda prepísaniu zadaných hodnôt na dynpre.

Ďalšou chybou bola nesprávna práca s tabuľkou `gt_outtab`, ktorá udržuje hodnoty na dynpre zadané užívateľom. V situácii, keď užívateľ klikol na ikonu ukončenia programu ho vyskakovacie okno informovalo o tom, že sa na obrazovke nachádzajú neuložené zmeny a opýtalo sa, či si naozaj želá program ukončiť. V prípade, ak nechcel report opustiť, boli neuložené hodnoty z dynpra aj tak zahodené a pregenerované z databázy, ktorá sa nachádzala v poslednom uloženom stave. Pôvodne tiež nebolo možné z databázy vymazať textový záznam `JUSTIFICATION`, pretože uloženie tohto záznamu do databázy bolo prevádzané iba

v prípade, ak sa v textovom poli nachádzala nejaká hodnota. Keď ju teda užívateľ zmazal, bola táto bunka pri ukladaní do databázy ignorovaná a zostávala v nej neaktuálna hodnota.

4.3 Priebeh testovania

Testovanie tohto ABAP reportu prebiehalo viacerými spôsobmi. Ako už bolo spomenuté, metódy, ktoré zabezpečujú prácu s elementami dynpra sa dajú najjednoduchšie otestovať samotným spustením programu. Keďže tento typ reportu neprevádza žiadne výpočty nad zadanými dátami, sú jednotkové testy zamerané najmä na testovanie práce s databázou. Po dokončení vývoja boli prevádzané ešte *FIT* a *SAT testy*, ktoré sú bežnou súčasťou pri dodávaní nového softwaru v SAP.

4.3.1 Unit testy

Jednotkové testy sú implementované v *include* `GLO_FIN_HU_VAT_DECL_TEST` a pri ich implementácii využívam *ABAP Unit* 3.3. Daný *include* obsahuje štyri testovacie triedy a vlastnosti všetkých sú nastavené na `RISK LEVEL HARMLESS`, pretože nespôsobujú zmeny v databáze a `DURATION SHORT`, vzhľadom na krátku dobu ich vykonávania.

Všetky testovacie triedy využívajú metódu `setup`, v ktorej je vytvorená inštancia príslušnej produkčnej triedy do premennej `cut`, nad ktorou sú následne volané produkčné metódy. Trieda `ltcl_prepare_container` testuje metódu, ktorá nastavuje spôsob rozloženia pri zobrazení na dynpre a trieda `ltcl_grid_event_receiver` testuje metódu pre nastavenie príznaku pri zaznamenaní zmeny na obrazovke. Testovacie triedy `ltcl_vat_declaration` a `ltcl_process_database` slúžia obe k testovaniu správnej práce nad databázou. Metódy `class_setup` a `class_teardown` využívajú *ABAP SQL Test Double Framework* 3.3.1 k odstráneniu závislostí na databázových tabuľkách.

Testovacia trieda ltcl_vat_declaration

Testovacie metódy s prefixom `date_range_validation` testujú rovnomennú produkčnú metódu. Overujú, či v prípade zadania platného intervalu je premenná `sy-subrc` nulová a naopak v prípade neplatného intervalu musí byť zachytená výnimka `daterange_not_valid`.

Testovacie metódy s prefixom `delete_data` testujú metódu, ktorá je zodpovedaná za vymazanie záznamov z databázovej tabuľky `SND_HU_VAT_DECL`. V metódach si teda vytvorím lokálne tabuľky s dátami pre rôzne testovacie prípady a metóda `insert_test_data` volaná nad triednym atribútom `sql_test_environment` ich vloží do náhradnej databázovej tabuľky. Následne je zavolaná produkčná metóda `delete_data`. Pomocou ABAP SQL príkazov získam všetky záznamy náhradnej tabuľky `SND_HU_VAT_DECL` do lokálnej tabuľky. Metóda `assert_equals` následne medzi sebou porovná získanú tabuľku s ďalšou lokálne vytvorenou, do ktorej vložím záznamy, ktoré po vykonaní metódy `delete_data` s danými parametrami očakávam.

Testovacia trieda ltcl_process_database

Táto trieda obsahuje dve testovacie metódy, ktoré overujú správne načítavanie z databázy textového vysvetlenia v zázname `JUSTIFICATION` a príznaku pre potvrdenie podmienok v zázname `CONFIRMATION`.

Ďalej sa v tejto testovacej triede nachádza niekoľko metód s prefixom `update_database`, ktoré predstavujú rôzne testovacie scenáre nad rôznymi testovacími dátami a tabuľkami.

Overujú správny zápis do databázy všetkých typov záznamov na dynpre – číselných hodnôt, príznakov, textových záznamov a tiež špeciálnych hodnôt `CONFIRMATION` a `JUSTIFICATION`.

Štatistiky pokrytia jednotkovými testami

Nástroj pre jednotkové testovanie *ABAP Unit* poskytuje tiež súhrnné štatistiky o pokrytí kódu. V rámci zaručenia kvality dodávaných softwarov je interne stanovená minimálna hranica pokrytia kódu unit testami 15%, v opačnom prípade systém neumožní uvoľniť nový vývoj na zákaznícke systémy. Túto hranicu sa celkovo podarilo prekročiť aj napriek tomu, že viacero produkčných tried nemá svoje metódy pokryté unit testami. Veľkú časť príkazov tvorí volanie metód ALV knižnice, ktorých unit testovanie by postrádalo zmysel. Taktiež nebolo možné implementovať unit testy pre metódy, ktorých vykonávanie je závislé na globálne definovaných dátach program. Tie teda nie je možné v testoch nahradiť, no museli zostať globálne práve z dôvodu, aby k nim malo prístup hlavné dynpro a umožnilo užívateľovi ich editáciu. V tabuľke 4.1 je možné vidieť súhrnné výsledky dosiahnuté pre celý ABAP report, ale aj pokrytie unit testami produkčných tried, ktoré pracujú s databázou.

	Pokrytie vetiev	Pokrytie volaní metód	Pokrytie príkazov
Report `GLO_FIN_HU_VAT_DECL`	22.01 %	28.81 %	24.69 %
Trieda `LCL_PROCESS_DATABASE`	95.45 %	100.00 %	98.53 %
Trieda `LCL_VAT_DECLARATION`	66.67 %	75.00 %	56.00 %

Tabuľka 4.1: Percentuálne pokrytie kódu unit testami

4.3.2 Testy FIT a SAT

Do tejto chvíle boli všetky testy prevádzané iba v rámci vývojového systému. Po dokončení a uvoľnení nových zmien nasleduje fáza, v ktorej je funkcionalita programu overovaná na testovacích systémoch, ktoré simulujú podmienky u zákazníkov. K tomu sa v SAP využívajú *FIT testy (Functional Integration Test)*, ktoré overujú splnenie funkčných požiadaviek na dátach reálneho sveta.

Testovacie prípady pre FIT testy vytvára vždy samotný vývojár, keďže má najlepší prehľad o tom, ktoré situácie treba otestovať, na aké špeciálne prípady nezabudnúť, kde by prípadne mohli nastať problémy a ako celkovo vytvoriť testovacie prípady tak, aby pokrývali celú funkcionalitu programu. V prípade reportu `GLO_FIN_HU_VAT_DECL` som do testovacích prípadov okrem bežného spustenia a zadania hodnôt zahrnula ďalšie scenáre, ktoré simulovali situácie vyvolávajúce vyskakovacie okná pre potvrdenie užívateľom, ďalej tiež viacnásobné spustenie programu s rovnakými parametrami, kedy sa očakáva načítanie hodnôt, ktoré boli naposledy uložené, či scenáre overujúce zmazávanie dát. Testy samozrejme zahŕňajú zadávanie všetkých typov hodnôt, vrátane špeciálnych záznamov pre vysvetlenie a súhlas s podmienkami.

Po úspešnom ukončení *FIT testov* nasledovali *SAT testy (Solution Acceptance Test)*, ktoré sú prevádzané osobami mimo vývojový tím, bežne koncovými zákazníkmi, a sústreďujú sa na overenie, či daná aplikácia spĺňa skutočné požiadavky trhu. *SAT testy* pre tento vývoj prevádzal maďarský *LPM (Local Product Manager)*, ktorý je expertom na daňové zákony pre danú krajinu, a akceptoval dodanú funkcionalitu bez požiadaviek na dodatočné zmeny.

Kapitola 5

Exportovanie tabuliek do formátu CSV

Funkcionalita programu `GLO_FIN_HU_VAT_DECL` je rozšírená o možnosť exportovania manuálne zadaných hodnôt do súborov vo formáte *CSV (Comma-separated values)*, v ktorom sú hodnoty oddelené čiarkami, prípadne iným oddeľovačom. Táto dodatočná funkcionalita, popísaná v nasledujúcej kapitole, je implementovaná agilnou metodikou *Test Driven Development* 2.4.

Navrhovaná funkcionalita

Pokiaľ by export do CSV bol skutočnou zákazníckou požiadavkou, pravdepodobne by cieľom bolo vygenerovať pre zadaný *company kód* a reportovacie obdobie jeden súbor, ktorý by obsahoval všetky tri typy hodnôt, vrátane dvoch špeciálnych `JUSTIFICATON` a `CONFIRMATION`. Keďže je ale táto časť implementovaná pre účely demonštrácie metodiky vývoja riadeného testami, bude program poskytovať možnosť výberu, ktorú z troch tabuliek na hlavnej obrazovke s ich príslušnými hodnotami exportovať. Následne program vygeneruje pre zvolené tabuľky samostatné CSV súbory. Toto riešenie je v porovnaní s prvým navrhovaným implementačne rozsiahlejšie a tiež zaujímavejšie z pohľadu demonštrácie na objektovo-orientovanom vývoji.

Export tabuliek bude umožnený pridaním špeciálnej ikony do *GUI Statusu* hlavného programu, ktorá zavolá ďalšie dynpro. Na ňom užívateľ zaškrtne, ktoré tabuľky chce exportovať a do akých súborov, a zároveň zadá, aký oddeľovač medzi hodnotami použiť.

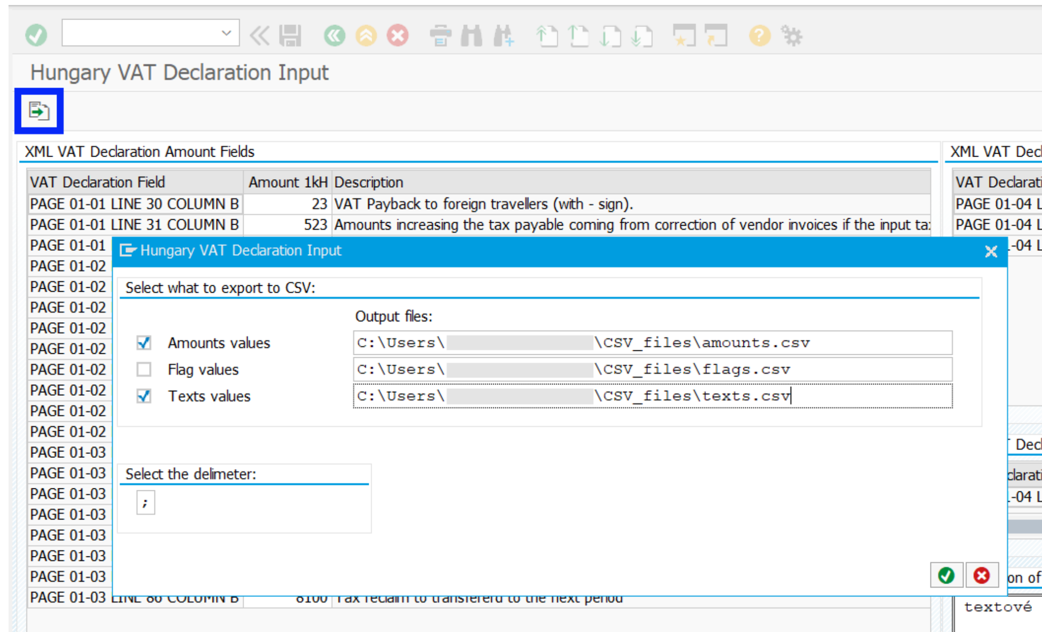
5.1 Implementácia CSV exportu

Rozšírenie o export do formátu CSV nepatrí do plánovaného vývoja a nebude teda dodávané zákazníkom SAP. Z toho dôvodu je program `GLO_FIN_HU_VAT_DECL` vrátane všetkých jeho komponent a tiež potrebné databázové tabuľky skopírované do lokálneho balíka. V ňom sú programy, ich *includy* a databázové tabuľky pomenované rovnako, no je k nim pridaná predpona 'Z', ktorá zabezpečuje, že objekty nebudú dodané zákazníkom.

Ešte pred implementáciou samotných tried, ktoré budú zabezpečovať správne formátovanie CSV, je potrebné upraviť hlavné telo programu.

Do existujúceho *GUI Statusu* hlavného programu je pridaná špeciálna ikona a je jej priradený funkčný kód `EXPORT`. Po kliknutí na túto ikonu je zavolané nové dynpro 200, vytvorené pomocou nástroja *Screen painter*, ktoré obsahuje tri zaškrtávacie políčka pre výber

tabuliek a textové polia pre zadanie výstupného súboru. Ďalší element slúži pre zadanie jednoznakového oddeľovača. Ikony pre potvrdenie alebo zrušenie sú pridané pomocou *GUI Statusu*, ktorý sa viaže na toto dynpro. Elementy tejto obrazovky, vrátane ikony, ktorá ju vyvolá, je možné vidieť na obrázku 5.1.



Obr. 5.1: Dynpro 200 pre výber kritérií CSV exportu a ikona, ktorá ho vyvolá

Pre všetky elementy obrazovky sú v globálnej definícii programu vytvorené rovnomenné premenné príslušného dátového typu, aby sa dalo pristupovať k hodnotám zadaným užívateľom. Zdrojový kód zabezpečujúci volanie tejto obrazovky je pridaný do metódy `d0100_fcode`, ktorá je volaná z PAI modulu hlavnej obrazovky, a reaguje na užívateľské akcie. Do logického vetvenia podľa funkčného kódu stlačenej ikony je pridaná nová vetva pre CSV export. V tej je overené, či sa dáta na obrazovke zhodujú s tými v databáze a ak nie, informačné okno užívateľa upozorní, že pred exportom musí hodnoty uložiť.

Ak sa na obrazovke nenachádzajú žiadne neuložené zmeny, je zavolané dynpro 200 a riadenie programu je predané jeho PBO modulom. V module `set_status_0200` je nastavený príslušný *GUI Status* a funkčný modul `set_default_values_0200` predvyplní cesty k výstupným súborom a nastaví predvolený oddeľovač na znak ';', ktoré sú samozrejme editovateľné.

Ak užívateľ potvrdí exportovanie súborov, tak podľa zaškrtnutých políčok, ktoré tabuľky sa majú vyexportovať, sú v PAI funkčnom module `user_command_0200` volané metódy príslušných tried, ktoré CSV export zabezpečujú.

Po vyexportovaní súborov, alebo v prípade, ak užívateľ export nepotvrdil, je tok programu prevedený naspäť na hlavné dynpro, kde užívateľ môže pokračovať vo vykonávaní ďalších akcií.

Triedy a ich rozhrania

Definície a implementácie tried, ktoré zabezpečujú export do CSV, sa vrátane rozhraní, ktoré implementujú, nachádzajú v *include* `ZGLO_FIN_HU_VAT_DECL_CSV_EX`.

Sú to nasledujúce tri triedy, pričom každá implementuje rozhranie `lif_csv_export` a k tomu ešte nasledovné:

- `lcl_export_amounts` – implementuje rozhranie `lif_process_table_amounts`
- `lcl_export_flags` – implementuje rozhranie `lif_process_table_flags`
- `lcl_export_texts` – implementuje rozhranie `lif_process_table_texts`

Každá z týchto tried obsahuje nasledovné `private` atribúty a vyššie spomenuté rozhrania k nim poskytujú prístupové metódy.

- `m_file_name` – uchováva názov a cestu k súboru
- `mt_table_for_csv` – udržuje tabuľku, ktorá obsahuje záznamy pre zvolené reportovacie kritériá a v nich len polia, ktoré sú relevantné k príslušnému typu (číselné/príznakové/fulltextové). Dátový typ záznamov tejto tabuľky je teda pre každú triedu mierne odlišný.
- `mt_csv_table` – obsahuje záznamy typu `string`, ktoré sú v podobe pripravenej na exportovanie do súboru

Tri rozhrania s prefixom `lif_process_table` definujú vlastné dátové typy pre záznamy tabuliek `mt_table_for_csv` tak, aby zodpovedali požadovaným hodnotám vo výstupnom súbore pre konkrétny typ manuálne zadaných hodnôt. Každé z týchto rozhraní definuje metódu `create_table_from_dtbs` a ich triedne implementácie zabezpečujú získanie záznamov z databázy. Cez pole primárneho kľúča `field_name` spájajú tabuľku `ZSND_HU_VAT_DECL`, ktorá obsahuje záznamy manuálne zadaných hodnôt a k nim viažúce sa *company kódy*, a tabuľku `ZSNI_HU_VAT_FLD`, ktorá obsahuje informáciu o tom, akého typu daná hodnota je a pre ktoré reportovacie obdobie je relevantná. Tak sú získané položky všetkých záznamov, ktoré majú byť exportované, a sú vložené do tabuľky `mt_table_for_csv`.

Metóda `create_csv_table` rozhrania `lif_process_table` túto tabuľku načíta a jej záznamy spojí do textovej štruktúry s príslušných oddeľovačom. Zabezpečuje tiež, že ak sa v niektorých hodnotách záznamu nachádza zvolený oddeľovač, je táto hodnota z oboch strán ohraničená úvodzovkami. Tabuľku obsahujúcu textové záznamy vloží do tabuľky `mt_csv_table`.

Metóda `export_real_file` rozhrania `lif_csv_export` volá funkciu `GUI_DOWNLOAD`, ktorá zabezpečuje vytvorenie súboru a jeho stiahnutie do klientského počítača. Získaná tabuľka s textovými záznamami vo formáte CSV a príslušná cesta k výstupnému súboru sú zadané ako vstupné parametre tejto funkcie.

5.2 Test Driven Development a jeho výsledky

Keďže bol tento vývoj riadený metodikou *Test Driven Development* 2.4, tak vždy pred pridaním novej funkcionality boli napísané jednotkové testy overujúce správnosť jej implementácie. Unit testy, ktoré pokrývajú triedy zabezpečujúce export do CSV, sa nachádzajú v *include* `ZGLO_FIN_HU_VAT_DECL_TEST_CSV`.

V súlade so zásadami agilného vývoja a metodikou TDD, nielen zdrojové kódy, ale aj samotné triedy a metódy unit testov boli popri vývoji postupne upravované a podľa potreby refaktorizované. Výsledná implementácia obsahuje tri testovacie triedy:

- `ltcl_csv_amounts` – testuje metódy triedy `lcl_export_amounts`
- `ltcl_csv_flags` – testuje metódy triedy `lcl_export_flags`
- `ltcl_csv_texts` – testuje metódy triedy `lcl_export_texts`

Podľa postupu popísaného v podkapitole 3.3.1 sú vo všetkých testovacích triedach odstránené závislosti na skutočných databázových tabuľkách a v metódach `setup` je testovacie prostredie naplnené vzorovými záznamami tabuľky `ZSNI_HU_VAT_FLD`.

Všetky tri testovacie triedy obsahujú metódy typu `setup` a `teardown` a tiež metódu `filename`, ktorá overuje prístupové metódy k atribútu uchovávajúcemu cestu k súboru. V metódach `create_table_from_dtbs` a `create_csv_table`, ktoré testujú rovnomenné produkčné metódy, sú najskôr vytvorené ukážkové záznamy tabuľky `ZSND_HU_VAT_DECL`, ktoré sú vložené do testovacieho prostredia. Následne sú vytvorené záznamy tabuliek príslušných dátových typov s očakávanými hodnotami. Po zavolaní produkčnej metódy nad inštanciou `cut` danej testovanej triedy, sú prístupovými metódami získané vytvorené tabuľky a pomocou metódy `assert_equals` je skutočná tabuľka porovnaná s očakávanou. V oboch metódach sú okrem bežných hodnôt overené aj prázdne hodnoty a tiež záznamy, ktoré obsahujú zvolený oddeľovač pre CSV export.

Štatistiky pokrytia kódu unit testami

Rozhranie `lif_csv_export` definuje jednu metódu, ktorá nie je jednotkovo testovaná. Tou je práve metóda `export_real_file`, keďže vytváranie súborov počas tohto typu testovania nie je žiadúce. Z toho dôvodu nie je pokrytie unit testami sto percentné.

Pokrytie volaní metód je pre všetky tri triedy rovnaké a to 8 otestovaných produkčných metód z celkových 9, čo predstavuje bezmála 89 %.

Štatistiky pokrytia unit testami pre jednotlivé triedy je možné vidieť v tabuľke 5.1, ktorá zobrazuje pokrytie jednotlivých vetiev, a v tabuľke 5.2, zobrazujúcej pokrytie príkazov.

	Celkovo	Vykonané	Nevykonané	Percentuálne
Trieda `lcl_amounts`	15	11	4	73,33 %
Trieda `lcl_flags`	15	11	4	73,33 %
Trieda `lcl_texts`	17	13	4	76,47 %

Tabuľka 5.1: Pokrytie vetiev jednotkovými testami

	Celkovo	Vykonané	Nevykonané	Percentuálne
Trieda `lcl_amounts`	31	26	5	83,87 %
Trieda `lcl_flags`	31	26	5	83,87 %
Trieda `lcl_texts`	34	29	5	85,29 %

Tabuľka 5.2: Pokrytie príkazov jednotkovými testami

Kapitola 6

Zhodnotenie vývoja s využitím objektovej orientácie a agilných prístupov

Jazyk ABAP je hybridným programovacím jazykom a podporuje vývoj podľa dvoch programovacích modelov. Objektovo-orientovaný model programovania, ktorý využíva *ABAP Objekty*, je postavený na triedach a ich rozhraniach. Triedy koncepčne nahrádzajú klasické programy a modularizácia je zabezpečená pomocou ich metód. Procedurálny model programovania je založený na klasických blokoch spracovania. Modularizáciu teda implementujú funkčné moduly, podprogramy (*formy*), bloky udalostí či dialógové moduly. Funkčné moduly sú procedúry definované v špeciálnych funkčných skupinách, a následne môžu byť volané z ABAP programov. Oba tieto modely umožňujú interoperabilitu, teda klasické bloky spracovania môžu volať metódy tried a naopak [28].

Hybridná nátura tohto jazyka je daná historicky, keďže ABAP bol pôvodne procedurálnym programovacím jazykom. *ABAP Objekty* boli do jazyka integrované vo verzii 4.5 [30] v roku 1999 [33], pričom musela zostať zaistená kompatibilita s procedurálnym modelom a možnosť znovupoužiteľnosti pôvodných programov a klasických blokov spracovania.

Použitie *ABAP Objektov* namiesto klasického procedurálneho prístupu je výhodnejšie z viacerých dôvodov. Objektová orientácia umožňuje zapúzdrenie dát, čím je možné obmedziť k nim prístup, umožňuje znovupoužiteľnosť tried vďaka využitiu dedičnosti, objekty môžu byť adresované pomocou samotných rozhraní a *ABAP Objekty* tiež zjednodušujú implementáciu programov, ktorých tok je riadený udalosťami. Možnosti práce s *ABAP Objektami* sa s novými verziami jazyka postupne rozširovali. Dnes je už interným pravidlom pri vývoji SAP softwaru využívať objekty všade, kde je to len možné [29]. Využitie samotných *ABAP Objektov* je doporučené dokonca aj vtedy, ak nie je využívaný objektovo-orientovaný model, keďže objekty poskytujú určité vlastnosti, ktoré procedurálne programovanie neumožňuje. Pri práci s dynprami či volaní funkčných modulov sa ale použitiu procedurálnych elementov jazyka ABAP vyhnúť nedá [28].

Zavedenie agilných prístupov do praxe pri vývoji SAP produktov prišlo v roku 2009, kedy bola v tejto spoločnosti predstavená všeobecná filozofia *Lean Developmentu* a metodika *Scrum*, ktorou sa postupne začal vývoj riadiť. Napriek zavedeným zmenám v celkovej organizácii práce vývojových tímov a prerozdeleniu zodpovedností, ktoré priniesli významné zlepšenia, spoločnosť dospela k uvedomeniu, že samotné zavedenie *Scrumu* nestačí. Táto metodika zabezpečuje hladké a priebežné dodávanie softwaru po menších častiach, no ne-

zaručuje už vnútornú kvalitu softwaru. To z dlhodobého hľadiska predstavuje veľké problémy, keďže zavádzanie novej funkcionality a tiež oprava chýb sa stáva čoraz viac časovo náročná [13].

V roku 2010 si spoločnosť SAP definovala *'Agile Software Engineering'* - *ASE*, čo predstavuje sadu hodnôt, princípov a technických zručností, ktoré musia vývojári ovládať, aby boli schopní úspešne pracovať v agilnom kontexte. *ASE* sa teda zameriava na technické postupy vývoja, ktoré zabezpečujú správne fungovanie v Scrume. V rámci *ASE* programu poskytovala spoločnosť SAP svojim vývojárom školenia moderných agilných prístupov, medzi ktoré patrili párové programovanie, jednotkové testovanie vo všeobecnosti, ale aj s využitím vývoja riadeného testami, súvisiacu izoláciu testov, či princípy refaktorizácie. Zavedenie týchto praktík prinieslo preukázateľné benefity vo výsledných návrhoch aplikácií, zvýšenie kvality na všetkých leveloch, aj stabilnejší kód. Znížil sa tiež čas nového vývoja, ale hlavne čas a množstvo námahy potrebnej pre údržbu softwarov, čo malo významný pozitívny dopad na fungovanie aj celkový úspech tejto spoločnosti [13].

Keďže program `GLO_FIN_HU_VAT_DECL` je objektovo-orientovaný, oproti pôvodnému poľskému programu, z ktorého implementácia vychádza, disponuje viacerými užitočnými vlastnosťami, ktoré vychádzajú z vyššie spomenutých výhod objektovo-orientovaného modelu oproti tomu procedurálnemu. Nie len samotná zmena štruktúry programu, ale aj následná refaktorizácia tried a metód spolu prinášajú vyššiu kvalitu, lepší návrh a hlavne jednoduchšiu údržbu v budúcnosti.

Implementovanie exportu do CSV metodikou *Test Driven Development* jednoznačne prinieslo vyššie pokrytie kódu jednotkovými testami, ako by tomu bolo bez jej využitia, a to aj napriek tomu, že pri tomto type funkcionality je veľké pokrytie kódu unit testami obzvlášť vhodné. Počas implementácie bolo očividné, o koľko jednoduchšie je implementovať kód takým spôsobom, aby bolo možné ho jednotkovo testovať, práve pri využití metodiky vývoja riadenom testami.

Pri spätnom ohľade sa javí, že ak by boli agilné princípy uplatnené do extrému, pokrytie jednotkovými testami základného programu by teoreticky mohlo byť ešte vyššie, aj keď pravdepodobne nie významne, vzhľadom na rozsah kódov pracujúcich nad elementami obrazovky. Vysoké pokrytie unit testami pri programoch, ktoré prevádzajú veľké množstvo výpočtov, má nepopierateľnú hodnotu. V tomto prípade by však neprirodzená snaha o zmenu štruktúry programu kvôli vyššiemu pokrytiu iných metód, než tých pracujúcich nad databázou, bola veľmi pravdepodobne kontraproduktívna a v praxi by sa teda neprevádzala.

Možnosť exportovania do formátu CSV bola implementovaná metodikou vývoja riadeného testami a funkčné požiadavky boli teda definované tak, aby boli čo najvhodnejšie pre ukážkovú demonštráciu tejto metodiky. Takáto požiadavka by sa však v budúcnosti potenciálne mohla stať reálnou legislatívnou požiadavkou. V takom prípade by výsledný súbor pravdepodobne musel obsahovať inú kombináciu hodnôt, teda zahrnúť ďalšie špecifické polia. Pri exportovaní by zadanie názvu výstupného súboru muselo byť implementované tak, aby ho bolo možné zadať pomocou prieskumníka súborov.

S tým sa prirodzene naskytuje aj možná požiadavka na import hodnôt z existujúceho súboru, ktorý by teoreticky nemusel obsahovať všetky hodnoty záznamu, ale len primárny kľúč a relevantnú hodnotu. Na základe toho by program nie len načítal hodnoty v súbore, ale vyplnil aj zvyšné potrebné polia. Vtedy by záznamy boli úplné a bolo by možné ich zobraziť na dynpre aj s príslušným popisom a následne ich, po prípadnej modifikácii či bez nej, perzistentne uložiť z tabuliek dynpra do databázy.

Kapitola 7

Záver

Táto práca bola venovaná agilným metodikám vývoja softwaru, akým spôsobom ich pri vývoji aplikovať a aké benefity spolu s využitím refaktorizácie prinášajú. Predstavené bolo vývojové prostredie spoločnosti SAP, jeho nástroje a špecifické prvky ich vlastného jazyka ABAP pre vývoj SAP softwaru. V práci bol tiež predstavený nástroj, ktorý umožňuje implementáciu jednotkových testov pre objektovo-orientovaný vývoj v jazyku ABAP, a tiež princípy, ktoré treba dodržiavať pri aplikovaní metodiky *Test Driven Development*.

Pre implementáciu softwaru v jazyku ABAP, ktorý umožňuje manuálne zadanie určitých položiek výkazu DPH, boli použité časti zastaralého procedurálneho riešenia pre inú oblasť. Výsledný program je objektovo-orientovaný a bol implementovaný s využitím princípov refaktorizácie. Funkcionalita tohto softwaru bola vývojom riadeným testami rozšírená o exportovanie zadaných hodnôt do formátu CSV.

Štatistiky o pokrytí unit testami celého programu a jednotlivých tried sa dajú získať z troch rôznych uhlov – pokrytie vetiev, príkazov a volaní metód. Interne stanovená hranica požaduje vo všetkých smeroch pokrytie minimálne 15 %, čo hlavný program splňuje. Celkové pokrytie programu sa pohybuje medzi 22 až 29 %, pričom v prípade tried, ktoré zabezpečujú operácie nad databázou je toto pokrytie vyššie, v rozmedzí 56 - 100 %. Výsledný program prešiel aj ďalšími štandardnými procesmi testovania v spoločnosti SAP, na základe čoho bol uvoľnený pre distribúciu na zákaznícke systémy.

Triedy implementujúce exportovanie zadaných hodnôt do súboru vo formáte CSV, sú vďaka využitiu metodiky *Test Driven Development* pokryté jednotkovými testami do značnej miery. Pokrytie týchto tried sa pohybuje v rozmedzí 73 - 89 % v závislosti na type pokrytia.

V predposlednej kapitole boli zhodnotené výsledky vývoja vyplývajúce z objektovo-orientovanej implementácie a využitia agilných princípov pri vývoji v jazyku ABAP. Popísané boli tiež potenciálne možnosti rozšírení. Vzhľadom na charakteristiku požiadavky, ktorú tento program splňuje, sa v budúcnosti dá očakávať nutnosť rozšírenia funkcionality či prevedenia zmien vyplývajúcich z nových zákonov definujúcich formát výkazu DPH. Spôsob vývoja tohto programu však zabezpečuje, že akékoľvek budúce funkčné požiadavky by malo byť možné zapracovať pomerne jednoducho a efektívne.

Vypracovanie tejto bakalárskej práce mi prinieslo nie len možnosť aplikovať naštudované agilné prístupy pri reálnom vývoji, ale novou skúsenosťou pre mňa tiež bola zodpovednosť za implementovanie konkrétnej legislatívnej zmeny cez všetky fázy jej realizácie.

Literatúra

[1] Bandari, K. *Complete ABAP*. 1. vyd. Boston: SAP PRESS, 2016. ISBN 978-1-4932-1272-9.

[2] Beck, K. *Extrémní programování*. 1. vyd. Praha: Grada, 2002. ISBN 80-247-0300-9.

[3] Beck, K. *Programování řízené testy*. 1. vyd. Praha: Grada, 2004. ISBN 80-247-0901-5.

[4] Beck, K., Beedle, M. a Bennekum, A. van. *History: The agile manifesto* [online]. 2001 [cit. 2021-12-08]. Dostupné z: https://agilemanifesto.org/history.html.

[5] Beck, K., Beedle, M. a Bennekum, A. van. *Manifest agilního vývoje software* [online]. 2001 [cit. 2021-12-08]. Dostupné z: https://agilemanifesto.org/iso/cs/manifesto.html.

[6] Beck, K., Beedle, M. a Bennekum, A. van. *Principy stojící za Agilním Manifestem* [online]. 2001 [cit. 2021-12-09]. Dostupné z: http://agilemanifesto.org/iso/cs/principles.html.

[7] Fernandez, M. *Why Is Refactoring Your Code Important in Agile?* [online]. 2021 [cit. 2022-03-07]. Dostupné z: https://www.coscreen.co/blog/refactoring-your-code-in-agile/.

[8] Fowler, M. *Refaktoring: zlepšení existujícího kódu*. 1. vyd. Praha: Grada, 2003. ISBN ISBN 80-247-0299-1.

[9] Fowler, M. *On Pair Programming* [online]. 2020 [cit. 2021-12-13]. Dostupné z: https://martinfowler.com/articles/on-pair-programming.html.

[10] Gillis, A. S. *SAP ERP* [online]. 2020 [cit. 2022-02-26]. Dostupné z: https://searchsap.techtarget.com/definition/SAP.

[11] Grossmann, L. *Metodiky vývoja softvéru - Extrémne programovanie* [online]. 2020 [cit. 2021-12-13]. Dostupné z: https://www.itnetwork.sk/navrh/metodiky/extremne-programovanie.

[12] Herout, P. *Testování pro programátory*. 1. vyd. České Budějovice: Kopp, 2016. ISBN 978-80-7232-481-1.

[13] Heymann, J. *Introducing Agile Software Engineering in development* [online]. 2018 [cit. 2022-05-06]. Dostupné z: https://blogs.sap.com/2018/05/02/introducing-agile-software-engineering-in-development/.

[14] Heymann, J. a Hammer, T. *Writing Testable Code for ABAP* [online]. Máj 2018. Dostupné z: https://open.sap.com/courses/wtc1.

[15] Heřt, J. *Occamova břitva* [online]. 2007 [cit. 2021-12-09]. Dostupné z: https://www.sisyfos.cz/clanek/965-occamova-britva.

[16] Hodges, N. *5 Reasons Why You Should Be Coding With Interfaces* [online]. 2020 [cit. 2022-04-11]. Dostupné z: https://betterprogramming.pub/5-reasons-why-you-should-be-coding-with-interfaces-c6a92560afa1.

[17] Hodges, N. *Code Against Interfaces, Not Implementations* [online]. 2020 [cit. 2022-04-11]. Dostupné z: https://betterprogramming.pub/code-against-interfaces-not-implementations-37b30e7ab992.

[18] Kadlec, V. *Agilní programování: metodiky efektivního vývoje softwaru.* 1. vyd. Brno: Computer Press, 2004. ISBN 80-251-0342-0.

[19] Kosa, M. *WS Agilne: Extrémne Programovanie (XP)* [online]. 2011 [cit. 2021-12-13]. Dostupné z: https://www.websupport.sk/blog/2011/03/ws-agilne-extremne-programovanie-xp/#site-header.

[20] Krüger, N. *Agile Testing Methodology — 5 Examples for the Agile Tester* [online]. 2018 [cit. 2021-12-09]. Dostupné z: https://www.perforce.com/blog/alm/what-agile-testing-5-examples.

[21] Myslín, J. *Scrum: průvodce agilním vývojem softwaru.* 1. vyd. Brno: Computer press, 2016. ISBN 978-80-251-4650-7.

[22] Netinbag. *Co je softwarová krize?* [online]. [cit. 2021-12-09]. Dostupné z: https://www.netinbag.com/cs/internet/what-is-a-software-crisis.html.

[23] Pecinovský, R. *OOP: naučte se myslet a programovat objektově.* 1. vyd. Brno: Computer Press, 2010. ISBN 978-80-251-2126-9.

[24] SAP Help Portal. *ABAP SQL Test Double Framework* [online]. [cit. 2022-03-14]. Dostupné z: https://help.sap.com/viewer/c238d694b825421f940829321ffa326a/1809.000/en-US/1432ca1fc7b547d493f691cdd09245ae.html.

[25] SAP Help Portal. *Company Code* [online]. [cit. 2022-03-20]. Dostupné z: https://help.sap.com/viewer/37199b05ddc94111ad69bd1e851e0dff/EHP4_HRSP_E2/en-US/c5c2dc53b5ef424de10000000a174cb4.html.

[26] SAP SE. *ABAP - Keyword Documentation - Delivery Class for Database Tables* [online]. [cit. 2022-04-16]. Dostupné z: https://help.sap.com/doc/abapdocu_751_index_htm/7.51/en-us/abenddic_database_tables_delivery.htm.

[27] SAP SE. *ABAP - Keyword Documentation - FORM* [online]. [cit. 2022-04-07]. Dostupné z: https://help.sap.com/doc/abapdocu_751_index_htm/7.51/en-us/abapform.htm.

[28] SAP SE. *ABAP Objects as a Programming Model* [online]. [cit. 2022-05-06]. Dostupné z: https://help.sap.com/doc/abapdocu_751_index_htm/7.51/en-us/abenabap_obj_progr_model_guidl.htm.

[29] SAP SE. *ABAP Programming Models* [online]. [cit. 2022-05-06]. Dostupné z: http://saphelp.ucc.ovgu.de/NW750/EN/c3/225b5354f411d194a60000e8353423/content.htm.

[30] SAP SE. *Introducing ABAP Objects for Release 4.5* [online]. [cit. 2022-05-06]. Dostupné z: https://help.sap.com/doc/abapdocu_752_index_htm/7.52/en-US/abennews-40-objects.htm.

[31] SAP SE. *SAP Documentation - Example Transaction* [online]. [cit. 2022-03-02]. Dostupné z: https://help.sap.com/doc/saphelp_nw73ehp1/7.31.19/en-us/9f/db9ce935c111d1829f0000e829fbfe/frameset.htm.

[32] Scrum.org. *The Scrum Framework Poster* [online]. 2020 [cit. 2021-12-14]. Dostupné z: https://www.scrum.org/resources/scrum-framework-poster.

[33] STechies. *SAP Versions Release and History of Evolution* [online]. [cit. 2022-05-06]. Dostupné z: https://www.stechies.com/about-sap-erp-solution-different-versions/.

[34] Stone, S. *Code Refactoring Best Practices: When (and When Not) to Do It* [online]. 2018 [cit. 2022-03-07]. Dostupné z: https://www.altexsoft.com/blog/engineering/code-refactoring-best-practices-when-and-when-not-to-do-it/.

[35] Strihová, B. *Scrum jednoduše* [online]. 2020 [cit. 2021-12-13]. Dostupné z: https://medium.com/@barborastrihova/scrum-jednodu%C5%A1e-b55ff145bb35.

[36] Tutorials Point. *Software Development Life Cycle - Waterfall Model* [online]. [cit. 2021-12-09]. Dostupné z: https://www.tutorialspoint.com/sdlc/sdlc_waterfall_model.htm.

[37] Unadkat, J. *What is Test Driven Development (TDD) : Approach & Benefits* [online]. 2021 [cit. 2021-12-11]. Dostupné z: https://www.browserstack.com/guide/what-is-test-driven-development.

[38] Zaidi, R. *SAP ABAP Objects : A Practical Guide to the Basics and Beyond*. 1. vyd. Apress L. P., 2019. ISBN 9781484249642. Dostupné z: https://ebookcentral.proquest.com/lib/vutbrno/detail.action?docID=5906797.

Príloha A

Diagramy tried

lcl_prepare_container
+ d0100_prepare_container_vat( )
+ d0200_prepare_container_flags( )
+ d0500_prepare_container_text( )
+ d0300_prepare_container_txt( )
+ d0400_prepare_container_txt_ro( )
+ get_layout( )
+ get_outtab_text( )
+ get_outtab( )
+ get_outtab_flag( )

lcl_grid_setup
+ set_grid_fcat( )
+ set_grid_vari( )
+ set_grid_edit( )
+ set_grid_fcat_flag( )
+ set_grid_fcat_text( )
+ set_grid_edit_text( )

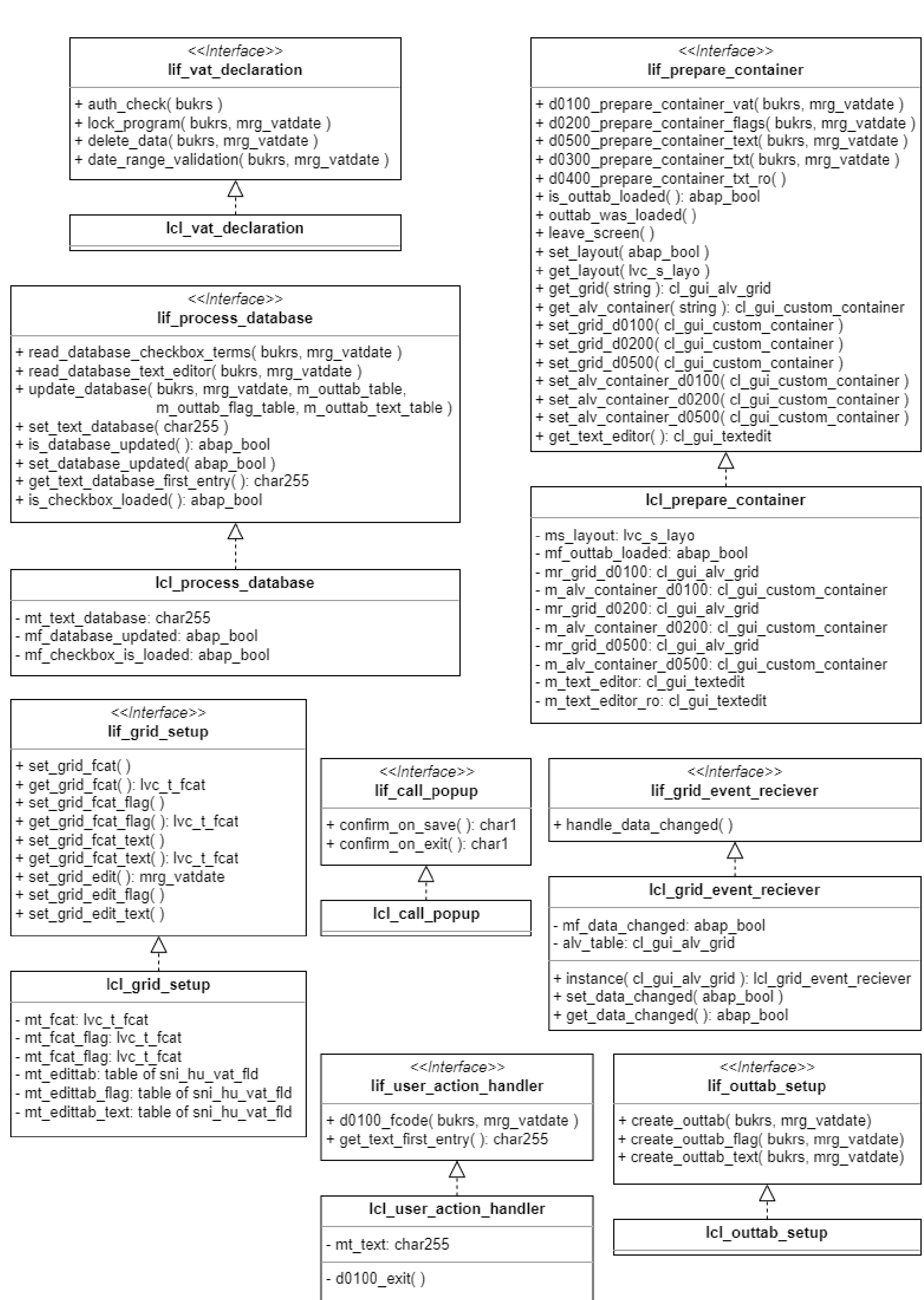
lcl_grid_event_receiver
+ instance( )
+ handle_data_changed( )
+ data_container_changed( )

lcl_user_action_handler
+ d0100_fcode( )
+ d0100_exit( )
+ call_popup_to_confirm_on_exit( )
+ call_popup_to_confirm_on_save( )

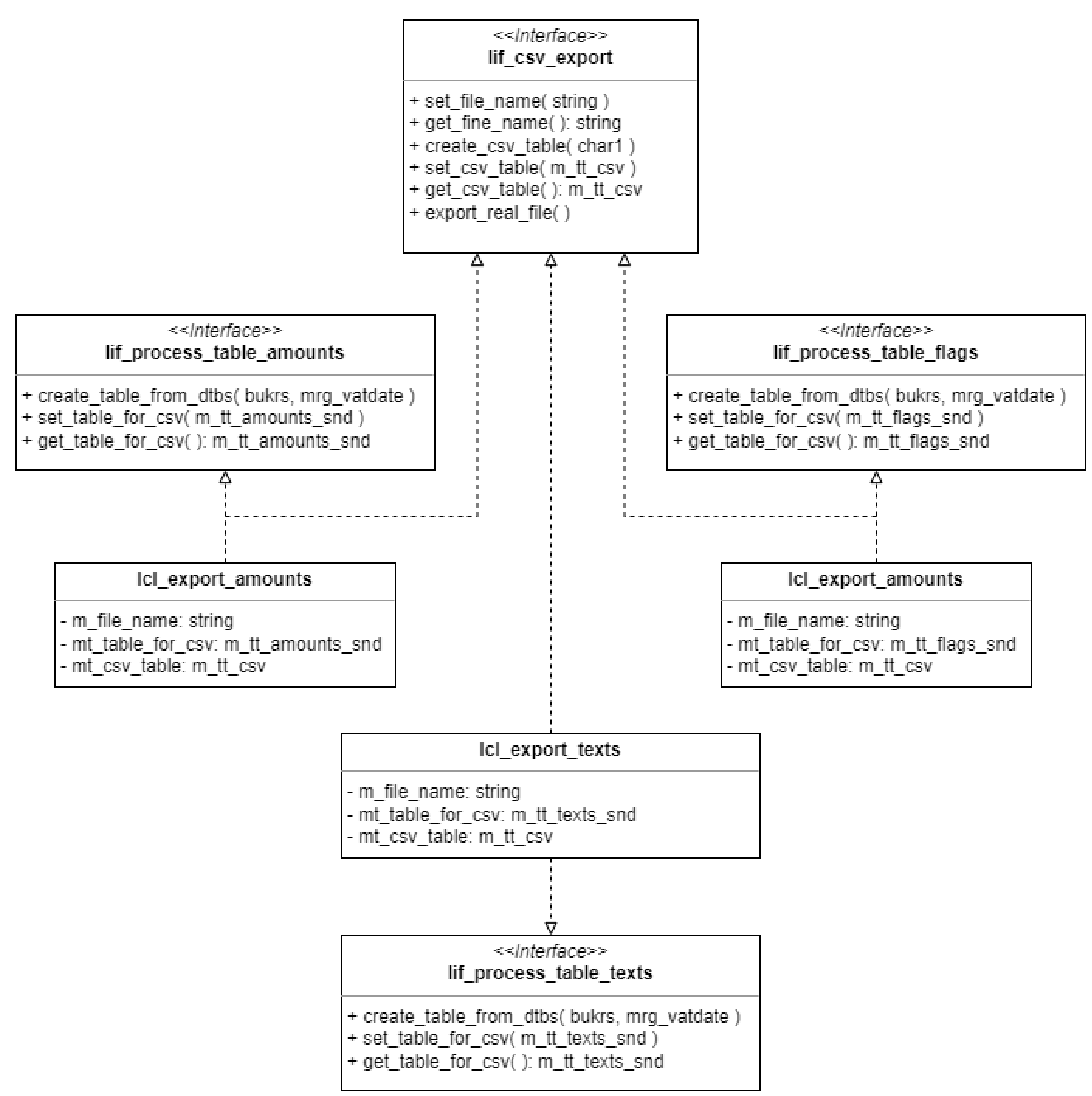
lcl_process_database
+ read_database_checkbox_terms( )
+ update_database( )
+ read_database_text_editor( )
+ delete_data( )

lcl_vat_declaration
+ auth_check( )
+ lock_program( )
+ date_range_validation( )

Obr. A.1: Diagram prvotného návrhu tried programu GLO_FIN_HU_VAT_DECL



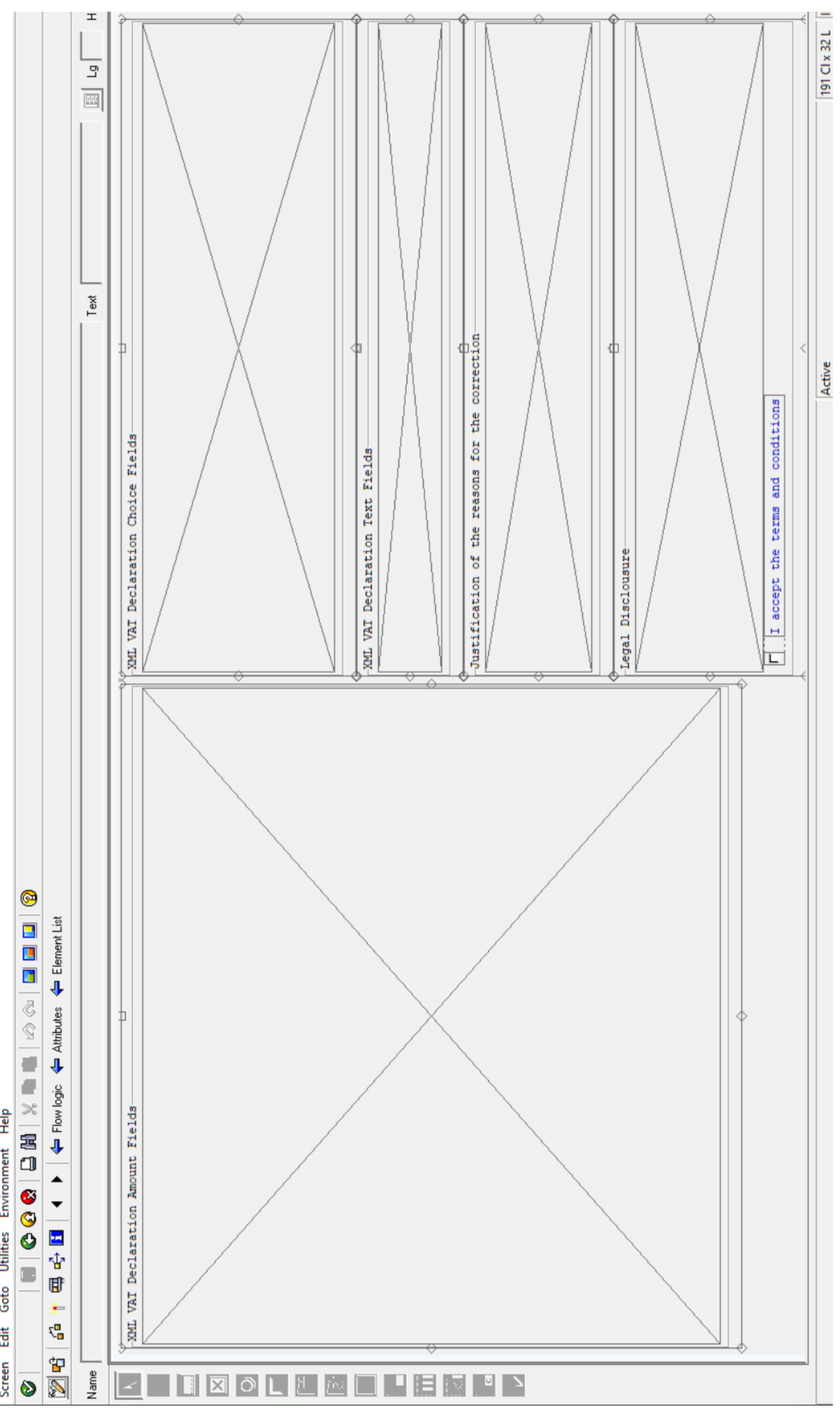
Obr. A.2: Finálny diagram tried programu GLO_FIN_HU_VAT_DECL



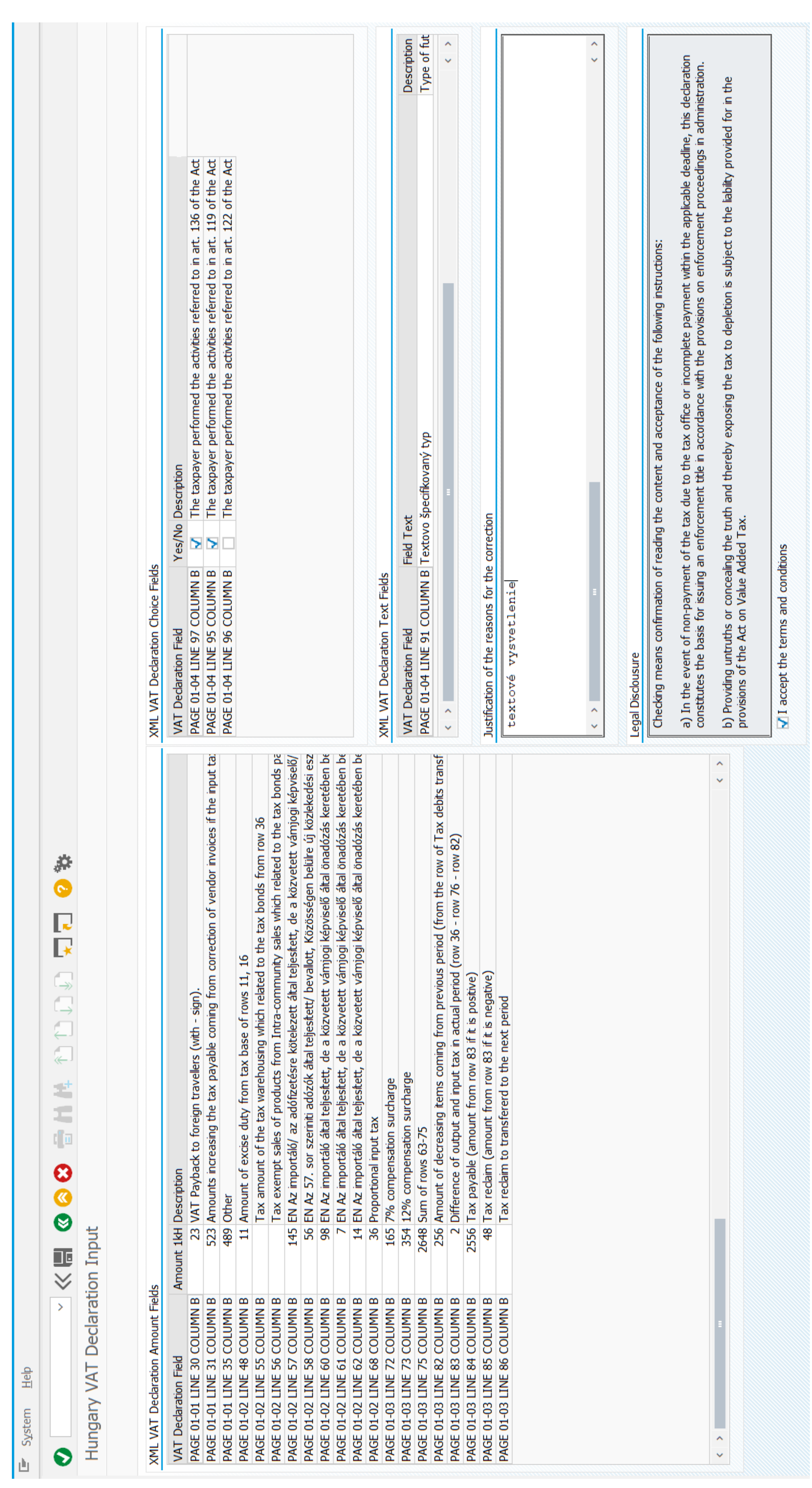
Obr. A.3: Finálny diagram tried umožňujúcich CSV export

Príloha B

Dynpro 100 ABAP reportu GLO_FIN_HU_VAT_DECL



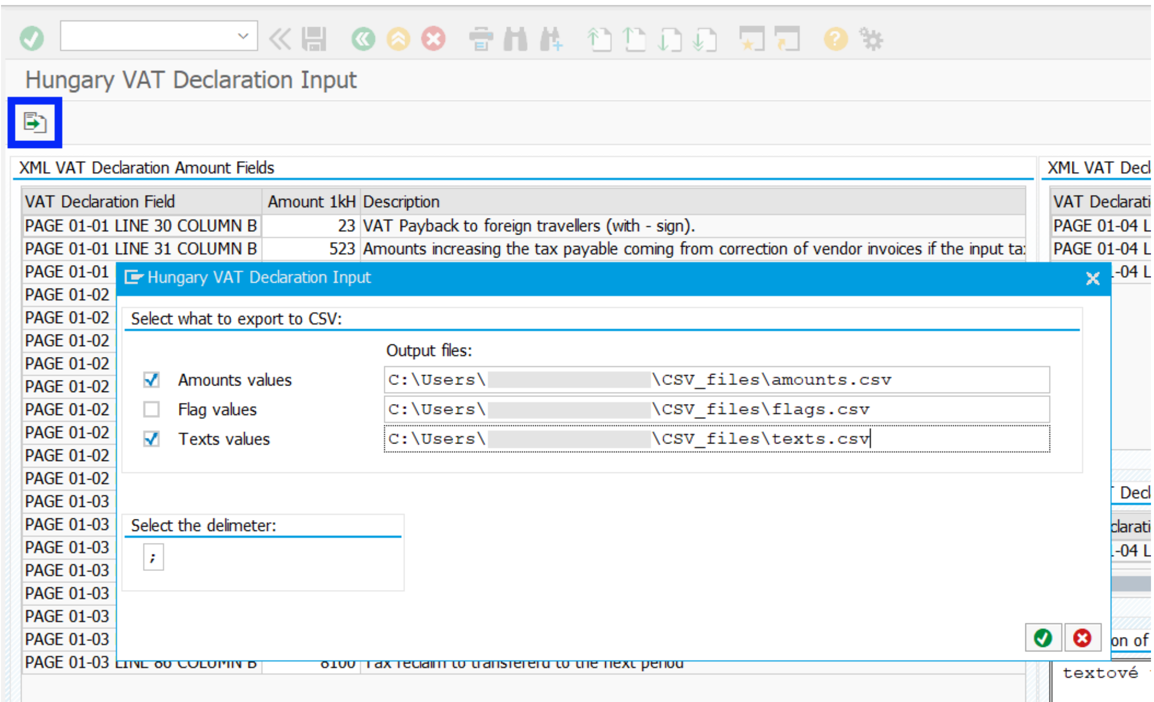
Obr. B.1: Rozloženie elementov hlavného dynpra reportu GLO_FIN_HU_VAT_DECL vytvorené nástrojom *Screen painter*



Obr. B.2: Hlavná obrazovka po spustení ABAP reportu GLO_FIN_HU_VAT_DECL

Príloha C

Dynpro 200 pre CSV export



Obr. C.1: Dynpro 200 slúžiace pre špecifikovanie kritérií exportovania do CSV

Príloha D

Obsah priloženého pamäťového média

Priložené CD obsahuje nasledujúce súbory a adresáre:

- `xbagin00.pdf` - bakalárska práca vo formáte PDF
- `latex_src` - adresár obsahuje LaTeX súbory pre vygenerovanie tohto dokumentu
- `source_codes` – adresár obsahuje:
  - `ZGLO_FIN_HU_VAT_DECL` - zdrojový kód hlavného programu
  - `includes` - adresár so zdrojovými kódmi *includov* hlavného programu
  - `dynpros_flow_logic` - adresár s logikou toku dynpier
  - `database_tables.pdf` - podrobnosti databázových tabuliek

Pamäťové médium obsahuje všetky súbory, ktoré je možné zo SAP GUI získať, keďže komplexné exportovanie zdrojového kódu SAP systém neumožňuje.