Pedagogická fakulta
Faculty of Education

Jihočeská univerzita v Českých Budějovicích
University of South Bohemia in České Budějovice

JIHOČESKÁ UNIVERZITA V ČESKÝCH BUDĚJOVICÍCH

PEDAGOGICKÁ FAKULTA

Katedra informatiky

Mgr. Radim Remeš

Problematika výuky objektově orientovaného programování v kurzech programování v terciárním vzdělávání

Disertační práce

Vedoucí disertační práce: doc. Ing. Ladislav Beránek, CSc., MBA

Studijní program: Specializace v pedagogice

Studijní obor: Informační a komunikační technologie ve vzdělávání

České Budějovice 2021

Pedagogická fakulta
Faculty of Education

Jihočeská univerzita v Českých Budějovicích
University of South Bohemia in České Budějovice

UNIVERSITY OF SOUTH BOHEMIA IN ČESKÉ BUDĚJOVICE

FACULTY OF EDUCATION

Department od informatics

Mgr. Radim Remeš

Problems of teaching object-oriented programming in programming courses in tertiary education

Doctoral thesis

Supervisor: doc. Ing. Ladislav Beránek, CSc., MBA

Study programme: Specialization in pedagogy

Field of study: Information and communication technologies in education

České Budějovice 2021

Bibliografická identifikace

Jméno a příjmení autora:	Mgr. Radim Remeš
Název disertační práce:	Problematika výuky objektově orientovaného programování v kurzech programování v terciárním vzdělávání
Název disertační práce anglicky:	Problems of teaching object-oriented programming in programming courses in tertiary education
Studijní program:	Specializace v pedagogice
Studijní obor:	Informační a komunikační technologie ve vzdělávání
Školitel:	doc. Ing. Ladislav Beránek, CSc., MBA
Rok obhajoby:	2021

Klíčová slova v češtině:
výuka programování, objektově orientované programování, objektově orientovaný návrh, prahové koncepty

Klíčová slova v angličtině:
teaching programming, object oriented programming, object oriented design, threshold concepts

Chtěl bych vyjádřit svou nejhlubší vděčnost panu docentovi Ing. Ladislavu Beránkovi, CSc., MBA za odborné vedení mé disertační práce, za cenné rady a ochotné pomoci, které mi byl vždy připraven nabídnout, nikoliv jen při mém studiu a psaní této práce, ale na celé mé akademické cestě.

Také bych chtěl upřímně poděkovat všem mým vyučujícím a kolegům na univerzitě, kteří, ať už vědomě či nevědomě, svou rozličnou činností a pomocí zdárně přispěli k dokončení této práce. Rád bych také poděkoval všem studentům, kteří věnovali svůj čas a byli ochotni se účastnit výzkumu.

Zvláštní velké poděkování patří celé mé rodině, která ve mne po celou dobu věří a trpělivě a neutuchajíce podporuje mé konání.

Prohlašuji, že svoji disertační práci jsem vypracoval samostatně pouze s použitím pramenů a literatury uvedených v seznamu citované literatury.

Prohlašuji, že v souladu s § 47b zákona č. 111/1998 Sb. v platném znění souhlasím se zveřejněním své disertační práce, a to v nezkrácené podobě elektronickou cestou ve veřejně přístupné části databáze STAG provozované Jihočeskou univerzitou v Českých Budějovicích na jejích internetových stránkách, a to se zachováním mého autorského práva k odevzdanému textu této kvalifikační práce. Souhlasím dále s tím, aby toutéž elektronickou cestou byly v souladu s uvedeným ustanovením zákona č. 111/1998 Sb. zveřejněny posudky školitele a oponentů práce i záznam o průběhu a výsledku obhajoby kvalifikační práce. Rovněž souhlasím s porovnáním textu mé kvalifikační práce s databází kvalifikačních prací Theses.cz provozovanou Národním registrem vysokoškolských kvalifikačních prací a systémem na odhalování plagiátů.

V Českých Budějovicích dne 19. července 2021.

Mgr. Radim Remeš

Abstrakt

Výuka programování se zaměřením na objektově orientované programování v různých informatických oborech na vysokých školách patří mezi základní, ale náročné disciplíny. Prezentovaný výzkum se zabýval problémem, jak studenti začátečníci rozumí základním konceptům objektově orientovaného přístupu v programování. Výzkum identifikoval prahové koncepty tohoto přístupu k programování, které jsou pro studenty obtížné, ale jejich překonání je pro dobré zvládnutí objektově orientovaného programování klíčové, a dále se zabýval otázkou, jaké strategie používají studenti ve výuce programování při překonávání těchto obtížných konceptů. Využili jsme fenomenografický rámec; pomocí deduktivní a induktivní obsahové analýzy s otevřeným kódováním jsme identifikovali koncepty objektově orientovaného programování a jejich vlastnosti podle teorie prahových konceptů. Uskutečnili jsme polostrukturované rozhovory, které nám pomohly získat pohled, jak studenti rozumí a používají jednotlivé koncepty objektově orientovaného programování v úlohách zadaných v rámci výzkumu. Použili jsme i metodu hodnocení porozumění konceptům pomocí konceptuálních map a hodnocení úloh a výsledků studentských řešení pomocí SOLO metriky kognitivních procesů. Na základě využití těchto metod jsme identifikovali v rámci výuky objektově orientovaného programování dva prahové koncepty: „Rozhraní" a „Událost (resp. Delegát)". Pro úspěšné osvojení objektového přístupu v programování je nutné zvládnutí obou prahových konceptů, nicméně zvládnutí každého z nich znamená výrazný posun („otevření portálu") ve schopnostech řešit komplexnější programové úlohy pomocí objektově orientovaného paradigmatu. Oba tyto koncepty také patří do abstraktní úrovně SOLO metriky kognitivních procesů, tedy schopnost zpracovávat abstraktní koncepty a myšlenky. Ta je klíčová při navrhování a implementaci složitých informačních systémů.

Abstract

Teaching programming with a focus on object-oriented programming in various computer science disciplines at universities is one of the basic but challenging disciplines. The presented research dealt with the problem of how beginner students understand the basic concepts of object-oriented approach in programming. The research identified the threshold concepts of this approach to programming that are difficult for students, but overcoming them is the key to good mastery of object-oriented programming, and addressed the question of what strategies students use in teaching programming to overcome these difficult concepts. We used a phenomenographical framework; using deductive and inductive content analysis with open coding, we identified the concepts of object-oriented programming and their properties according to the theory of threshold concepts. We conducted semi-structured interviews that helped us gain insight into how students understand and use the various concepts of object-oriented programming in the tasks assigned in the research. We also used the method of evaluating the understanding of concepts using conceptual maps and evaluating the tasks and results of student solutions using SOLO taxonomy of cognitive processes. Based on the use of these methods, we identified two threshold concepts in the teaching of object-oriented programming: ″Interface″ and ″Event (or Delegate)″. To successfully master the object-oriented approach in programming, it is necessary to master both threshold concepts, however, mastering each of them means a significant shift (″opening the portal″) in the ability to solve more complex program tasks using an object-oriented paradigm. Both of these concepts also belong to the abstract level of SOLO metrics of cognitive processes, i.e. the ability to process abstract concepts and ideas. This is the key in the design and implementation of complex information systems.

Obsah

1 Přehled

Na ekonomické fakultě Jihočeské univerzity působím jako učitel již přes 15 let. Většinu z tohoto času vyučuji předmět Programování v bakalářském studiu v prvním a druhém semestru pro studenty informatických oborů. Na začátku jsme učili standardním přístupem, nejdříve jsme probrali témata, jako jsou proměnné, datové typy, příkazy, výrazy, výběr, smyčky, podmínky, funkce a parametry. Poté jsme zavedli objekty a další koncepty objektově orientovaného programování (dále též jen OOP). Nicméně jsme brzy zjistili, že řešení a programy, které studenti vytvořili, se většinou podobaly nikoliv objektovému kódu, ale kódu zapsaném procedurálním stylem v syntaxi jazyka C♯, který jsme používali jako objektově orientovaný jazyk. Proto jsme postupně přistoupili na výuku s důrazem na objekty od samého začátku. Důležitou součástí bylo použití a vývoj explicitní metody pro objektově orientovanou analýzu a návrh. To je naštěstí i součástí vývojového prostředí jazyka C♯ (jedná se o vývojové prostředí Visual Studio, které vyvíjí společnost Microsoft). Ukázalo se také, že dobré příklady na podporu zavedení orientace objektů prostřednictvím objektů je těžké najít a navrhnout. Příklady musí být jednoduché, věrohodné a příkladné, aby ukázaly podstatu paradigmatu OOP, a aby sloužily jako určité vzory pro nováčky. Přesto se nám stále ukazuje, že je nemalý podíl studentů, kteří mají problém předmět Programování se zaměřením na objektově orientované programování zvládnout. V této práci jsme se proto zaměřili na problematiku, jak studenti zvládají základní koncepty objektově orientovaného programování s důrazem na porozumění možným problémům a jejich překonávání.

1.1 Úvod

Výuka programování se zaměřením na objektově orientované programování v různých informatických oborech na vysokých školách patří mezi základní, ale nároč-

né disciplíny. Objektově orientované programování je totiž v současné době dominantním paradigmatem programování používaným při vývoji informatických aplikací pro nejrůznější účely od webových aplikací po složité komplexní podnikové systémy. Někteří autoři doporučují, aby úvodní kurzy programování začínaly výuku programování již od začátku výkladem a použitím objektově orientovaných konceptů a technik (Pecinovský, 2013b; Kořínek, 2017). Koncepce objektově orientovaného programování však klade určité požadavky na abstraktní přemýšlení. Poměrně značné množství studentů nedokončí studium příslušného informatického oboru, protože nesplní požadavky na absolvování předmětu, který se zabývá právě programováním. Mnoho provedených studií také prokazuje, že studenti po jednom až dvou semestrech výuky neumí bez problémů navrhovat a aplikovat základní objektově orientované konstrukce, jako jsou například abstraktní třídy. Je to důsledkem toho, že studenti nepochopí základní koncepty objektově orientovaného programování a přesto, že absolvují nakonec výuku programování, nenajdou k programování vztah a v praxi se programování složitějších aplikací vyhýbají i přesto, že pracují v IT firmách.

Někteří autoři (např. Barnes a kol., 1997) uvádějí, že hlavním příčinou toho, že studenti nejsou příliš úspěšní při učení se programování, je obecný nedostatek schopnosti řešit problémy nebo nedostatek vytrvalosti. Jiní autoři se soustředili na výzkum pochopení klíčových konceptů při učení se programování. Např. Bayman a Mayer (1983) zjistili, že studenti si často vytvoří mylné představy o klíčových programovacích konceptech, což způsobuje obtíže při návrhu a řešení programovacích úloh. Také další autoři (např. Lui a kol., 2004; Ben-Ari, 2001a) zdůrazňují, že pochopení správných představ o klíčových konceptech je pro naučení programování zásadní, zejména v oblasti objektově orientovaného návrhu a programování.

Důležitým aspektem výuky začátečníků objektově orientovaného paradigmatu je seznámit studenty s přístupem k řešení problémů. Bez znalosti toho, jak přistupovat k problému a jak vhodně aplikovat techniky objektově orientované-

ho návrhu v problémových oblastech, je obtížné získat dovednosti v objektově orientovaném způsobu řešení problémů a programování. Studenti musí být podporováni při vytváření nových životaschopných modelů a koncepty musí být prezentovány v pořadí a způsobem, který umožňuje správnou konstrukci vzájemně závislých modelů. Není ale jednoduché, jak to udělat. Předností objektově orientovaného programování je orientace na abstraktní koncepty, které umožňují popsat chování nejrůznějších prvků a vazeb mezi nimi při návrhu komplexního systému a vytvoření informačního systému s vysokými nároky na údržbu, účinnost a opětovné použití.

Ve výuce programování je však z tohoto pohledu situace většinou odlišná. Úvodní příklady, které jsou předkládány studentům pro řešení, nejsou komplexní nebo rozsáhlé, jsou poměrně malé. Rozsah příkladů pro studenty je omezen z důvodu omezeného referenčního rámce nováčka, omezeného počtu dostupných syntaktických prvků a skutečnosti, že počet řádků kódu by měl být, pokud možno, omezen na minimum. Určitým důsledkem je to, že student na začátku nevidí výhodu objektového orientovaného návrhu a programování a má snahu, pokud to jde, se vyhnout návrhu zahrnující složitější koncepty OOP. Zejména, pokud studenti mají již určité počáteční znalosti programování webových stránek nebo skriptů, kde OOP koncepty nepotřebovali. Ve výuce objektově orientovaného programování je obtížné navrhnout příklady ukazující přednosti objektového programování a zároveň se vyhnout příliš složitým příkladům, které by začínající studenti nebyli schopni zvládnout. Pokud je totiž příklad příliš jednoduchý a neukáže přednost OOP, mohou nováčci dojít k závěru, že objektově orientovaný design přináší spíše složitost než řešení problému. Pokud je příklad příliš složitý, studenti nepochopí celkový obraz a tápají při řešení takového příkladu bez jeho hlubšího porozumění. Učení řešení problémů a programování se zdá být obtížnější v objektově orientovaném paradigmatu než v imperativním paradigmatu. Velmi málo však víme o tom, jak studenti uvažují při objektově orientované analýze a návrhu svých aplikací.

Téma disertační práce bylo vybráno s ohledem na výše uvedené problémy, na zjištění, proč je tak obtížné naučit se programování a konkrétně aspektům objektově orientovaného programování. Autor této práce má mnohaleté zkušenosti s výukou programování, a téma tak souvisí s jeho odborným zaměřením a zájmem. V rámci této práce byla zvolena metoda kvalitativního výzkumu s cílem identifikovat různé koncepty v oblasti objektově orientovaného programování, které studenti vnímají jako zvlášť výrazné, například tím, že jsou obtížné nebo obohacující, nebo také problematické a transformativní, z pohledu studentů vysokoškolských kurzů oborů aplikované informatiky. V rámci tohoto kvalitativního výzkumu je využívána především metodika fenomenologického výzkumu (Marton, 1981) a teorie prahových konceptů zavedených autory Meyerem a Landem (2003).

1.2 Vymezení základních pojmů

Pro zpracování disertační práce a zodpovězení výzkumných otázek je důležité vymezení termínů používaných v této disertační práci. Těmto termínům odpovídají i různé programovací styly a různé přístupy k výuce, kde se tato témata mohou nebo nemusí vyskytovat.

1.2.1 Objektově orientované programování

Objektově orientované programování (OOP) je přístup k vývoji systémů, který je zaměřen na objekty a způsob jejich interakce. Objektově orientované programování vychází z předpokladu, že objekty nás obklopují i ve skutečném světě. Například stůl, počítač a šálek vody, to jsou všechno objekty, které mají určité vlastnosti a určité funkce (něco provádějí).

Metoda objektově orientovaného programování (objektově orientovaného paradigmatu) se liší od procedurálního paradigmatu, s nímž mnoho programátorů začíná na své výukové cestě. Kód je psán k definování objektů a tříd a pro návrh

objektově orientovaného programu jsou důležité určité principy, jako jsou zapouzdření, dědičnost a polymorfismus. Programátoři často používají k modelování objektů v programu a jejich interakcí diagramatický jazyk zvaný Unifikovaný Modelovací Jazyk (Unified Modeling Language, UML). Objektově orientovaný návrh je tedy nedílnou součástí objektově orientovaného přístupu k programování.

1.2.2 Objektově orientovaný návrh

Autoři Tsui, Karam a Bernal (2017) definují objektově orientovaný návrh (object oriented design, OOD) jako techniku, která vytváří návrh pomocí tříd, jejich vztahů a vzájemnými interakcemi mezi třídami. OOD se zabývá řízením závislostí. Jedná se o sadu kódovacích technik, které spravují závislosti mezi jednotlivými objekty a tyto jsou pak tolerantní vůči případným změnám. Pokud nepoužijeme techniky OOD, neřízené závislosti vytvoří při změně kódu chaos, protože objekty vzájemně o sobě znají příliš mnoho. Změna jednoho objektu si vyžádá změnu na dalším provázaném objektu, toto zase změní další propojené objekty a tak dále. Zdánlivě malé vylepšení kódu tak může způsobit škodu, která se spirálovitě šíří, což nakonec nezanechává žádnou část kódu nedočtenu (Metz, 2018; Weisfeld, 2019).

1.2.3 Teorie prahových konceptů

Zakladateli teorie prahových konceptů (také prahových pojmů; Threshold Concepts, TC) jsou Jan Meyer a Ray Land (Meyer a Land, 2003). Prahové koncepty definují následovně:

> „...existují určité pojmy nebo určité studijní zážitky, které se podobají průchodu portálem, ze kterého se otevírá nová perspektiva, která umožňuje vidět věci, které dříve nebyly vnímány. To umožňuje nový a dříve nepřístupný způsob uvažování o něčem. Představuje transfor-

> movaný způsob porozumění nebo interpretace nebo prohlížení něčeho, bez něhož by student nemohl postupovat, a vede k přeformulování významového rámce studentů. Přístup prahových hodnot také zdůrazňuje význam disciplinárních kontextů. V důsledku pochopení konceptu prahové hodnoty tak může dojít k transformovanému vnitřnímu pohledu na předmět, oblast subjektu nebo dokonce na svět. Typickými příklady může být „osobnost“ ve filozofii; „testovatelná hypotéza“ v biologii; „gravitace“ ve fyzice; „jalový výkon“ v elektrotechnice nebo „limita“ v matematice.“ (Meyer a Land, 2003, 2010)

Meyer a Land (2003, 2005) tvrdí, že jakmile se student prahové koncepty plně naučí, nemůže se snadno vrátit k vidění světa pouze prostřednictvím předchozího paradigmatu. Prahové koncepty sdílejí několik klíčových kritérií (podle Meyer a Land, 2005), jsou:

- transformativní, protože vytvářejí významný posun v chápání žáka;
- nezvratné, protože je nepravděpodobné, že by změna pohledu, kterou usnadňují, byla zapomenuta nebo odnaučena;
- integrativní v tom, že odhalují dříve skryté souvislosti a pomáhají vysvětlovat jevy novými způsoby;
- někdy ohraničené, což znamená, že pomáhají definovat limity perspektivy nebo disciplíny;
- diskurzivní, což znamená, že často vyvolávají živou debatu, protože odmítají nebo překreslují hranice a odchylují se od základních předpokladů existujícího paradigmatu.

Je třeba poznamenat, že Meyer a Land (2005) uznávají, že ne všechna tato kritéria musí být splněna, aby mohlo být něco považováno za prahový koncept.

Podrobněji se teorií prahových konceptů budeme zabývat v kapitole 1.5.2.

1.2.4 Fenomenografická analýza

Fenomenografie byla vyvinuta z empirického vzdělávacího rámce Ference Martonem a spolupracovníky ve švédském Göteborgu od 70. let (Marton, 1981). Fenomenografie je studium toho, jak lidé zažívají, chápou nebo si představují jevy ve světě kolem nás. Vyšetřování není zaměřeno jen na jev jako takový, ale na rozdíly ve způsobech chápání tohoto jevu lidmi.

Předmětem výzkumu ve fenomenografických studiích je získat informace o způsobu chápání nebo o představách lidí o daném jevu. Fenomenografickou metodou sběru dat jsou zpravidla rozhovory. V těchto rozhovorech jsou respondenti vyzýváni, aby popsali své zkušenosti, a uvedli konkrétní příklady. Cílem zde je, vyhnout se povrchním popisům toho, jak by věci měly být. Hloubkové rozhovory jsou nahrávány a přepisovány. Zkušenosti z velkého počtu fenomenografických studií ukázaly, že údaje od dvaceti informátorů jsou obvykle dostačující k objevení všech různých způsobů chápání daného jevu (Holmström a kol., 2003).

1.3 Definice problému

Umění programovat je komplexní kognitivní dovednost, kterou není lehké zvládnout. Kognitivní vývojové procesy, jež vedou k učení, byly předmětem diskuse mnohých výzkumníků v oblasti informatického vzdělávání již řadu let. Množství publikací v této oblasti poukazuje na to, že učení se programovat je obtížné. Máme však jen malou představu o tom, jak se studenti naučí programovat (Grover a kol., 2015; Lister a kol., 2006; McCracken a kol., 2001; Soloway a Spohrer, 1989.

Bylo provedeno několik empirických výzkumů, kde se sledovalo, s jakými problémy se studenti setkávají při učení různých konceptů jazyka (používání proměnných, využívání cyklů, apod.) (Izu a kol., 2016; Corney a kol., 2012; Du Boulay, 1986).

Další výzkumy se soustředily na sledování problémů začátečníků při učení se konceptům ob-jektově orientovaného programování (Reges, 2006; Lister a kol.,

2006; Fleury, 2000).

Autoři Spohrer a Soloway (1986) ve své studii zjistili, že i přesto, že začátečníci znali syntaxe a sémantiky jednotlivých příkazů, ne vždy věděli, jak tyto známé konstrukty použít pro vytvoření platného kódu. Ve svém výzkumu autoři došli k závěru, že vyučující by měli být schopni zlepšit výkon svých studentů pomocí výuky strategií, jak sestavit jednotlivé části kódu do výsledného funkčního programu a také tím, že jim pomohou naučit se syntaktické a sémantické konstrukty programovacího jazyka.

To by poukazovalo na to, že schopnost začátečníků vyřešit problémy a potíže při psaní kódu vyžaduje ovládnutí i dalších dovedností, kromě dovedností znalosti syntaxe a sémantiky příslušného programovacího jazyka. Většina chyb, které studenti ve svých programech dělají, se týká nedostatku organizačních znalostí a strategií pro řešení problémů. Těmi mohou být např. neschopnost vidět vnitřní souvislosti mezi problémy nebo přenos dobrého nápadu k řešení mezi podobnými problémy v rámci různých kontextů (Muller, 2005).

Problémy týkající se učení se programovat jsou rovněž dobře zdokumentovány v rámci meziinstitucionálních výzkumů. Pracovní skupina zabývající se výukou programování u začátečníků zjistila, že studenti učící se programovat, jsou v prvním roce výuky méně zdatní, než předpokládali jak učitelé, tak i studenti (McCracken, 2001). Tento výzkum inicioval mnohé další meziinstitucionální výzkumy v oblasti výuky studentů, kteří se začínají učit programovat.

Všeobecně udávané vysvětlení pro selhání začátečníků při psaní správného a spolehlivého kódu s vysokou kvalitou je u studentů absence abstrakce zadaného problému. Studenti nejsou schopni nastalý problém analyzovat, rozdělit jej na dílčí jednodušší časti a tyto vyřešené části následně sestavit do požadovaného řešení. Lister a kol. (2004), který navázal na výzkum McCrackena (2001), studoval porozumění kódu. Ve svém výzkumu došel k závěru, že studenti také selhávají právě při porozumění kódu.

Whalley a kol. (2006) rovněž studovali studentovu schopnost porozumění kó-

du a dospěli k závěru, že studenti, kteří neumí číst v kódu a neumí vysvětlit jeho souvislosti, nemají dobré předpoklady pro psaní kódu. Někteří autoři se zaměřili na výzkum souvislostí mezi sledovaním, vysvětlováním a psaním kódu (Kumar, 2013; Murphy a kol., 2012; Lister a kol., 2009; Venables a kol., 2009; Lopez a kol., 2008). Zatímco výsledky většiny výzkumů poukazují na to, že psaní kódu je mnohem obtížnější než jeho čtení, Denny a kol. (2008) a Yamamoto a kol. (2012) dospěli k opačným výsledkům. Winslow (1996) ve svém výzkumu zjistil, že mezi schopností napsat kód a schopností kód číst není žádná významná souvislost. Lister a kol. (2009) také zpochybnil myšlenku posloupnosti hierarchie používaných dovedností, která předpokládá, že příslušná úroveň hierarchie dovedností bude odpovídat příslušným úrovním obtížností problémů, které byly studentům zadávány.

Jiné výzkumy (Ginat a Menashe, 2015; Whalley a kol., 2011; Whalley a kol., 2006) se zaměřily na posouzení úrovně obtížnosti čtení a psaní kódu. Dospěly k závěrům, že jedním z důvodů neúspěchu studentů může být nevhodně zvolený návrh výukového kurzu nebo nevhodně zvolená obtížnost zadaných programových úloh.

Je ověřeno, že v terciárním vzdělávání je výuka programování těžší než výuka ostatních informatických předmětů (Oliver a kol., Dobele a kol., 2004). Předpokládá se, že je to z důvodu provázanosti mezi jednotlivými programovacími koncepty. Student musí nejprve zcela porozumět jednomu programovacímu konceptu, než přistoupí k výuce dalšího konceptu. Teprve poté je schopen využít řešení jednoho programovacího problému při řešení jiného (Robins, 2010).

Výuka programování a modelování pomocí objektově orientovaných metod se během posledních deseti let stala běžným způsobem úvodních kurzů programování v informatice. Dřívější výzkum naznačoval, že učení se objektové orientaci je obtížnější než zvládnutí jiných paradigmat, zatímco novější studie programovacího učení neukázala žádný významný rozdíl mezi procedurálními a objektově orientovanými studenty (Kořínek, 2017). Nicméně zdá se, že pokud studen-

ti začínají s procedurálním programováním nebo programování webů, naučí se syntaxi a sémantiku objektově orientovaného programovacího jazyka, ale bojují pak s tvorbou větších programů. Při učení objektově orientovaného programování má jazyk více konceptů než dříve používané jazyky, což by mohlo znamenat, že osvojení tohoto paradigmatu se stává obtížnějším. Na druhou stranu objektově orientovaný jazyk má koncept třídy, který má usnadnit design a strukturování programů. V kurzech s většími programovacími úkoly by tedy měla být objektová orientace výhodou.

Při výuce programování je důležité si uvědomit, že výuka programování znamená, že se studenti implicitně učí „navrhovat" programy. To znamená to, že ve výuce programování je třeba klást větší důraz na správný návrh včetně zavedení návrhových vzorů zejména v pokročilejších kurzech programování. Pedagogicky je správné proto učit nováčky navrhovat své programy třeba i dříve, než začnou programovat, protože přinejmenším to způsobí, že bude vědomě věnována určitá pozornost přemýšlení o problému a jeho řešení na konceptuální úrovni.

1.4 Výzkumné otázky

Po představení obecné motivace pro tuto práci jsou řešeny výzkumné otázky. Sledují potřebu prozkoumat znalosti a schopnosti začínajících programátorů řešit výše uvedené problémy. Tuto práci lze tedy rozdělit do tří částí. Po představení teoretického rámce pro mé studium je v první části uveden přehled výuky informatiky a reprezentace objektové orientace a objektově orientovaného programování v této oblasti. To vede k následujícím výzkumným otázkám:

RQ1 – Jaké koncepty objektově orientovaného přístupu k programování jsou pro studenty obtížné a zejména transformativní dle teorie prahových konceptů (Meyer a Land, 2005)?

RQ2 – Jaké strategie používají studenti ve výuce programování při překonávání těžkostí u problematických konceptů?

RQ3 – Jak studenti začátečníci při výuce programování chápou základní koncepty objektově orientovaného přístupu v programování?

V rámci disertační práce bude prezentován výzkum chápání základních konceptů a strategií překonávání obtížných konceptů studenty ve výuce programování. Výzkum bude vycházet také ze studií a výzkumů jiných autorů.

1.5 Teoretický rámec

Práce se řídí dvěma základními teoretickými rámci. Prvním z nich je fenomenografie a druhým teorie thresholdů (prahů). Oba tyto rámce si nyní stručně představíme, ale detailněji budou zmíněny v kapitole 4.

1.5.1 Fenomenografie

Fenomenografie je kvalitativní výzkumný přístup, který se stále více používá ke zkoumání způsobu, jakým jednotlivci v určité populaci zažívají určitý jev (Marton, 1981). V rámci výzkumu ve vzdělávání lze fenomenografii použít k identifikaci a kvantifikaci toho, jak studenti vnímají určitý aspekt učení. Přesněji řečeno, fenomenografii lze definovat jako „“ (Marton, 1986)„metodu výzkumu pro mapování kvalitativně odlišných způsobů, jakými lidé zažívají, pojímají, vnímají a chápou různé aspekty a jevy ve světě kolem sebe“.

Fenomenografii použili například Bucks a Oakes (2011) jako výzkumnou metodu ve své studii (Bucks a Oakes, 2011). Použili v rámci teorie například následující výzkumné otázky, které se snažily odhalit různé způsoby, jak studenti prvního ročníku inženýrství chápou různé koncepty programování:

- Jaké jsou kvalitativně odlišné způsoby chápání podmíněných a opakovacích struktur nalezených ve většině programovacích jazyků?
- Jaké jsou způsoby, jak studenti prvního ročníku chápou tyto pojmy?

U malého počtu účastníků výzkumu je často preferovanou metodou polostrukturovaný rozhovor, protože poskytuje bohatý a podrobný popis. U většího počtu účastníků je upřednostňovanou metodou otevřený dotazník, protože je snazší jej spravovat a umožňuje zachytit širší škálu vnímání určitého jevu (Han a Ellis, 2019). V praxi se obě metody často používají ve spojení, protože to umožňuje zahrnout do dat jak šířku, tak hloubku variací.

Marton a Booth (1997, 1998) ukazují, že učení zahrnuje dva aspekty, aspekt „co" a aspekt „jak". Aspekt „co" poukazuje na obsah učení a aspekt „jak" na způsob, jakým učení probíhá. U obou aspektů se zjistilo, že v přístupu studentů k učení existují kvalitativní variace a rozdíly, to uvádí Marton a Booth (1998). Pro studium a vnímání učení určité problematiky, včetně programování, je proto fenomenografie vhodná jako výzkumná metoda, protože se zaměřuje na určování způsobů vnímání specifických jevů studenty, v našem případě konceptů objektově orientovaného programování studenty.

1.5.2 Teorie prahových konceptů

Teorie prahových konceptů (Threshold Concepts, TC) je relativně nový výzkumný rámec (Meyer a Land, 2003), kteří prahové koncepty popsali jako určité body které, jakmile dosáhnou, znamenají určitý skok vpřed v chápání jednotlivce, znamenají dosažení jasnosti komplexnosti konceptu a jeho propojení s jinými myšlenkami. Ve svých dalších pracích tuto definici zjednodušili do několika charakteristik prahových konceptů.

Definice a proces identifikace konceptů prahových hodnot nicméně zůstává subjektivní a sporné (Barradell, 2012; Sanders a McCartney, 2016). Zaprvé chybí shoda ohledně toho, kolik z charakteristik definovaných autory Meyer a Land (2005) je zapotřebí k tomu, aby se koncept stal spíše „prahovou hodnotou" než jednoduše „hlavním" konceptem (Barradell, 2013). Příležitostně se také vyžaduje, aby pojmy prahových hodnot byly hraničními značkami, které označují limity oblasti předmětu (Eckerdal a kol., 2006). Davies (2006) dále připouští, že prahové

koncepty mohou být navíc obtížně identifikovatelné, protože v rámci určité disciplíny mohou být „považovány za samozřejmost“ a jako takové nebudou výslovně uvedeny.

Při pozdějším rozpracování teorie prahových konceptů bylo navrženo, aby identifikace prahových konceptů měla určité přechodné období - od jednoho stavu chápání a poznávání k druhému (Rountree a Rountree, 2009). V takových přechodných obdobích neboli „mezních prostorech“ se studenti s největší pravděpodobností „zaseknou“ (Rountree a Rountree, 2009) a identifikace prahových konceptů je možná. Zmíněné studie také naznačují, že při pochopení podstaty problémů, kterým studenti čelí, je třeba vzít v úvahu jemnější aspekty. Například studenti mohou použít „mimikry“ jako způsob, jak se vypořádat s obtížnými a nepříjemnými koncepty a reprodukovat to, co jim bylo ukázáno, bez konkrétního porozumění. Teorie prahových konceptů se však stala jedním z teoretických rámců používaných při výzkumu v oblasti počítačových věd včetně programování (Sanders a McCartney, 2016). To je dáno poměrně náročnou povahou učení se programování zejména objektově orientovanému programování.

V této práci se zaměříme zejména na problematické a transformativní znalosti, první dvě charakteristiky konceptů prahů, které navrhli Meyer a Land (2005). Sanders a McCartney (2016) argumentují, že kvalita transformace je jedinou kvalitou nezbytnou u základních konceptů. Sledovali jsme rovněž pojmy konceptuálního porozumění. K identifikaci případů spojených s obtížnými znalostmi, kdy došlo pouze k částečnému porozumění. Částečné porozumění lze prokázat abstraktním porozuměním pojmům bez možnosti jejich konkrétního použití a naopak. Dále jsme využili přístup Eckerdal a kol. (Eckerdal a kol., 2006) zjišťováním více aspektů transformativní charakteristiky, včetně „pocitu jako programátora“ a hledali jsme další vodítka, například důkazy o emoční reakci.

1.6 Metodologie

Pro provedení výzkumu v rámci této práce jsme zvolili kvalitativní přístup. Byl použit přístup kvalitativního rozhovoru a hlasitého komentování postupu práce nad zadanými příklady. Tento přístup byl formován fenomenologickým rámcem a teoretickým rámcem založeným na teorii prahových konceptů. Na začátku jsme se zaměřili na první dvě charakteristiky teorie prahových konceptů (Meyer a Land, 2005), a to problematické a transformativní. Výzkumné otázky a pozorování byly proto směrovány s ohledem na tyto charakteristiky. Tedy v rámci dotazníků to byly otázky týkající se aspektů programování, s nimiž účastníci měli potíže, a které jim potenciálně poskytly významný posun v chápání učiva. Postupně jsme se v našem výzkumu začali zabývat i další problematickou charakteristikou, kterou popsala Eckerdal a kol. (2006). Jedná se o to, že je třeba vzít v úvahu, že studenti mohou používat „mimikry“ jako způsob potírání obtížných konceptů a reprodukovat to, co jim bylo ukázáno, bez konkrétního porozumění. Ačkoli tento přístup studentů někteří autoři považují za užitečný krok k úplnějšímu pochopení konceptu (Meyer a Land, 2005; Hughes a Peiris, 2006; Rountree a Rountree, 2009). Využitím prahových konceptů ve výuce programování se zabývají Yeomans a kol. (2019). Docházejí k závěru, že transformativní aspekt definice prahového konceptu v kontextu programování je to dobrým základem pro identifikaci důležitých konceptů, které mohou studentům dělat problémy, a ty které jsou pro ně transformativní a znamenají velký posun v chápání a zejména použití důležitých konceptů v programování.

Pozorování a nestrukturované rozhovory s vysokoškolskými studenty prvního ročníku probíhaly po dobu dvou semestrů v průběhu školního roku 2019/2020, kdy studentům byly kladeny otázky týkající se jejich zkušeností s programováním a které aspekty kurzu se jim líbily a nelíbily. I když tato data nebyla při analýze rozsáhle využívána, vytvořila zásadní krok při informování otázek kladených pro další fázi sběru dat a také, v praxi, ke zlepšení míry účasti studentů, když by-

li pozváni k účasti v cílových skupinách navržených tak, aby cíleněji zkoumaly potenciální koncepty prahových hodnot.

Tento cílený výzkum byl proveden v průběhu dvou semestů v přůběhu školního roku 2020/2021. Ve snaze dosáhnout určité shody ohledně koncepcí prahových hodnot pro začínající programátory studentů byla shromážděna data z jedné vzorové populace. Jednalo se o studenty prvního ročníku na Ekonomické fakultě Jihočeské univerzity oboru Ekonomická informatika. Obsah předmětu Programování, který je dvousemestrový, u těchto studentů odpovídá doporučením organizace ACM a IEEE (JTFCC, 2013) a odpovídá obsahu výuky předmětu programování na vysokých školách s informatickým zaměřením. Účastníci byli požádáni zejména o popis aspektů programování, které považovali za náročné, ale byli také požádáni, aby diskutovali o všech koncepcích, které změnily jejich vnímání samotného programování. Při těchto rozhovorech byly již používány polostrukturované rozhovory a byly využity i konceptuální mapy pro získání přehledu, jak studenti chápou základní koncepty objektového přístupu v programování a vztahy mezi nimi.

V rámci této práce se tedy snažíme o porozumění, jak studenti chápou základní koncepty objektově orientovaného programování, které koncepty jim dělají potíže, kde si pomáhají „mimikry“, a kdy nastane případný posun v chápání a použití těchto programovacích konceptů. Popisu výsledků zjištění na základě pozorování, analýzy návrhu a kódu a polostrukturovaných rozhovorů, sběru a analýze dat se budeme detailněji věnovat v kapitole 5.

1.7 Přehled zjištění

Prezentovaný výzkum se zabýval problémem, jak studenti začátečníci rozumí základním konceptům objektově orientovaného přístupu v programování. Výzkum identifikoval prahové koncepty tohoto přístupu k programování, které jsou pro studenty obtížné, ale jejich překonání je pro dobré zvládnutí OOP klíčové, a dá-

le se zabýval otázkou, jaké strategie používají studenti ve výuce programování při překonávání těchto obtížných konceptů. Využili jsme fenomenologický rámec (Marton, 1981) v rámci deduktivní obsahové analýzy a pomocí induktivní obsahové analýzy s otevřeným kódováním jsme identifikovali koncepty OOP a jejich vlastnosti podle teorie prahových konceptů (Meyer a Land, 2005). Uskutečnili jsme polostrukturované rozhovory, které nám pomohly získat pohled, jak studenti rozumí a používají jednotlivé koncepty OOP v úlohách zadaných v rámci výzkumu. Použili jsme i metodu hodnocení porozumění konceptům pomocí konceptuálních map a hodnocení úloh a výsledků studentských řešení pomocí SOLO metriky kognitivních procesů. Na základě využití těchto metod jsme identifikovali v rámci výuky objektově orientovanému programování dva prahové koncepty: „Rozhraní" a „Událost". Pro úspěšné osvojení objektového přístupu v programování je nutné zvládnutí obou prahových konceptů, nicméně zvládnutí každého z nich znamená výrazný posun („otevření portálu", jak prahové koncepty popisují Meyer a Land (2005)) ve schopnostech řešit komplexnější programové úlohy pomocí objektově orientovaného paradigmatu. Oba tyto koncepty také patří do abstraktní úrovně SOLO metriky kognitivních procesů, tedy schopnost zpracovávat abstraktní koncepty a myšlenky. Ta je klíčová při navrhování a implementaci složitých informačních systémů. Ukazuje se také, že většina konceptů objektově orientovaného přístupu k programování se jeví pro studenty jako problematická. To odpovídá i tomu, co píše velké množství autorů, kteří se zabývají výukou programování, že naučit se dobře programovat není lehké. Detailnímu pohledu na zjištění v rámci jednotlivých výzkumných otázek a přehledu a diskuzi k těmto zjištěním se věnujeme v kapitolách 5 a 6.

2 Teoretická východiska – přehled literatury

2.1 Objektově orientované programování

V literatuře můžeme nalézt mnoho definic toho, co nazýváme programováním. Např. dle Wassberg (2020):

„Programování je umění a věda, jak nappsat pokyny, které může počítač dodržovat při plnění úkolu. Tímto úkolem může být například hraní hry, provádění výpočtů nebo procházení webu.. “ (Wassberg, 2020)

Jak tento pojem můžeme popsat jednoduše? Přestože se mnoha neprogramátorům může zdát, že jde o jedinou aktivitu, ve skutečnosti to může být velmi různorodá řada aktivit zaměřených na řešení nejrůznějších problémů. Například existuje velký rozdíl mezi návrhem a implementací komplexního podnikového systému a malým skriptem pro výpočet a vykreslení nějakého grafu. Pokud bychom měli vyjádřit jednoduše obsah „programování“, pak by naše definice mohla být:

„Programování je činnost sestávajících z porozumění zadaného problému, formulování řešení a jeho přepis tak, aby bylo možné využít počítač k vyřešení zadaného problému. “

Naše definice znamená, že programátor (nebo tým programátorů) musí nejprve porozumět zadanému problému. Například při vytváření systému pro účetnictví musí programátor rozumět účetnictví a tomu, jak účetní fungují. Poté může programátor použít nástroje a znalosti pro formulaci řešení a návrh (např. navrhnout řešení pomocí různých schémat a strukturovaných grafů. V posledním kroku musí programátor návrh převést do formy, která umožní, aby navržený systém fungoval na počítači a prováděl předepsané úkony. Nejběžnější formou je zápis pomocí některého programovacího jazyka jako je Java, C, C♯, Python, Ja-

vaScript nebo PHP a další.

Návrh řešení a jeho přepis do programovacího jazyka není jednoduchý úkol a je stále předmětem řady diskuzí. Byly navrženy a diskutovány nejen paradigmata (Bieliková a Návrat, 2009; Konečný a Vychodil, 2008; Hubálovský a Kořínek, 2015a; Hubálovský a Kořínek, 2015b; Kořínek, 2017) různých jazyků, jako je objektově orientované a logické programování, ale také to, jak by měla vypadat notace různých paradigmat. Tyto paradigmata ovlivňují způsob návrhu a tvorby programů. Programovací paradigma lze tedy definovat jako základní programovací styl, která představuje způsob vývoje konkrétních informačních systémů. Různá paradigmata se liší v pojmech a abstrakcích, které tvoří jednotlivé prvky programu (objekty, funkce, proměnné, omezení, aj.), a krocích, ze kterých se výpočet skládá (přiřazení, vyhodnocení, datové toky, atd.). Programovací jazyk může podporovat i více paradigmat (Konečný a Vychodil (2008). Například programy psané v jazyce Python mohou být čistě procedurální, čistě objektové nebo kombinace obou paradigmat. Vývojáři pak mají možnost se rozhodnout, jak a které prvky paradigmat použijí.

Ve výuce programování je proto důležitý i výběr paradigmatu a programovacího jazyku, se kterým se studenti seznámí. Tento výběr má vliv na to, jakým způsobem budou studenti vytvářet programy a zda budou schopni navrhovat a programovat komplexní systémy. Různé práce se zabývají problémem, jaké paradigma zvolit pro výuku v úvodních kurzech programování na vysokých školách pro informatické obory, např. Kořínek (2017), Hubálovský a Kořínek (2015a, 2015b) a další. Ve výuce programování, kde jsme analyzovali problémy studentů s různými programovacími koncepty, je zaveden přístup objektového programování. V této práci se proto nebudeme zabývat popisem jiných paradigmat, byť jsou poměrně také zajímavé a v různých oblastech mohou být pro studenty velmi užitečné.

Základní myšlenky, na nichž je objektově orientované programování postaveno, se objevily na počátku 60. let. Na počátku 70. let vznikl konceptuálně zcela

nový programovací jazyk nazvaný Smalltalk. Tento jazyk stavěl na myšlenkách objektově orientovaného přístupu k programování (OOP). Díky jazyku Smalltalk se OOP rozšířilo na univerzity a dále do praxe včetně vývoje jazyků využívajících toto paradigma. Pojem a obsah termínu objektově orientované programován může být různými autory chápáno rozdílně. Většina autorů definuje OOP pomocí termínu objekt a termínu vlastnit, nebo že objekty mohou vzájemně komunikovat a předávat si nějaké vlastnosti či metody, např. Drbal (1994) definuje OOP takto:

> „objektově programovat znamená používat objekty jako stavební kameny, ze kterých se staví program. Nějaké dědění zde nemá žádný smysl. Daleko důležitější vztahy zde jsou: obsahuje, vlastní, používá a vztahy inverzní je obsažen, je vlastněn, je používán. “ (Drbal, 1994)

Jinou definici OOP uvádí např. Asagba a Ogheneovo (2008, s. 42), píší, že OOP

„...používá pro tvorbu aplikací a počítačových programů tzv. „objekty“ a interakce mezi nimi. “ (Asagba a Ogheneovo, 2008, s. 42)

Trochu jinak definuje OOP Pecinovský (2009), který píše:

„objektově orientované programování vychází z myšlenky, že všechny programy, které vytváříme, jsou simulací buď skutečného, nebo nějakého námi vymyšleného virtuálního světa. “ (Pecinovský, 2009)

Jiní autoři definují OOP v podstatě jako nadstavbu imperativního programování. Např. Klimeš a kol. (2008) píší:

> „každé, tj. i objektově orientované, programování v sobě na úrovni programování základních operací nese vlastnosti strukturovaného programování a ani to, že při tvorbě aplikací využíváme komponenty, neznamená, že programujeme objektově. “ (Klimeš a kol., 2008)

Tyto definice však podle autora této disertační práce nevystihují podstatu OOP. Také např. Kořínek (2017) píše:

„...formulace OOP, na základě imperativního paradigmatu programování, není příliš vhodná. Jedná se o odlišný způsob návrhu programu. “ (Kořínek, 2017)

> „Definice OOP jako vzájemná komunikace objektů s voláním metod se jeví nejpřijatelněji, protože tato formulace OOP je nezpochybnitelná. Komunikace mezi objekty s voláním metod může být v některých programovacích jazycích realizována různým způsobem, ale princip komunikace zůstává stejný. “ (Kořínek, 2017)

Nejlépe přístup k OOP a jeho podstatu vystihuje, podle autora této disertační práce, Pecinovský (2007):

> „Typickým přístupem začátečníků postavených před vyřešení nějakého problému je to, že začnou okamžitě přemýšlet na úrovni jednotlivých instrukcí. Tento problém jsme řešili i při výuce podle starších metodik. Vzhledem k tomu, že značnou část strukturovaných programů lze považovat jako v programovacím jazyce zapsané posloupnosti příkladů, vedoucí k vyřešení problému.
>
> Objektové programy mají ale trochu jinou filozofii. Mohli bychom je charakterizovat jako množinu objektů a zpráv, které si mezi sebou posílají, zapsanou v nějakém programovacím jazyce. Způsob myšlení nutný při návrhu objektového programu se od klasického liší stejně zásadně, jako se liší jejich definice.
>
> Seznámíme-li studenty s objektovými konstrukcemi až poté, co už zažili jiný způsob uvažování a návrhu programu, velmi těžko je budeme přeučovat na takový způsob, který moderní programování vyžaduje. “ (Pecinovský, 2007)

Na výběru prvního paradigmatu programování v úvodních programovacích kurzech (v zásadě imperativní /strukturované/ nebo OO) je vidět stav paradigmat programování obecně a zejména vývoj objektové orientace a současný stav. V jednom z prvních článků o různých paradigmatech programování v kontextu vzdělávání píše o roli objektově orientovaného programování např. Luker (1989):

„...měli bychom hlasovat pro tento styl programování (objektově orientované programování), protože je to přirozený. Jazyk by měl vynucovat použití objektů, stejně jako SMALLTALK. “ (Luker, 1989, s. 256)

Postupně se tedy od konce devadesátých let do výuky v úvodních kurzech programování začíná zavádět vedle imperativního paradigmatu (procedurálního programování) i objektově orientované paradigma (objektově orientované programování). Tento vývoj však probíhá postupně.

Např. Mitchell (2000) shrnuje posun paradigmatu v průmyslu a snaží se vysvětlit, proč je obtížné integrovat „nové“ (objektově orientované) paradigma do výuky programování v informatických oborech. Navrhuje následující čtyři postupy pro zavedení objektově orientovaného paradigmatu. Navrhuje, aby studenti neupřednostňovali žádnou konkrétní metodiku řešení problémů. Říká, že existují rozdíly mezi přístupy k výuce spojenými s procedurálním nebo objektově orientovaným paradigmatem a že je třeba zohlednit dřívější zkušenosti studentů s programováním. Poslední krok bylo zohlednit dřívější zkušenosti studentů s programováním, a ten měl nakonec nejsilnější vliv. Znamenalo to, že objektově orientované programování lze zařadit do výuky v kterémkoliv okamžiku do výuky dle preferencí a zkušeností daného vyučujícího.

Jiný autor popisuje vliv změny paradigmatu programování ve výuce informatiky takto:

> „Objektově orientované programování bylo po dlouhou dobu považováno za pokročilý předmět, který byl vyučován pozdě v učebních osnovách. To se pomalu mění: stále více univerzit začalo již ve svém prvním kurzu programování učit objektovou orientaci. Hlavním důvodem je často citovaný problém změny paradigmatu. “ (Kölling, 1999)

Nejdůležitější změnou, která přišla se změnou paradigmatu a která má velmi silný vliv na vzdělávání, je změna pohledu na procesy. Zatímco v procedurálním paradigmatu je jeden proces zodpovědný za funkčnost celého programu,

v objektově orientovaném paradigmatu působí objekty pokud možno samy o sobě. To znamená, že v procedurálním pohledu jsou záznamy měněny procedurami zvenčí, zatímco v objektově orientovaném pohledu objekty volají metody k zahájení změny hodnot objektu. Zejména v objektově orientovaném paradigmatu jsou procesy vázány přímo na objekty (Vujoševic-Janicic a Tošic, 2008; Hubwieser 2007). Další významná změna pohledu se týká implementace variant. Objektově orientované paradigma umožňuje polymorfismus. Toto umožňuje podmíněný výběr v závislosti na „typu akčního" prvku (Garrigue 1998, 2000).

Při zavádění objektově orientovaných poznámek je třeba vzít v úvahu další obtíž. Objekty jsou implementovány pomocí odkazů nebo ukazatelů téměř ve všech programovacích jazycích. Přesněji řečeno, objekt je pouze alokovanou částí v paměti počítače s ukazatelem nebo odkazem směřujícím na něj. Pro studenty je těžké se naučit, pokud dva odkazy směřují na stejný objekt. Při přechodu z procedurálního paradigmatu na objektově orientované paradigma je toto porozumění zásadní, protože změna hodnot je odlišná. Zatímco v procedurálním paradigmatu přístup k proměnné ovlivňuje pouze proměnnou, která je adresována, v objektově orientovaném paradigmatu mohou být ovlivněny i odkazy související s jinými atributy (Hubwieser 2007).

2.2 Základní vlastnosti objektově orientovaného programování

Základní vlastnosti objektově orientovaného programování, kterým musí studenti porozumět a naučit se využívat (včetně i historie OOP, která studentům poskytuje širší pohled), jsou popsány v následujících kapitolách.

2.2.1 Historie objektově orientovaného programování

Koncepty OOP se začaly objevovat v polovině 60. let pomocí programovacího jazyka s názvem Simula a dále se vyvinuly v 70. letech s příchodem Smalltalku. Ačkoli vývojáři softwaru tyto rané pokroky v jazycích OOP v naprosté většině nepřijali, objektově orientované metodiky se nadále vyvíjely. V polovině 80. let došlo k obnovení zájmu o objektově orientované metodiky. Konkrétně OOP jazyky jako C++ a Eiffel se staly oblíbenými u běžných počítačových programátorů. OOP nadále rostl v popularitě v 90. letech, zejména s příchodem Javy, které vyvolalo obrovský zájem. V roce 2002 společnost Microsoft představila spolu s vydáním .NET Framework nový OOP jazyk C♯ a vylepšila svůj široce oblíbený existující jazyk Visual Basic, který je od této verze skutečně objektově orientovaný. Dnes jazyky OOP nadále vzkvétají a jsou základem moderního programování.

2.2.2 Důvody použití objektově orientovaného programování

Proč se OOP vyvinul do tak široce používaného paradigmatu pro řešení obchodních problémů dnes? Během sedmdesátých a osmdesátých let byly k vývoji obchodně orientovaných softwarových systémů široce používány programovací jazyky orientované na procedury, jako jsou C, Pascal a Fortran. Procedurální jazyky organizují program lineárně - běží shora dolů. Jinými slovy, program je řadou kroků, které probíhají jeden po druhém. Tento typ programování fungoval dobře pro malé programy, které se skládaly z několika stovek řádků kódu, ale jak se programy zvětšovaly, bylo obtížné je spravovat a ladit.

Ve snaze řídit stále rostoucí velikost programů bylo zavedeno strukturované programování, které rozdělilo kód na spravovatelné segmenty zvané funkce nebo procedury. Jednalo se o vylepšení, ale jak programy prováděly složitější obchodní funkce a spolupracovaly s jinými systémy, začaly se objevovat následující nedostatky strukturálního programování:

- Programy se těžce udržovaly.

- Stávající funkce bylo obtížné změnit, aniž by to nepříznivě ovlivnilo všechny funkce systému.
- Nové programy byly vytvořeny v podstatě od nuly. Investice předchozího úsilí tedy byla málo návratná.
- Programování nebylo příznivé pro vývoj týmu.
- Programátoři museli znát všechny aspekty toho, jak program funguje, a nedokázali nasadit své úsilí na jeden aspekt systému.
- Bylo těžké převést obchodní modely do programovacích modelů.
- Strukturální programování fungovalo dobře izolovaně, ale nebylo dobře integrováno s jinými systémy.

Kromě těchto nedostatků způsobil některý vývoj výpočetních systémů další tlak na přístup strukturálního programu, například:

- Neprogramátoři požadovali a získali přímý přístup k programům prostřednictvím zabudování grafických uživatelských rozhraní a jejich stolních počítačů.
- Uživatelé požadovali intuitivnější a méně strukturovaný přístup k interakci s programy.
- Počítačové systémy se vyvinuly v distribuovaný model, kde obchodní logika, uživatelské rozhraní a backendová databáze byly volně spojeny a byly přístupné přes internet a intranety.

Ve výsledku se mnoho vývojářů podnikového softwaru obrátilo na objektově orientované metody a programovací jazyky. Mezi výhody patří:

- intuitivnější přechod z modelů obchodní analýzy na modely implementace softwaru
- schopnost efektivněji a rychleji udržovat a implementovat změny v programech

- schopnost efektivněji vytvářet softwarové systémy pomocí týmového procesu, což umožňuje specialistům pracovat na částech systému
- možnost opětovného použití komponent kódu v jiných programech a nákup komponent napsaných vývojáři třetích stran za účelem zvýšení funkčnosti stávajících programů s malým úsilím
- lepší integrace s volně vázanými distribuovanými výpočetními systémy
- vylepšená integrace s moderními operačními systémy
- schopnost vytvářet pro uživatele intuitivnější grafické uživatelské rozhraní

2.2.3 Charakteristika objektově orientovaného programování

V této části prozkoumáme některé základní pojmy a pojmy společné pro všechny jazyky OOP (viz též obr. E.1 a D). Cílem je seznámit se s koncepty a spojit je s již známými každodenními zkušenostmi, aby měly později větší smysl, když se podíváme na design a implementaci OOP. Budeme vycházet z Amstrongova seznamu konceptů objektově orientovaného přístupu (Amstrong, 2006). Podrobněji popíšeme základní pojmy a koncepty OOP v kapitole 5.

Objekty

Jak již bylo uvedeno, žijeme v objektově orientovaném světě. Jsme objekt. Interagujeme s jinými objekty. Ve skutečnosti jsme objekt s údaji, jako je naše výška a barva vlasů. Máme také metody, které provádíme nebo které se provádějí na nás, jako je jídlo a chůze.

Co jsou tedy objekty? Z hlediska OOP je objekt strukturou pro začlenění dat a postupy pro práci s těmito daty. Například pokud by nás zajímalo sledování dat souvisejících s inventářem produktů, vytvořili bychom objekt produktu, který je zodpovědný za údržbu a používání dat vztahujících se k produktům. Pokud bychom chtěli mít ve své aplikaci možnosti tisku, pracovali bychom s objektem

tiskárny, který je zodpovědný za data a metody používané k interakci s našimi tiskárnami.

Abstrakce

Při interakci s objekty na světě nám často jde pouze o podmnožinu jejich vlastností. Bez této schopnosti abstrahovat nebo odfiltrovat cizí vlastnosti objektů bychom těžko zpracovali nepřeberné množství informací, které nás bombardují, a těžko bychom se mohli soustředit na daný úkol.

V důsledku abstrakce, když dva různí lidé interagují se stejným objektem, často se zabývají jinou podmnožinou atributů. Když například řídíme auto, potřebujeme znát rychlost vozu a směr, kterým jede. Protože auto používá automatickou převodovku, nepotřebujeme znát otáčky motoru za minutu (RPM), proto tyto informace odfiltrujeme. Na druhou stranu by tato informace byla zásadní pro řidiče závodního vozu, který by si ji neodfiltroval. Při konstrukci objektů v aplikacích OOP je důležité začlenit tento koncept abstrakce. Objekty obsahují pouze informace, které jsou relevantní v kontextu aplikace. Pokud bychom vytvářeli přepravní aplikaci, postavili bychom produktový objekt s atributy, jako je velikost a hmotnost. Barva položky by byla cizí informací a byla by ignorována. Na druhou stranu, při konstrukci aplikace pro zadávání objednávek může být důležitá barva a bude zahrnuta jako atribut objektu produktu.

Zapouzdření

Další důležitou vlastností OOP je zapouzdření. Zapouzdření je proces, při kterém není poskytnut přímý přístup k datům; místo toho je skrytý. Chceme-li získat přístup k datům, musíme komunikovat s objektem odpovědným za data. V předchozím příkladu inventáře, pokud bychom chtěli zobrazit nebo aktualizovat informace o produktech, budeme muset projít objektem produktu. Chceme-li číst data, odešleme objektu produktu zprávu. Objekt produktu by poté přečetl hodnotu a poslal zprávu, která nám řekne, o jakou hodnotu jde. Objekt produktu de-

finuje, které operace lze s daty produktu provést. Pokud odešleme zprávu o úpravě dat a objekt produktu určí, že jde o platný požadavek, provede operaci za nás a odešle zprávu zpět s výsledkem.

Zapouzdření zažíváme ve svém každodenním životě neustále. Přemýšlejme o oddělení lidských zdrojů. Zapouzdřují (skrývají) informace o zaměstnancích. Určují, jak lze tato data použít a manipulovat. Prostřednictvím nich musí být směrován jakýkoli požadavek na údaje o zaměstnancích nebo požadavek na aktualizaci údajů. Dalším příkladem je zabezpečení sítě. Jakýkoli požadavek na informace o zabezpečení nebo na změnu zásad zabezpečení musí být podán prostřednictvím správce zabezpečení sítě. Bezpečnostní data jsou zapouzdřena od uživatelů sítě.

Zapouzdřením dat zvýšíme bezpečnost a spolehlivost dat našeho systému. Víme, jak se k datům přistupuje a jaké operace se s daty provádějí. Díky tomu je údržba programu mnohem jednodušší a také výrazně zjednodušuje proces ladění. Můžeme také upravit metody používané k práci s daty, a pokud nezměníme, jak je metoda požadována a typ odpovědi odeslané zpět, nemusíme měnit jiné objekty pomocí metody. Přemýšlejme o tom, když pošlete dopis e-mailem. Požádáme poštu o doručení dopisu. To, jak toho pošta dosáhne, nám není ukázáno. Pokud změní trasu, kterou používá k odeslání dopisu, nemá to vliv na to, jak zahájíme odeslání dopisu. Nemusíme znát interní postupy pošty používané k doručení dopisu.

Polymorfismus

Polymorfismus je schopnost dvou různých objektů reagovat na stejnou žádost požadavku svým vlastním jedinečným způsobem. Například bychom mohli trénovat svého psa, aby reagoval na příkaz štěkáním, a svého papouška, aby reagoval na příkaz pískáním. Na druhou stranu jsme je mohli trénovat, aby oba reagovali na povel. Díky polymorfismu víme, že pes bude reagovat štěkáním a papoušek bude reagovat pískáním. Jak to souvisí s OOP? Můžeme vytvářet objekty,

které reagují na stejnou zprávu v jejich vlastních jedinečných implementacích. Můžeme například odeslat tiskovou zprávu objektu tiskárny, který vytiskne text na tiskárně, a stejnou zprávu můžeme odeslat objektu obrazovky, který vytiskne text do okna na obrazovce našeho počítače.

Dalším dobrým příkladem polymorfismu je používání slov v běžném jazyce. Slova mají mnoho různých významů, ale z kontextu věty lze odvodit, o jaký význam jde. Víme, že když někdo říká: „Dejte mi pokoj!“, tak nás nežádá, abychom mu poskytli místnost.

V OOP implementujeme tento typ polymorfismu prostřednictvím procesu zvaného přetížení. Můžeme implementovat různé metody objektu, které mají stejný název. Objekt pak může určit, kterou metodu implementovat, v závislosti na kontextu (jinými slovy, počtu a typu předaných argumentů) zprávy. Můžeme například vytvořit dvě metody objektu inventáře a vyhledat cenu produktu. Obě tyto metody by byly pojmenovány getPrice. Jiný objekt by mohl volat tuto metodu a předat buď název produktu, nebo ID produktu. Objekt inventáře mohl zjistit, kterou metodu getPrice spustit podle toho, zda byla s požadavkem předána hodnota řetězce nebo celočíselná hodnota.

Dědičnost

Většina skutečných objektů může být rozdělena do hierarchií. Můžeme například klasifikovat všechny psy dohromady, že mají určité společné vlastnosti, jako je mít čtyři nohy a srst. Jejich plemena je dále zařazují do podskupin se společnými atributy, jako je velikost a chování. Také klasifikujeme objekty podle jejich funkce. Například existují užitková vozidla a rekreační vozidla. K dispozici jsou nákladní automobily a osobní automobily. Klasifikujeme auta podle jejich značky a modelu. Chceme-li dát světu smysl, musíte použít hierarchie a klasifikace objektů. Dědičnost v OOP používáme ke klasifikaci objektů v našich programech podle společných charakteristik a funkcí. Díky tomu je práce s objekty jednodušší a intuitivnější. Také usnadňuje programování, protože umožňuje kombinovat

obecné charakteristiky do nadřazeného objektu a zdědit tyto vlastnosti v podřízených objektech. Můžeme například definovat objekt zaměstnance, který definuje všechny obecné charakteristiky zaměstnanců v naší společnosti.

Poté můžeme definovat objekt správce, který zdědí vlastnosti objektu zaměstnance, ale také přidá vlastnosti jedinečné pro správce v naší společnosti. Z důvodu dědičnosti bude objekt správce automaticky odrážet jakékoli změny charakteristik objektu zaměstnance.

Kompozice

Skládání objektů z jiných objektů, které spolupracují, se nazývá kompozice. Například náš objekt sekačky na trávu je složený z objektů kola, objektu motoru, nože atd. Objekt motoru je ve skutečnosti složen z mnoha dalších objektů. Ve světě kolem nás existuje mnoho příkladů kompozice. Schopnost používat kompozici v OOP je výkonná funkcionalita, která nám umožní přesně modelovat a implementovat obchodní procesy v našich programech. Jak uvádí Reynolds-Haertle (2002, s. 23) u vlastností, tak „jsou to charakteristiky neboli kvantitativní znaky, jimiž se objekt vyznačuje…“.

Softwarová architektura

Softwarová architektura je vyšší úroveň pohledu na systém využívající techniky OOD. Kazman, Clements a Bass (2012) ji definují jako soubor struktur potřebných k uvažování o systému, který zahrnuje softwarové prvky, vztahy mezi nimi a vlastnosti obou. Autoři též zmiňují několik důležitých bodů, které se týkají softwarové architektury. Každý systém má architekturu. Nezávisí na tom, zda je explicitně uvedena a zdokumentována či nikoliv, systém architekturu má. Může existovat více než jedna struktura. U velkých systémů a dokonce i u mnoha malých systémů existuje více než jen jeden způsob, jak je systém strukturován. Je potřeba si všechny tyto struktury uvědomovat a dokumentovat je několika pohledy. Architektura se zabývá vlastnostmi zevně pro každý modul. Na úrovni ar-

chitektury je třeba přemýšlet o důležitých modulech a o tom, jak vzájemně komunikují s ostatními moduly. Důraz je nutné klast spíše na rozhraní mezi moduly než na detaily týkající se vnitřních částí každého modulu.

Návrhové vzory

Návrhový vzor (anglicky design pattern) v softwarovém inženýrství představuje obecné řešení problému, které se využívá při návrhu počítačových programů (Reynolds-Haertle, 2002). Návrhový vzor není knihovnou nebo částí zdrojového kódu, která by se dala přímo vložit do našeho programu, jedná se o popis řešení problému nebo šablonu, která může být použita v různých situacích. Objektově orientované návrhové vzory typicky ukazují vztahy a interakce mezi třídami a objekty, aniž by určovaly implementaci konkrétní třídy. Algoritmy nejsou považovány za návrhové vzory, protože řeší konkrétní problémy a nikoliv problémy návrhu.

Metrika kódu

Metrika je opakované měření, které má informační, diagnostickou, motivační nebo prediktivní schopnost nějakého druhu. Pomáhá pochopit, zda hrozí riziko, že budou chybět očekávané výsledky, nebo zda změny v procesu nebo postupech povedou ke zlepšení výkonu (Nicolette, 2015). Metrikou kódu rozumíme metriku zdrojového kódu. Sole (2012) definuje metriku kódu jako nástroj pro analýzu kódu, pomocí kterého lze získat informace o snadnosti udržitelnosti kódu. Příkladem takových informací mohou být počet řádků kódu, počet použitých tříd či úroveň hloubky dědičnosti (Baley a Belcham, 2010).

Konstruktor

Konstruktor je speciální metoda, která slouží k vytvoření instance. Konstruktor inicializuje objekt při jeho vytvoření. Má stejný název jako jeho třída a je syntakticky podobný metodě Schildt (2012). Konstruktory však nemají žádný explicitní

návratový typ“.

Metoda

Popis, co je to metoda v rámci OOP, podává podle našeho názoru nejvýstižněji Pecinovský. Ve své publikaci (Pecinovský, 2009) uvádí, že „metoda je část programu, kterou instance spustí v reakci na obdržení zprávy“.

jednoduché datové typy

Jednoduché datové typy jsou jasně definovány a zakomponovány přímo v příslušném programovacím jazyce, kde je uživatel využívá. Pecinovský (2009) je nazývá primitivní datové typy a uvádí, že „primitivní datové typy, mezi něž patří např. čísla, jsou zabudovány hluboko v jazyku a jejich chování je pevně dané“. Mezi jednoduché datové typy patří např. celá čísla, reálná čísla, řetězce a další. V různých programovacích jazycích mohou být u jednoho datového typu, např. celého čísla, další možnosti jemnějšího rozdělení, např. podle rozsahu, znaménka apod.

Třída

Drbal (1996) uvádí následující definici třídy: „třída je identifikovatelný typ entity nebo konceptu v daném prostředí. Třída je skupina datových proměnných a chování (operace a funkce), což je sdíleno instancemi toho typu“. Například všechny objekty třídy Person mají vlastnost jméno. Tato vlastnost má však obecně pro různé osoby různé hodnoty – osoby mají různá jména. Konkrétní použití nějaké třídy se nazývá instance (objekt). Pomocí třídy tedy vytváříme objekty, které nesou kvalitativně stejné vlastnosti. Asagba a Ogheneovo (2008) uvádějí, že „objekty lze deklarovat pomocí vytvoření vzoru pro lokální stav a metody. Tento vzor se nazývá třída a v zásadě se jedná o datový typ“.

Princip SOLID

V objektově orientovaném programování značí SOLID mnemotechnickou zkratku pro pět principů návrhu, jejichž cílem je zajistit, aby byly softwarové návrhy srozumitelnější, flexibilnější a snadněji udržovatelnější. Principy jsou podmnožinou mnoha principů prosazovaných americkým softwarovým inženýrem a instruktorem Robertem C. Martinem, poprvé představeným ve své knize z roku 2000 Design Principles and Design Patterns (Martin, 2000).

Zkratka SOLID s popisem příslušné koncepce se poprvé objevuje v knize Martin, R. C. Clean Architecture: A Craftsman's Guide to Software Structure and Design (Martin, 2017)

Koncepty SOLID jsou

- Princip jediné odpovědnosti: „Nikdy by neměl existovat více než jeden důvod ke změně třídy." Tedy, každá třída by měla mít pouze jednu odpovědnost.
- Princip otevřenosti – uzavřenosti (open–close): „Softwarové entity (...) by měly být otevřené pro rozšíření, ale uzavřené pro modifikaci."
- Princip Liskovové substituce: „Funkce, které používají ukazatele nebo odkazy na základní třídy, musí být schopny používat objekty odvozených tříd, aniž by o tom věděly."
- Princip segregace rozhraní: „Mnoho rozhraní specifických pro klienta je lepších než jedno univerzální rozhraní."
- Princip inverze závislosti: „Závisí na abstrakcích, ne na konkrétním."

Ačkoli principy SOLID platí pro jakýkoli objektově orientovaný design, mohou také tvořit základní filozofii metodik, jako jsou agilní vývoj nebo adaptivní vývoj softwaru.

Princip jediné odpovědnosti Princip (zásada) jediné odpovědnosti poskytuje další podstatnou výhodu. Třídy, softwarové komponenty a mikroslužby, které

mají pouze jednu odpovědnost, lze mnohem snadněji vysvětlit, pochopit a implementovat než ty, které poskytují řešení pro všechno. Tím se sníží počet chyb, zlepší se rychlost vývoje a váš život vývojáře softwaru se výrazně usnadní.

Doporučení zní: nesnažme se však svůj kód zjednodušit za každou cenu. Někteří vývojáři posouvají princip jediné odpovědnosti do extrému vytvářením tříd pouze s jednou funkcí. Později, když chtějí napsat nějaký skutečný kód, musí vložit mnoho závislostí, díky nimž je kód velmi nečitelný a matoucí.

Proto je zásada jediné odpovědnosti důležitým pravidlem, aby byl náš kód srozumitelnější, ale nepoužívejme ho jako svoji programovací bibli. Při vývoji kódu používejme zdravý rozum. Nemá smysl mít více tříd, které obsahují pouze jednu funkci.

Princip otevřenosti – uzavřenosti Obecná myšlenka tohoto principu je skvělá. Řekne nám, abychom napsali svůj kód, abychom mohli přidat nové funkce beze změny stávajícího kódu. To zabrání situacím, ve kterých změna jedné z našich tříd také vyžaduje, abychom přizpůsobili všechny závislé třídy. Bohužel Meyer (1997) navrhuje k dosažení tohoto cíle využít dědičnosti:

> „Třída je uzavřena, proto může být kompilována, uložena v knihovně a používána klientskými třídami." Ale je také otevřená, protože ji může jakákoli nová třída používat jako nadřazenou a přidávat nové funkce. Když je definována třída potomka, není třeba měnit originál ani rušit jeho klienty. " (Meyer, 1997)

Ale jak jsme se v průběhu let dozvěděli a jak velmi podrobně vysvětlili další autoři, např. Robert C. Martin (Martin, 2000, 2009, 2017) ve svých publikacích o principech SOLID nebo Joshua Bloch ve své knize Java Efektivně (Bloch, 2002), dědičnost zavádí těsné propojení, pokud podtřídy závisí na implementačních podrobnostech o jejich mateřské třídě.

Proto Robert C. Martin (Martin, 2000, 2017) a další předefinovali princip otevřenosti – uzavřenosti na polymorfní princip otevřenosti – uzavřenosti. Použí-

vá rozhraní namísto nadtřídy, aby umožnil různé implementace, které můžeme snadno nahradit beze změny kódu, který je používá. Rozhraní jsou uzavřena kvůli úpravám a můžeme poskytnout nové implementace, které rozšíří funkčnost našeho softwaru.

Hlavní výhodou tohoto přístupu je, že rozhraní zavádí další úroveň abstrakce, která umožňuje volné propojení. Implementace rozhraní jsou na sobě nezávislá a není nutné sdílet žádný kód. Pokud považujeme za prospěšné, že dvě implementace rozhraní sdílejí nějaký kód, můžeme buď použít dědičnost (inheritance), nebo kompozici (composition).

Princip Liskovové substituce Princip substituce Liskovové představila Barbara Liskovová ve svém hlavním příspěvku na konferenci „Abstrakce dat“ v roce 1979 (Liskov, 1980). O několik let později vydala článek s Jeanette Wing (Liskov a Wing, 1994), ve kterém definovali princip jako:

„Nechť $\phi(x)$ je vlastnost prokazatelná o objektech x typu T. Pak $\phi(y)$ by mělo platit pro objekty y typu S, kde S je podtyp T. “ (Liskov a Wing, 1994)

Princip definuje, že objekty nadtřídy musí být vyměnitelné za objekty jejích podtříd bez porušení aplikace. To vyžaduje, aby se objekty vašich podtříd chovaly stejně jako objekty vaší nadtřídy. Toho můžete dosáhnout dodržováním několika pravidel, která jsou velmi podobná konceptu návrhu podle smlouvy definovaného (design by contract concept) Bertrandem Meyerem (Meyer, 1997).

Přepsaná metoda podtřídy musí přijímat stejné hodnoty vstupních parametrů jako metoda nadtřídy. To znamená, že můžete implementovat méně omezující pravidla ověřování, ale nemáte povoleno vynucovat přísnější pravidla ve vaší podtřídě. V opačném případě může jakýkoli kód, který volá tuto metodu na objektu nadtřídy, způsobit výjimku, pokud bude volán s objektem podtřídy.

Podobná pravidla platí pro návratovou hodnotu metody. Návratová hodnota metody podtřídy musí splňovat stejná pravidla jako návratová hodnota metody nadtřídy. Můžete se rozhodnout použít ještě přísnější pravidla vrácením konkrét-

ní podtřídy definované návratové hodnoty nebo vrácením podmnožiny platných návratových hodnot nadtřídy.

Princip segregace rozhraní Princip segregace rozhraní definoval Robert C. Martin (2017) při konzultacích se společností Xerox, která jim má pomoci vytvořit software pro jejich nové tiskové systémy. Jeho definice zní:

„Klienti by neměli být nuceni spoléhat na rozhraní, která nepoužívají. “ (Martin, 2017)

Podobně jako princip jediné zodpovědnosti je cílem principu segregace rozhraní snížit vedlejší účinky a frekvenci požadovaných změn rozdělením softwaru na několik nezávislých částí.

Princip inverze závislosti Obecná myšlenka tohoto principu je jednoduchá, ale velmi důležitá: Moduly na vysoké úrovni, které poskytují komplexní logiku, by měly být snadno znovu použitelné a neměly by být ovlivněny změnami modulů na nízké úrovni, které poskytují funkce obslužného programu. Abychom tohoto dosáhli, musíme zavést abstrakci, která od sebe odděluje moduly na vysoké a nízké úrovni.

Na základě této myšlenky se definice principu inverze závislostí podle Martina (2017) skládá ze dvou částí:

1. Moduly vysoké úrovně by neměly záviset na modulech nízké úrovně. Oba by měly záviset na abstrakcích.
2. Abstrakce by neměly záviset na detailech. Podrobnosti by měly záviset na abstrakcích.

Důležitým detailem této definice je, že moduly na vysoké a nízké úrovni závisí na abstrakci. Princip návrhu nemění jen směr závislosti, jak bychom možná očekávali při prvním čtení jeho názvu. Rozdělí závislost mezi moduly na vysoké a nízké úrovni zavedením abstrakce mezi nimi. Nakonec tedy získáme dvě závislosti (Martin, 2017):

1. modul na vysoké úrovni závisí na abstrakci a
2. nízká úroveň závisí na stejné abstrakci.

2.3 Objektově orientované programování ve výuce

Následující sekce poskytuje přehled metodiky zavádění základních konceptů programování obecně a zejména objektové orientace ve výuce. Je možno zkoumat různé přístupy a motivace pro zavádění různých přístupů k výuce programování. Může se jednat například o otázku: bude se objektově orientovaný koncept učit nejprve nebo později? Je třeba se zabývat nejprve aspektem modelování (návrhem) jako první nebo se jedná o aspekt, který je nedílnou součástí programování a jako takový bude probírán společně s programovacími koncepty? Další důležitý aspekt souvisí s jazykem a prostředím, v němž lze pracovat s programovacími koncepty. Existuje několik typů jazyků nebo prostředí, každé s výhodami a nevýhodami, které je třeba vzít v úvahu při plánování kurzu. Například v publikaci ACM/IEEE (ACM/IEEE-CS Joint Task Force on Computing Curricula 2013) se uvádí:

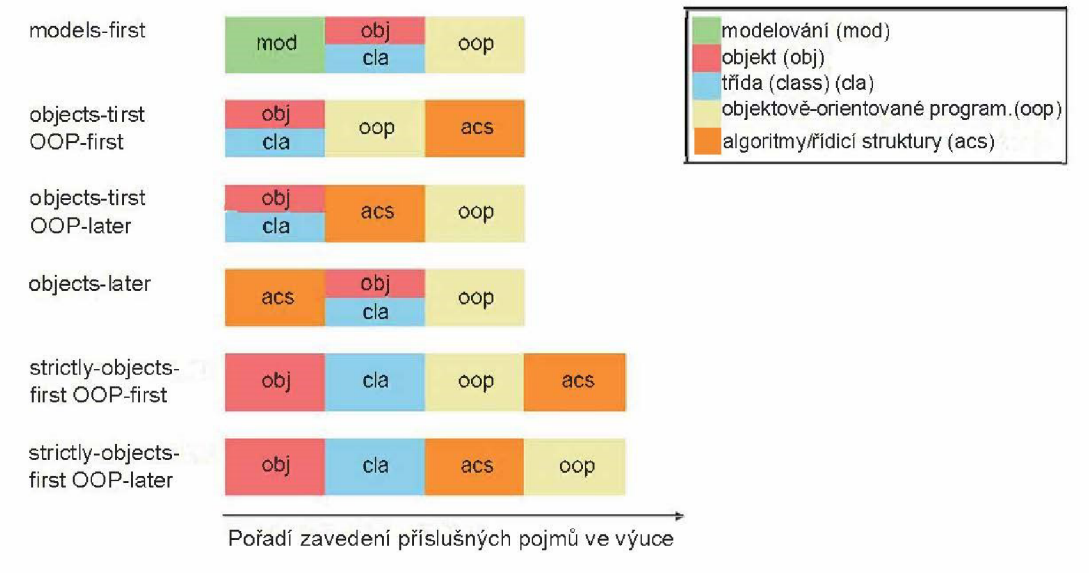
„Používání jazyka nebo prostředí určeného pro úvodní pedagogiku může studentům usnadnit učení, ale může mít omezené použití. Naopak, profesionálně používaný jazyk nebo prostředí může studenty brzy vystavit příliš velké složitosti. “ (JTFCC, 2013)

Velká pozornost byla věnována otázce vhodného vzdělávacího „paradigmatu“ pro zavedení objektové orientace. Například van Roy a kol. (2003) ve své práci píší:

> „Objektová orientace je někdy považována za pokročilé téma, které je těžké naučit. To může být pravda, pokud jej učíte studenty, kteří mají zázemí v jiném paradigmatu programování, ale naše zkušenost je, že OO je ideální použít jako první paradigma. “ (Roy a kol., 2003)

Podobně se vyjadřuje u nás zejména Pecinovský (viz např. Pecinovský, 2007; Pecinovský, 2009).

Existuje mnoho literárních zdrojů popisujících metodiku zavedení objektově orientovaného paradigmatu v úvodním kurzu. Většina z nich zahrnuje různé přístupy „objekty nejprve“ (objects-first) nebo „objekty později“ (objects-later). Například velmi podrobně se tímto tématem zabývá Kořínek (2017). Kromě čistých přístupů existují další metodologie pro zavedení programování. To ovlivňuje různé aspekty objektové orientace obecně (modelování nebo programování) a zejména dobu zavádění konkrétních programovacích konceptů. Obrázek 2.1 poskytuje přehled všech těchto přístupů. Modelování zahrnuje zavedení všech aspektů modelování (návrhu). Objekt nebo třída znamenají zavedení konceptů objektu a třídy, zatímco objektově orientované programování zahrnuje programovací pojmy objektově orientovaného paradigmatu. Nakonec struktury Algoritmus/Řízení zahrnují pojmy algoritmů a jejich implementace včetně řídicích struktur.



Obrázek 2.1: Přehled různých vzdělávacích „paradigmat“ pro pořadí zavádění objektové orientace a odpovídajících programovacích pojmů

Zdroj: upraveno dle (Berges, 2018)

Otázkou, kdy zavést objektově orientované pojmy, se zabývá ve své dizertační

práci Kořínek (2017). Zda je vhodné zavést jako první paradigma ve výuce imperativní (strukturované) programování nebo jako první zavést objektově orientované programování, popřípadě začínat jiným paradigmatem programování. Konstatuje, že vhodnější je pro sledované studenty koncepce výuky programování objects-first. Nicméně píše, že mezi skupinou studentů začínajících paradigmatem strukturovaného programování a skupinou studentů vyučovaných metodou object-first není na konci kurzu velký rozdíl.

Většina kurzů programování v terciární výuce organizované pro terciární obory se v poslední době změnila na přístup object-first, který je více či méně přísný. Objektově orientovaný přístup nicméně měl a má několik výhrad. Na začátku změny paradigmatu ve vzdělávání uvedli Decker a Hirshfield (1994) prvních deset výhrad a pokusili se diskutovat o tom, zda jsou omezení pravdivá, nebo se jen jedná o strach z nových věcí. Nakonec doporučují přístup object-first s určitými výhradami (Decker a Hirshfield 1994). Ve studii publikované o deset let později bylo provedeno nepřímé srovnání dvou výše popsaných hlavních přístupů. Decker (2003) zkoumal výkon dvou skupin studentů v objektově orientovaném kurzu. První skupina měla úvodní kurz podle přístupu „objekty nejprve“, zatímco druhá skupina začala s procesním paradigmatem na samém začátku (objekty později). Skupina, která od začátku používala objekty, si vedla lépe v pokročilejších objektově orientovaných konceptech, ačkoli druhé skupině byly také představeny objektově orientované koncepty. Kromě toho byl algoritmický výkon v obou skupinách stejný, což naznačuje, že oba přístupy jsou v procedurálních tématech úspěšné (Decker 2003). To v podstatě odpovídá i závěrům práce Kořínka (2017).

Jedním z autorů zabývajících se objektovou orientací v programování a způsobu, jak se ji naučit, je Lewis (2000). Píše o tom, že jeden z ústředních aspektů, má-li se student naučit objektově orientované programování je, že:

„Musí se rychle rozlišovat mezi prvotním psaním tříd, které definují objekty, a používáním objektů definovaných již existujícími třídami. “ (Lewis, 2000)

Píše dále na straně 246 (Lewis, 2000), že:

„Vyhodnocení jakéhokoli přístupu by se mělo zaměřit na souborovou prezentaci základního modelu. “ (Lewis, 2000, s. 246)

Podrobný výzkum v oblasti vyučování programování provedli Bennedsen a Schulte (2008). Jejich respondenti byli učitelé předmětu programování na různých typech škol, středních i vysokých. Jejich dotazy se týkali toho, jak studenti v předmětech programování rozumí konceptů objektově orientovaného programování. Jejich odpovědi jsou rozděleny do tří kategorií:

Používání objektů: „Na začátku kurzu student používá předem implementované objekty. Když student pochopil předmět koncepce, přejde k definování tříd sám. Důraz je kladen na využití před implementací. “ (Bennedsen a Schulte, 2008, s. 246)

Vytváření tříd: „Student definuje a implementuje třídy a vytváří instance definovaných tříd. Důraz je kladen na konkrétně kreativní část programování. “ (Bennedsen a Schulte, 2008, s. 246)

Koncepty: „Jedná se o výuku obecných principů a myšlenek objektově orientovaného paradigmatu se zaměřením nejen na programování, ale na vytváření objektově orientovaných modelů obecně. Důraz je kladen na konceptuální aspekty objektové orientace. “ (Bennedsen a Schulte, 2008, s. 246)

2.3.1 Koncepce výuky objektově orientovaného programování

V rámci výuky programování, které využívají přístup objektově orientovaného přístupu v programování od začátku výuky předmětu programování, můžeme rozlišit následující koncepce výuky:

- Objects-first
- Architecture First
- Design Patterns First
- Model-first

Objects-first

Zpráva ACM Computing Curricula (JTFCC, 2013, kapitola 7) uvádí čtyři přístupy k výuce programování na začátku druhého tisíciletí a konstatuje, že přístup „programování nejprve" je nejpoužívanějším přístupem ve výuce programování. Zpráva popisuje tři implementační strategie pro dosažení přístupu založeného na programování: imperativní (strukturovaný) na prvním místě, funkční na prvním místě a objektový na prvním místě. Zatímco první dvě strategie byly používány po nějakou dobu, je to strategie zaměřená na objekty, která v současné době přitahuje velký zájem. Studie uvádí:

„Objects-first důrazňuje principy objektově orientovaného programování a designu od samého počátku.... OOP začíná okamžitě pojmy objektů a dědičnosti.... poté pokračuje v zavádění radikálnějších řídicích struktur, ale vždy v kontextu zastřešujícího zaměření na objektově orientovaný design" (JTFCC, 2013).

Pecinovský (2007) uvádí, že tato koncepce výuky programování vznikla kolem roku 2000. Autor ve zmíněné studii uvádí, že:

„tato koncepce výuky programování prosazovala zahájení výuky přímo výkladem objektů a prací s nimi" (Pecinovský, 2007).

Autoři studie (JTFCC, 2013) uvádějí, že strategie object-first zvyšuje složitost výuky a učení v úvodních kurzech programování. Ve způsobu výuky pomocí strukturovaného paradigmatu je klasickou metodikou výuky pro úvod do programování začít s jednoduchými programy a postupně postoupit ke komplexním programovacím příkladům a projektům. Klasický přístup umožňuje poněkud jemnou křivku učení, poskytuje studentovi čas na postupné osvojování a budování znalostí. Strategie zaměřená na objekty má za cíl umožnit studentům okamžitou práci s objekty. To znamená, že studenti se musí ponořit přímo do tříd a objektů, jejich zapouzdření (veřejná a soukromá data, atd.), metod (konstruktorů), rozhraní atd. To vše je nutné kromě ještě zvládnutí obvyklých konceptů typů, proměnných, hodnot a odkazů, stejně jako detailů syntaxe. Přidávají se další koncepty založené na událostech na podporu interaktivity s grafickými uživatelskými roz-

hraními. Alphonce a Ventura (2002) uvádějí, že učení se programování objektovým způsobem nejprve vyžaduje, aby studenti pochopili „mnoho různých konceptů, nápadů a dovedností... téměř současně. Každá z těchto dovedností však představuje mentální výzvu." (Alphonce a Ventura, 2002)

V několika publikacích je přístup založený na objektech, s posunem paradigmatu od procesního k objektově orientovanému paradigmatu, považován za velmi obtížný a nepoužitelný v úvodních kurzech (Saeli a kol. 2011, s. 80).

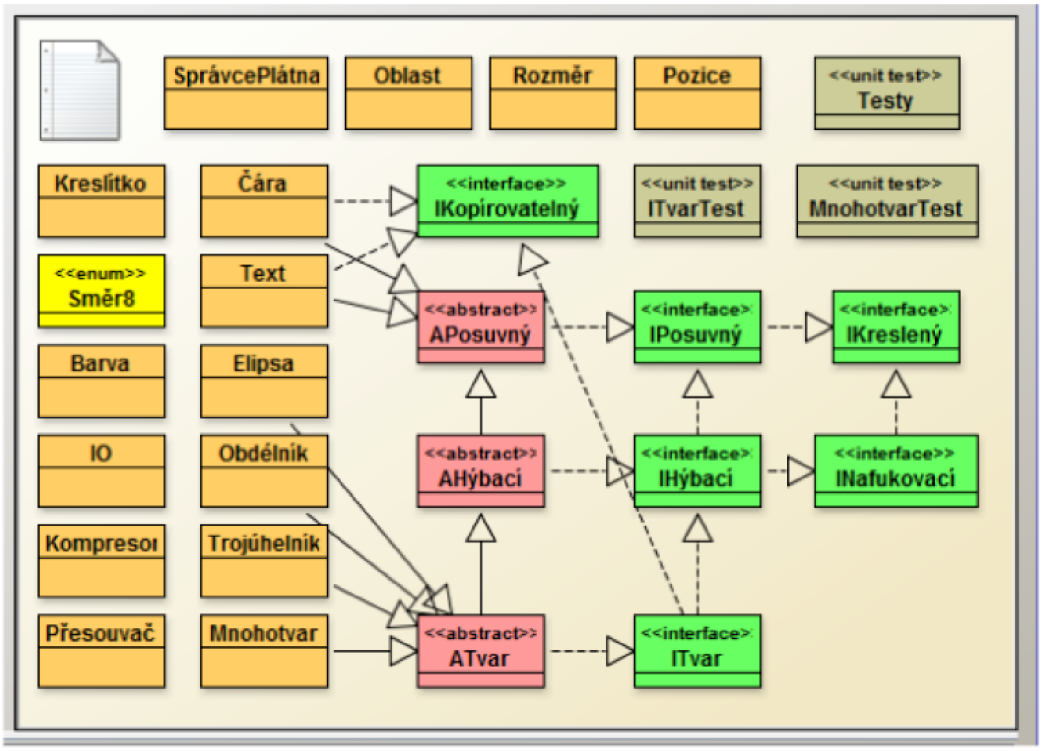
Diethelm (2007) představil přísnější verzi přístupu založeného na objektech. Uvádí, že většina problémů s orientací na objekty, a zejména s aspektem modelování, spočívá v použití tříd místo objektů. Dalším aspektem zmíněným Diethelmem (2007) je přístup, který používá techniky modelování pro zavedení objektově orientovaných pojmů. Tento přístup se nazývá „model-first". Tento přístup spočívá v tom, že objekty jsou nejprve modelovány samostatně a poté klasifikovány. Podle Diethelma (2007) modelování i programování by se měly zaměřovat spíše na objekty než na třídy. Tento přístup Diethelma (2007) se nazývá „přísně objektový".

Architecture first

Tvůrcem a propagátorem metodiky je Pecinovský. Autor tuto metodiku používá ve výuce na Vysoké škole Ekonomické v Praze. Metodiku představil např. v příspěvcích Pecinovský – Methodology Architecture First (2013a) nebo Pecinovský – Metodika Architecture First (2015).

Autor v této metodice navrhuje, aby nejdůležitější témata byla vysvětlena jako první, aby měl student co největší prostor pro osvojení tohoto tématu. Výuka programování podle metodiky Architecture First kurzy programování nezačínají kódováním, ale výukou architektury, tedy jakýmsi konceptuálním návrhem sestávajícím z jednotlivých komponent objektově orientovaného programování. Na začátku se studenti nemusejí rozptylovat pravidly syntaxe daného programovacího jazyka a mohou se soustředit na návrh správné architektury vytvářeného

programu. Navržené programy vytváří automatický generátor kódu, který je součástí vývojového prostředí. Na kódování se přechází až v okamžiku, kdy řešené problémy jsou již za hranicemi použitého generátoru kódu. V tomto okamžiku by již měl student mít osvojenou řadu znalostí a dovedností (Pecinovský, 2013).



Obrázek 2.2: Příklad projektu podle metodiky Architecture First
Zdroj: (Pecinovský, 2013)

Problémem tohoto přístupu (Architecture First) může být nutnost vyučovat ve výukovém prostředí, které je schopno generovat kód v optimální úrovni. Tento přístup není příliš rozšířený. Důvodem je zejména to, že pedagog musí tuto metodice dobře zvládat a aplikovat její postup ve výuce. Pro představu uvádíme zde obrázek (2.2), ukazující příklad projektu.

Design Patterns First

Tato koncepce výuky programování byla představena v některých článcích Pecinovského, zejména v jeho příspěvku Pecinovský (2006). O podstatě metodiky

autor (Pecinovský, 2006) píše:

„Jednou z důležitých vlastností metodiky je to, že studenti nemají většinou za úkol vytvářet nějaké samostatné, a proto nutně nesmírně jednoduché programy, ale že mají rozšiřovat funkčnost nějakého rozsáhlého (alespoň pro ně), předem připraveného projektu. Náplň výuky se tak mnohem více blíží potřebám praxe, protože převážnou většinou programátorských úkolů je právě rozšíření funkčnosti nějakého stávajícího projektu vytvořeného navíc většinou někým jiným." (Pecinovský, 2006).

Metodika Design Patterns First vylepšuje metodiku Object First. Jejím cílem vstupních kurzů programování je ještě více prohloubit seznamování začátečníků s moderními trendy programování. Studenti se seznamují s návrhovými vzory prakticky od samého počátku kurzu. Pecinovský (2006) uvádí, že:

„Moje zkušenosti ukazují, že uplatněním popsané metodiky je možné vštípit studentům zásady moderního programování mnohem efektivněji a pevněji. Studenti absolvující tyto kurzy se dokáží výrazně lépe orientovat ve složitých objektových programech, které mají doplnit o nějakou drobnou funkčnost, což bývá jedním z nejčastějších úkolů absolventů našich kurzů v praxi. Vedlejším, nicméně pro vyučujícího velmi příjemným efektem popsané metodiky je, že ve vstupních kurzech velice rychle sjednotí úroveň studentů, protože ani ti zkušenější studenti nebudou většinou umět použít jiné konstrukce, než ty přednášené." (Pecinovský, 2006).

Ke vzniku koncepce výuky programování Pecinovský (2010) uvádí, že

„...Design Patterns First se narodila v roce 2004 v reakci na nedostatky jiných, v té době používaných koncepcí výuky programování." (Pecinovský, 2010).

V článcích Pecinovský (2008) a Pecinovský (2010) uvádí zásady včetně detailního popisu jednotlivých fází výuky této koncepce výuky programování, které by se měly uplatňovat při návrhu programů a které koncepce výuky programování Design Patterns First splňuje.

Model-first

Synonymum přístupu založeného na objektech je zavedeno s představou, že začíná objektově orientovaný design. Modelování objektů je prvním krokem při zavádění objektově orientovaného přístupu. Kromě toho jsou koncepty jako dědičnost, polymorfismus a zobecnění zavedeny v návrhu (Adams a Frens 2003).

Přístup model-first spočívá v konceptuálním modelování, které předchází vytváření kódu. Zavedení různých jazykových konstruktů je podřízeno potřebám implementace daného konceptu v konceptuálním rámci. Po zavedení konceptu z konceptuálního rámce je zaveden odpovídající vzor kódování; kódovací vzor je vodítko pro překlad z UML do kódu prvku z konceptuálního rámce (Bennedsen Caspersen, 2004). Tento přístup model-first podporuje spirálové rozložení kurzu a v kurzu posiluje opakovaně nejdůležitější koncepty. Existují dvě kritéria pro návrh spirálového uspořádání: nejdříve jsou představeny nejběžnější koncepty konceptuálního rámce a v průběhu kurzu musí být studenti schopni vytvářet cvičné programy.

Výchozím bodem je třída a vlastnosti této třídy. Jednou z vlastností třídy může být přidružení k jiné třídě; v důsledku toho je dalším tématem asociace. To pěkně koreluje se skutečností, že asociace (reference) je nejběžnější strukturou mezi třídami (objekty). Složení je zvláštní případ sdružení; kompozice se vyučuje v dalším kole spirály. Poslední strukturou, kterou je třeba důkladně pokrýt, je specializace. Specializace je nejméně běžnou strukturou v konceptuálních modelech a pěkně přechází do druhé poloviny kurzu, kde je kladen důraz na kvalitu a design softwaru.

Na začátku této koncepce výuky programování je využíváno vývojové prostředí, např. Visual Studio jako vývojové prostředí pro programovací jazyk C♯ nebo jiný modelovací jazyk, např. UML. Jak Visual Studio tak i jazyk UML obsahuje několik typů diagramů, které mají různé využití pro návrh programů. Teprve po návrhu programu pomocí různých diagramů přichází výklad základních OO konstrukcí. Tato koncepce výuky programování po úvodním modelování přechází do

koncepce výuky programování objects-first, protože po objektově orientovaných konceptech se dále vykládají základní konstrukce programování.

Moderní vývojová prostředí, jako například MS Visual Studio a další podporují tuto koncepci výuky programování. Jejich součástí je nástroj umožňující vytváření diagramů tříd a jiných schémat sloužících pro návrh příslušné aplikace a po vytvoření takového modelového konceptu i vygenerování základního zdrojového kódu.

2.3.2 Vhodné pořadí pro zavedení konceptů objektově orientovaného přístupu

Jeden z možných přístupů založených na „programování na prvním místě" představují Alphonce a Ventura (2002). V tomto přístupu je objektová orientace zavedena designem. Zmínili, že se pokusili vyřešit slabiny přístupu „OOP-first", který je zmíněn v osnovách ACM / IEEE z roku 2013 (JTFCC, 2013). Autoři (Alphonce a Ventura, 2002) uvádějí:

"Všechny úvodní přístupy diskutované v CC2001 mají odlišné silné a slabé stránky. Přístup založený na prvním programování trpí dalšími slabostmi nad rámec výše uvedených. CC2001 konstatuje, že přetrvala díky některým významným silným stránkám, zejména proto, že studenti jsou již v prvním ročníku vybaveni programátorskými dovednostmi a že následující kurzy mohou tyto dovednosti využít ve svých učebních osnovách a mohou tyto dovednosti také zdokonalit. " (Alphonce a Ventura, 2002).

Tito autoři také zdůrazňují význam modelování a využití jazyku Unified Modeling Language (UML). Popisují například význam modelování (návrhu, designu) takto (Alphonce a Ventura 2002):

„V rámci úkolů raného programování dostávají studenti design vyjádřený v UML spolu s kosterním řešením. Úkolem studentů je dodat vlastní kód pro dokončení implementace daného návrhu. V pozdějších úkolech jsou studenti vybaveni po-

pisem požadavků v prostém jazyce, ze kterého musí před zahájením psaní kódu vyvinout design vyjádřený v UML.“ (Alphonce a Ventura, 2002).

První obecné rozhodnutí v rámci výuky objektově orientovaného přístupu, které je třeba učinit, je o pořadí programování a modelování. Bennedsen a Caspersen (2004) představují přístup založený na modelu. Začínají modelováním tříd a objektů v programovací úloze těsně před napsáním prvního řádku kódu. Programovací koncepty jsou představeny později v rámci určitého kaskádového přístupu. Takový přístup zdůrazňuje, že objektová orientace není jen dalším řešením nebo technologií pro problémy s programováním.

Autoři Bennedsen a Caspersen (2004) dále uvádějí:

”Výchozím bodem je třída a vlastnosti této třídy. Jednou z vlastností třídy může být přidružení k jiné třídě; v důsledku toho je dalším tématem asociace. To pěkně koreluje se skutečností, že asociace (reference) je nejběžnější strukturou mezi třídami (objekty). Složení je zvláštní případ sdružení; kompozice se vyučuje v dalším kole spirály. Poslední strukturou, kterou je třeba důkladně pokrýt, je specializace. Specializace je nejméně běžnou strukturou v konceptuálních modelech a pěkně přechází do druhé poloviny kurzu, kde je kladen důraz na kvalitu a design softwaru.“ (Bennedsen a Caspersen, 2004).

Autoři ve svém příspěvku popisují v podstatě přístup model-first a nacházejí v něm výhody: systematický přístup k programování, hlubší pochopení programovacího procesu a zaměření na obecné programovací koncepty namísto jazykových konstruktů v konkrétním programovacím jazyku.

Další příklad iterativního přístupu založeného na modelování a konstrukčních aspektech uvádějí Hadar a Hadar (2007). Zdůrazňují abstraktní modelování a návazně konkrétní implementaci s tím, že tyto fáze se prolínají. Tímto přístupem se snaží zdůraznit úkoly abstraktního myšlení. Změna úkolů modelování a programování by měla zabránit nutnosti zvolit jeden z výše uvedených přístupů (Hadar a Hadar, 2007).

Výuka objektově orientovaných konceptů může být provedena na začátku kur-

zu nebo později, po vysvětlení některých řídicích konstruktů. Podle toho se může lišit ve výuce programování pořadí objektově orientovaných konceptů probíraných v takových kurzech. Touto problematikou se zabývá např. Ehlert (2012). Pojednává o důležité otázce objektově orientovaného programování ve výuce informatiky. Poskytuje přehled typických konceptů zavedení objektově orientovaného programování (OOP) a jejich pořadí v rámci kurzu (Ehlert a Schulte 2009). Podobné koncepty a jejich řazení lze najít i v dizertační práci Kořínka (2017). Tato témata zahrnují pojmy obou programovacích paradigmat a pokrývají hlavní položky objektově orientovaných konceptů: objekty a třídy, datové typy, proměnné a atributy, konstanty, metody a postupy, řídicí struktury a dědičnost a sdružení. Přehled různých pořadí je uveden v tabulce 2.1.

Tabulka 2.1: Srovnání témat přístupů OOP-first a OOP-later a pořadí jejich zavedení

OOP-first	OOP-later
Třídy a objekty	Datové typy (vč. proměnných, konstant)
Atributy (vč. datových typů)	Řídící struktury
Metody (vč. řídících struktur)	Procedury
Dědičnost	Třídy a objekty
Kompozice	Dědičnost a kompozice

Zdroj: (upraveno dle Ehlert a Schulte, 2009)

Ehlert (2012) i Kořínek (2017) uvádí, že OOP-first se soustředí na stejná témata jako OOP-later. Jediným rozdílem je jejich pořadí. Kořínek (2017) podobně jako Ehlert (2012) zkoumá možné rozdíly ve dvou přístupech, OOP nejprve a OOP později, týkající se výsledku souvisejícího se zavedením objektově orientovaného programování. Výsledkem je, že mezi těmito dvěma přístupy nejsou žádné významné rozdíly. Jediným hlavním faktorem odlišujícím výsledek je osobní preference učitele.

Kořínek (2017) například píše:

„V jednotlivých akademických letech vyučoval každou skupinu jiný vyučující. Analýzou získaných dat z dotazníkového šetření bylo zjištěno, že v akademickém roce 2014/2015 mohl mít na výsledky studentů v jedné ze skupin vliv vyučující. V předešlém akademickém roce vliv vyučujícího na výsledky studentů v obou skupinách nebyl prokázán. Na výsledky studentů v jednotlivých akademických letech mají vliv studenti, kteří předmět opakovali z minulého akademického roku nebo se studenti účastnili pouze posttestu.

V akademickém roce 2013/2014 byl mezi výsledky studentů v jednotlivých částech posttestu prokázán statisticky významný rozdíl. V retestu 1 již mezi výsledky studentů statisticky významný rozdíl prokázán nebyl. Vyšší hodnotu skóre výsledků v obou testech získali studenti experimentální skupiny, kteří se vyučovali podle koncepce výuky programování objects-first. V následujícím akademickém roce nebyl mezi výsledky studentů v jednotlivých částech testů prokázán statisticky významný rozdíl. Vyšší hodnotu skóre výsledků v testech získali studenti kontrolní skupiny studentů, kteří byli vyučovaní podle koncepce výuky programování objects-later“ (Kořínek, 2017).

Kromě malé velikosti vzorku je dalším problémem v práci Ehlerta (2012), stejně jako u jiných v literatuře k danému tématu (např. i Kořínek, 2017), chybějící rozdíl mezi object-first nebo object-later a OOP-first nebo OOP-later. Zatímco první přístupy se vztahují k pojmům objektů a tříd a jejich postavení v úvodním kurzu, druhé dva přístupy se týkají programovacích aspektů orientace na objekty, jako je dědičnost a polymorfismus, stejně jako koncept metody a atributu (Diethelm 2007).

2.3.3 Vhodný jazyk pro úvodní výuku objektově orientovaného programování

Z předchozích odstavců je mimo jiné zřejmé, že naučit se programovat objektově, je více, než učit se nový jazyk, který je objektově orientovaný. V posledních něko-

lika letech byla vyvinuta celá řada jazyků, které podporují nebo přímo vyžadují objektový přístup při vytváření různých programových aplikací.

Například v dokumentu ACM/IEEE-CS Joint Task Force on Computing Curricula (JTFCC, 2013) se píše:

„Využívání „bezpečnějších" nebo více spravovaných jazyků a prostředí může pomoci lešení studentů. Tyto jazyky však mohou poskytovat úroveň abstrakce, která zakrývá pochopení skutečného provedení stroje a je obtížné vyhodnotit kompromisy výkonu. Rozhodnutí, zda použít jazyk „nižší úrovně" k podpoře konkrétního mentálního modelu provádění programu, který je blíže skutečnému provedení strojem, je často záležitostí potřeb místního publika" (JTFCC, 2013).

Postupně od 90. let docházelo ke změně programovacího paradigmatu směrem k objektově orientovanému programování. Tato změna nejprve proběhla v IT průmyslu (důvody viz kapitola Úvod).

Na začátku této změny paradigmatu na počátku 90. let existovalo pouze několik objektově orientovaných jazyků, které by byly použitelné pro vzdělávací účely. Proto se v prvních letech používaly jazyky z průmyslu, které měly převážně procedurální kořeny. Postupem času byly vyvinuty výukové jazyky a prostředí, které byly cílené na oblast výuky v objektově orientovaném paradigmatu. První programovací jazyky však byly převzaty z IT průmyslu a výběr byl podložen nutností připravit studenty na potřeby IT průmyslu (Pears a kol. 2007). Pro účely výuky byly vyvinuty programovací jazyky jako úprava běžného průmyslového programovacího jazyka, nebo prostředí upravené pro vzdělávací účely. Postupně vznikla množina jazyků speciálně pro vzdělávací účely. Většina z těchto jazyků poskytuje grafické programovací prostředí.

Kategorie jazyků s průmyslovými kořeny historicky souvisí s vývojem programovacího paradigmatu. Lewis (2000) uvádí ve své práci o mýtu o objektové orientaci dvou široce rozšířené jazyky v úvodních kurzech. Prvním jazykem je C++, což je hybridní jazyk. Lewis (2000) píše:

„ "Hybridní povaha C++ je jedním z problémů, které mají pedagogové s rozhod-

nutím učinit z C++ jazyk volby, který se vyučuje v úvodních programovacích kurzech". " (Lewis, 2000)

Druhým jazykem je Java, která je nejrozšířenějším jazykem. I když je Java spíše objektově orientovaný, nejde o čistý objektově orientovaný jazyk. Lewis (2000) uvádí:

"Existence primitivních typů je jedním z důvodů".

„...Ještě důležitější je, že požadavek, aby všechny metody byly definovány jako součást třídy, nevede automaticky k objektově orientovanému návrhu. Lze připustit, aby dominovala procedurální abstrakce, která by zničila konceptuální eleganci správného designu" Lewis (2000).

Kromě toho existují jazyky navržené speciálně pro výuku programování objektově orientovaným paradigmatem, ale na rozdíl od C++, Javy nebo C♯ nenalezly široké uplatnění. Zástupcem, který byl navržen s ohledem na vzdělání je Smalltalk nebo jazyk Eiffel. Existuje několik dalších jazyků, které jsou však méně známé.

Další sada jazyků zahrnuje ty, které souvisejí s jedním z výše uvedených, ale většinou pouze s podmnožinou konceptů. Programové výukové prostředí BlueJ je založeno na Javě, ale vynechává koncept hlavní metody. BlueJ vznikl na University of Sydney a Monash University v 90. letech 20. století jako nový programovací jazyk (The Blue Page, 2021), který byl primárně určen pro výuku objektově orientovaného programování pro studenty prvních ročníků. Jeho součástí je integrované vývojové prostředí s grafickým zobrazením tříd, editorem zdrojového kódu, nástrojem pro ladění, knihovnami a dalším. Výuku objektového programování pomocí jazyka BlueJ popisuje Pecinovský ve svých knihách, např. (Pecinovský, 2009, 2013b).

Jiný starší jazyk je robot Karel, který byl navržen v 80. letech 20. století a byl zaměřen na paradigma programování v dané době. Srovnání robota Karla a BlueJ popisuje Borge (2004). Alternativou, zejména pro první kontakt s orientací na objekt na samém začátku úvodního kurzu, je knihovna Java, která poskytuje jed-

noduché geometrické objekty, které lze snadno zobrazit. Knihovnu „ObjectDraw“ podrobně popisuje Bruce a kol. (2001).

V sadě jazyků určených pro vzdělávací účely existují některé jazyky, které se používají pouze v několika kurzech, a jiné, které se používají po celém světě. Téměř všechny příklady mají společné to, že poskytují určitý druh prostředí, jako je hra nebo film, kde mohou objekty interagovat navzájem nebo s prostředím. Schopnosti interakce a druh objektů se v různých jazycích výrazně liší. Zatímco jazyk, jako je Alice, poskytuje 3D prostředí a velkou sadu možných objektů a interakcí (Sattar a Lorenzen 2009), Robot Karel má jen několik objektů a omezenou scenérii, kde může robot jednat. Další rozdíl v jazycích je v samotném programování. Zatímco Robot Karel má své vlastní programovací jazyky, Scratch nebo Alice poskytují předdefinované programovací prvky, které lze měnit a kombinovat.

Programovací jazyk Alice je zaměřen na středoškolské studenty, kteří nemají žádné zkušenosti s programováním. Obsahuje řadu návodů, které provedou uživatele základními operacemi vývojového nástroje, aby mohl samostatně zkoušet a vytvářet projekty. Možnosti výsledného projektu jsou omezené množstvím objektů, které obsahuje knihovna. Nové objekty je možné přidávat, není ale možné využít možností programovacího jazyka Java.

Využití Alice jako programovacího jazyka v úvodních kurzech ukázalo výhody tohoto jazyka pro objektové programování. Cooper a kol. (2003) pozorovali silnou kontextualizaci objektů a tříd. Dále jsme zdůraznili pojem zapouzdření, protože objekty lze v prostředí přesouvat, ale prostorové souřadnice nejsou přímo přístupné. Uvádějí, že tato skutečnost navíc poukazuje na smysl předávání zpráv mezi objekty. Prostředí této sady nicméně poskytují použité ovládací struktury pro výběr, a proto skryjí veškerou syntaxi. Problémy se syntaxí se neprojevují, což je zpočátku výhodou, ale je třeba je vzít v úvahu v pozdějším vzdělávacím procesu.

Existuje dále velké množství jazyků pro výuku programování, lze zmínit např. LOGO (Logo, 2021) nebo Scratch (Scratch, 2021) a další. Scratch je vizuální pro-

gramovací jazyk, který umožňuje vytvářet programy manipulací s grafickými programovými elementy. Uživatelům dovoluje používat programování řízené událostmi s různými aktivními objekty.

2.3.4 Kompetenční modely pro objektově orientované programování

Ačkoli vydávání vzdělávacích standardů a doporučení učebních osnov v informatice má dlouhou tradici, kompetenční modely nejsou příliš ve výuce zavedeny. Weinert (2001) definuje kompetenci jako „hrubě specializovaný systém schopností, dovedností nebo dovedností, které jsou nezbytné k dosažení konkrétního cíle. To lze aplikovat na individuální dispozice nebo na distribuci těchto dispozic v rámci sociální skupiny nebo instituce. V posledních letech existuje několik snah o rozvoj kompetenčních modelů. Lze zmínit např. návrh, který popsali ve své práci model Bennedsena a Schulteho (2013) nebo projekt MoKoM (Magenheim, Nelles, Rhode a Schaper 2010. Asi nejvlivnější kompetenční model v oblasti počítačové vědy je kompetenční model vydaný společnou pracovní skupinou ACM/IEEE v roce 2020 (IS2020, 2020).

První verze byla vydána v roce 2001. V dalších letech byla revidována a nová verze byla zveřejněna v roce 2020. Podobně jako v předchozích svazcích má nový svazek revidovaný soubor znalostí a představuje několik příkladů kurzů, které implementují kurikulum.

Existuje 18 znalostních oblastí, které tvoří znalostní základ osnov. Tyto oblasti znalostí by neměly tvořit konkrétní kurzy v osnovách. Přesněji řečeno, většina kurzů vybírá témata z několika oblastí znalostí. Abychom mohli poradit, kterým tématům by se měli věnovat všichni studenti a která témata by měla být vyhrazena pro speciální kurzy, jsou témata identifikována na určité úrovni. Jsou buď „základní“, nebo „volitelné“, a pokud jsou základními prvky, mohou být „úrovní 1“ nebo „úrovně 2“. Pokud jsou jádrem úrovně 1, měli by je pokrýt všichni stu-

denti. Jádro úrovně 2 by měl pokrývat téměř každý student a navíc by měly být poskytnuty volitelné prvky, ale nemusí být pokryty všemi studenty. Pouhé pokrytí základních prvků však pro postgraduální studium nestačí.

Kromě oblastí znalostí se provádí subkategorizace. Znalostní jednotky poskytují témata a výsledky učení, které mají určité úrovně zvládnutí, které jsou převzaty z Bloomovy taxonomie (Bloom, 1956). Tyto úrovně zvládnutí jsou: znalost, použití, vytváření, analýza a hodnocení (viz IS2020, 2020).

Pokrytí orientace objektu ve znalostních oblastech nebo znalostních jednotkách lze vidět v následujících seznamech. První seznam zobrazuje témata související s orientací na objekty a odpovídající znalostní oblasti a znalostní jednotky. Tučná témata jsou znalostní jednotky s odpovídajícími znalostními oblastmi. Každá položka seznamu v IS2020 A Competency Model for Undergraduate Programs in Information Systems 2020 (IS2020, 2020) je doprovázena číslem stránky v závorkách. Úroveň je navíc přidána do hranatých závorek a pro výsledky učení jsou uvedeny úrovně zvládnutí.

V oblasti kompetencí pro objektově orientované programování je v publikaci IS2020 (2020) uvedeno:

„Většina softwarových implementací přesahuje jednoduché využití programovacích konstrukcí a směřuje k využití modulárních komponent, které jsou často postaveny proti paradigmatickým osvědčeným postupům pro rozšiřitelnou a zvládnutelnou konstrukci. Programovací paradigmata jsou často idiomatická a vytvářejí epistemologické hodnoty týkající se strukturování aplikací, knihoven opakovaně použitelných kódů a vzorů, které vedou k architektonickým rozhodnutím. Objektová orientace je paradigmatický pohled na to, jak organizovat data a rutiny do knihoven opakovaně použitelného kódu zaměřeného na organizaci dat a rutin do kontejnerů nazývaných třídy (pro specifikaci) a objekty (pro vytváření instancí). Sada pravidel chování obsažených v těchto strukturách, která určuje, jak skupiny a hierarchie těchto entit interagují, tvoří základ objektové orientace, který prostupuje nejuznávanějšími architektonickými vzory pro vývoj softwaru

a systémů. Objektová orientace, i když je vlastní současným programovacím jazykům, slouží také jako základ pro modelování problémových domén, které přesahují aplikace v programování. Velká část tohoto materiálu bude tedy obsažena ve většině kurzů systémové analýzy a designu. Proto se zde zaměřujeme na manifestované aplikace, které sahají od návrhu po implementaci, zatímco analýza a návrh systémů bude uvedena v jiné části" (IS2020, 2020).

Absolvent úvodních kurzů programování by podle této publikace IS2020 (2020), strana 136 a dále) měl zvládnout následující oblasti objektově orientovaného programování a měl by je schopen aplikovat ve svých úlohách:

1. Aplikovat základní prvky objektů a tříd.
2. Využít možnosti vytváření instancí.
3. Využít komunikaci a zasílání zpráv uvnitř entity.
4. Zapouzdření.
5. Správa dědičnosti a závislostí.
6. Abstrakce.
7. Aplikovat polymorfismus.
8. Využít návrhové vzory.
9. Využít objekty a třídy pro modelování entit.

Pro každý koncept je pak podrobněji popsáno, co by měl absolvent zvládnout a do jaké úrovně Bloomovy taxonomie.

Například pro oblast objektově orientovaného programování 5. Správa dědičnosti a závislostí (IS2020, 2020) jsou tyto požadované kompetence popsány v tab. 2.2.

Tabulka 2.2: Příklad kompetencí pro objektově orientované programování

Znalost konceptu	Úroveň dovedností (Bloomova kognitivní úroveň)
Rozšiřitelnost objektu a opětovné použití prostřednictvím dědičnosti („je druh z")	6 - Vytvořit
Jedno a vícenásobné dědictví	3 - Použít
Rozšiřitelnost objektu a opětovné použití prostřednictvím kompozice („má")	3 - Použít
Závislosti a konstruktory	3 - Použít
Injekce závislostí a inverze řízení	3 - Použít
Rozhraní jako alternativa k vícenásobné dědičnosti	3 - Použít
Rozšiřitelnost objektu a opětovné použití prostřednictvím delegování („zná")	3 - Použít

Zdroj: (upraveno dle IS2020, 2020)

Pro každou oblast objektově orientovaného programování jsou takto definovány požadované znalosti, které by měl student zvládat na určité úrovni.

2.4 Metriky složitosti kódu a kvalita softwaru

Objekt, třída, metoda, instanční proměnná, dědičnost a systém zpráv jsou hlavními koncepty objektově orientovaného přístupu k programování. Metrikami objektově orientovaného přístupu rozumíme především způsoby měření a zjišťování, jak jsou tyto konstrukce používány v procesu návrhu a vývoje aplikací.

Množství funkcí dostupných v objektově orientovaném aplikaci lze odhadnout na základě počtu definovaných tříd a metod a jejich obměn. Proto se základní metriky objektově orientovaného programování (OOP) vztahují ke třídám

a metodám a jejich velikosti, počtu logických řádek kódu (lines of code – LOC), nebo funkčních bodů tříd a metod. Pro měření návrhu a komplexnosti programu je nutné, aby metriky řešily rovněž specifické charakteristiky OOP, jako jsou dědičnosti, instanční proměnné, provázanost entit, použité abstrakce a zapouzdření.

V dnešní době existuje řada různých nástrojů pro měření metriky kódu. K dispozici jsou metriky pro měření různých aspektů kódu, od velikosti kódu, přes složitost kódu, chyby kódu, až po metriky výkonu kódu. Jedním z prvních vývojářů kódovacích metrik byl Thomas McCabe (1976), který vyvinul techniku, označovanou pojmem cyklomatická složitost, která měří řídící struktury softwaru. Tato metrika charakterizuje software pomocí číselného vyjádření. Měření programu se provádí pomocí grafu, který modeluje tok řízení daného kódu. Uzly grafu jsou neoddělitelné skupiny příkazů a orientované hrany reprezentují pořadí provádění jednotlivých příkazů. Cyklomatické číslo úseku kódu je pak vypočteno jako počet nezávislých cest v tomto úseku kódu. Pomocí těchto metrik je možné identifikovat softwarové moduly, třídy nebo metody, které testovat jinak by bylo velmi obtížné. Softwary nízké kvality a náchylné k chybám často vykazují vysokou cyklomatickou složitost.

Od doby, kdy McCabe vyvinul první metriku softwaru, byly vyvinuty mnohé další metriky, které zkoumají různé aspekty softwaru:

- Halsteadovy metriky komplexnosti kódu
- Metriky počtu řádků
- Objektově orientované metriky
- Booleovy metriky

Halsteadova (1977) metrika složitosti kódu měří výpočetní složitost kódu a je založena na měření operátorů v programu.

Počet řádků v programu je jedním z nejzákladnějších kódovacích metrik. Jako taková tato metrika sama o sobě nenabízí velkou hodnotu, pokud je brána samo-

statně. Tato metrika je také jednou z nejvíce chybně používaných metrik. Mnoho vývojářských organizací měří počet řádků v rámci jednoho projektu a pak použije tento údaj jako výchozí hodnotu pro měření budoucích projektů a výkonu vývojářů. Pro vývojáře je velmi snadné vytvářet obrovské množství řádků kódů. Ty však mají špatnou kvalitu a vykazují rovněž velmi špatné výsledky vůči jakékoli jiné metrice. Nicméně počet řádků kódu, při použití ve spojení s jinými metrikami, přispívá k pochopení programu. Obecně řečeno, menší moduly (s menšími počty řádků) budou snadněji pochopitelné a lépe se budou v budoucnu udržovat.

Lorenz a Kidd (1994) navrhnuli jedenáct metrik, které mají určovat návrhové ukazatele v OOP.

Pravidla, která pro metriky stanovil, jsou shrnuty v tabulce 2.3.

Tabulka 2.3: Metriky OOP a jejich pravidla

Metrika	Pravidla
1. Průměrná velikost metody (LOC)	Mělo by být méně než 8 LOC pro Smalltalk, méně než 24 LOC pro C++.
2. Průměrný počet metod ve třídě	Mělo by být méně než 20. Vyšší hodnota může znamenat, že třída má větší odpovědnost s příliš málo třídami.
3. Průměrný počet instančních proměnných ve třídě	Mělo by být méně než 6. Větší počet instančních proměnných značí, že jedna třída provádí více akcí než by měla.
4. Úroveň vnoření v hierarchii tříd (hloubka stromu dědičnosti, DIT)	Mělo by být méně než 6. Počínaje třídami použitého frameworku nebo hlavní třídou programu.
5. Počet subsystémů a vztahů mezi nimi	Mělo by být méně než 6.

6. Počet tříd a vztahů mezi nimi v každém subsystému	Hodnota by měla být relativně vysoká. Tato položka se týká vysoké soudržnosti tříd ve stejném subsystému. Pokud jedna nebo více tříd subsystému neinteraguje s mnoha dalšími třídami, může být lepší je přesunout do jiného subsystému.
7. Užití instančních proměnných	Pokud skupiny metod ve třídě používají různé sady instančních proměnných, zvažte, zda by v rámci těchto pomocných řádků neměla být třída rozdělena do více tříd.
8. Průměrný počet komentářů (v rámci metod)	Počet by měl být větší než 1.
9. Počet chybových hlášení v rámci třídy	Hodnota by měla být nízká (nejsou uvedena další upřesnění).
10. Počet znovupoužití třídy	Pokud se třída v různých aplikacích (zejména v abstraktní třídě) nepoužívá opakovaně, může být potřeba ji přepracovat.
11. Počet odstraněných tříd a metod	Hodnota by měla být ustálená po většinu vývojového cyklu. Pokud tomu tak není, pravděpodobně vývoj používá přírůstkový typ návrhu namísto iterativního OO vývoje.

Zdroj: (upraveno dle Lorenz a Kidd, 1994)

Detailnější soubor metrik, který sestavili autoři Kasto a Whalley (2013), uvádí tab. 2.4.

Tabulka 2.4: Softwarové metriky a jejich použitelnost v programovacích paradigmatech

Typ metriky	Metrika	Programovací paradigma		
		imperativní	procedurální	objektově orientované
Základní	Počet řádků kódu	×	×	×
	Počet prázdných řádků kódu	×	×	×
	Počet řádků komentářů v kódu	×	×	×
	Počet slov v komentářích	×	×	×
	Počet příkazů	×	×	×
	Počet metod		×	×
	Průměrný počet řádků kódu v metodě		×	×
	Počet parametrů	×	×	×
	Počet vložených příkazů		×	×
	Počet argumentů		×	×
	Počet metod ve třídě			×
	Počet odkazovaných tříd			×
	Průměrný počet atributů ve třídě			×
	Počet konstruktorů			×
	Průměrný počet konstruktorů ve třídě			×

Metrikysložitosti	Hustota kódu	×	×	×
	Cyklomatická složitost	×	×	×
	Hloubka vnořených bloků	×	×	×
	Počet operandů	×	×	×
	Počet operátorů	×	×	×
	Počet unikátních operandů	×	×	×
Halsteadovymetriky	Počet unikátních operátorů	×	×	×
	Úsilí implementace		×	×
	Čas implementace		×	×
	Délka program		×	×
	Úroveň program		×	×
	Objem program		×	×
	Index udržitelnosti		×	×
Objektověorientované metody	Váha metody na třídu			×
	Odpovědnost třídy			×
	Nedostatek soudržnosti			×
	Provázanost mezi třídami objektů			×
	Hloubka stromu dědičnosti			×
	Počet potomků			×

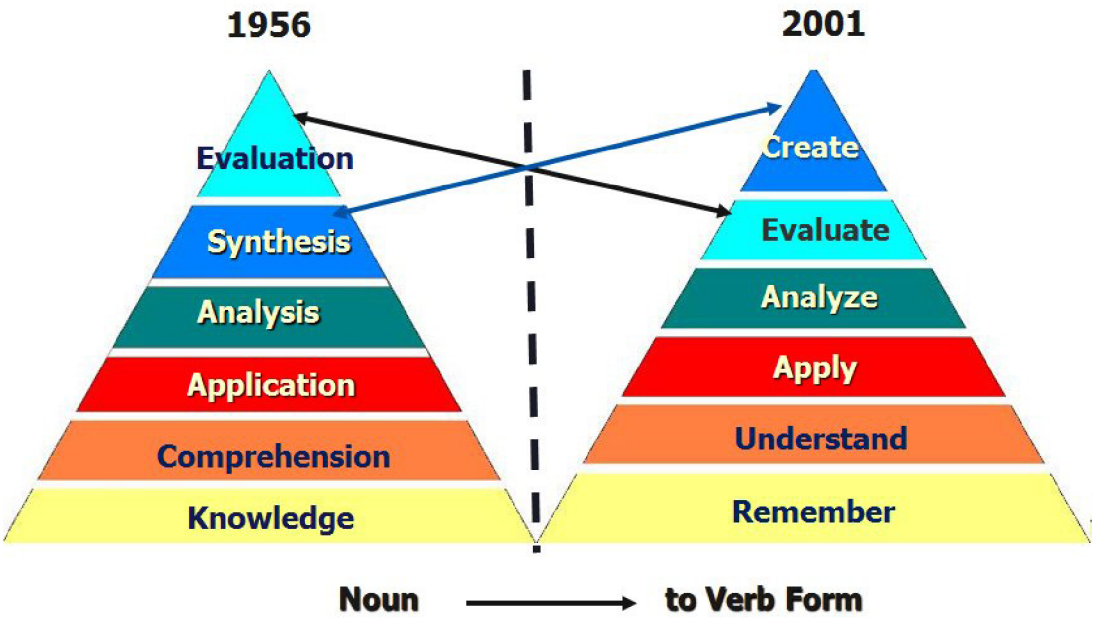
Zdroj: (upraveno dle Kasto a Whalley, 2013, p. 60)

Je pravděpodobné, že hodnoty metrik budou korelovat s rostoucí obtížností úlohy. To je dáno skutečností, že mnohé softwarové metriky se snaží měřit složi-

tost kódu. Lze předpokládat, že čím složitější je kód, tím je obtížnější jej naprogramovat.

2.5 Alternativy k metrikám složitosti kódu

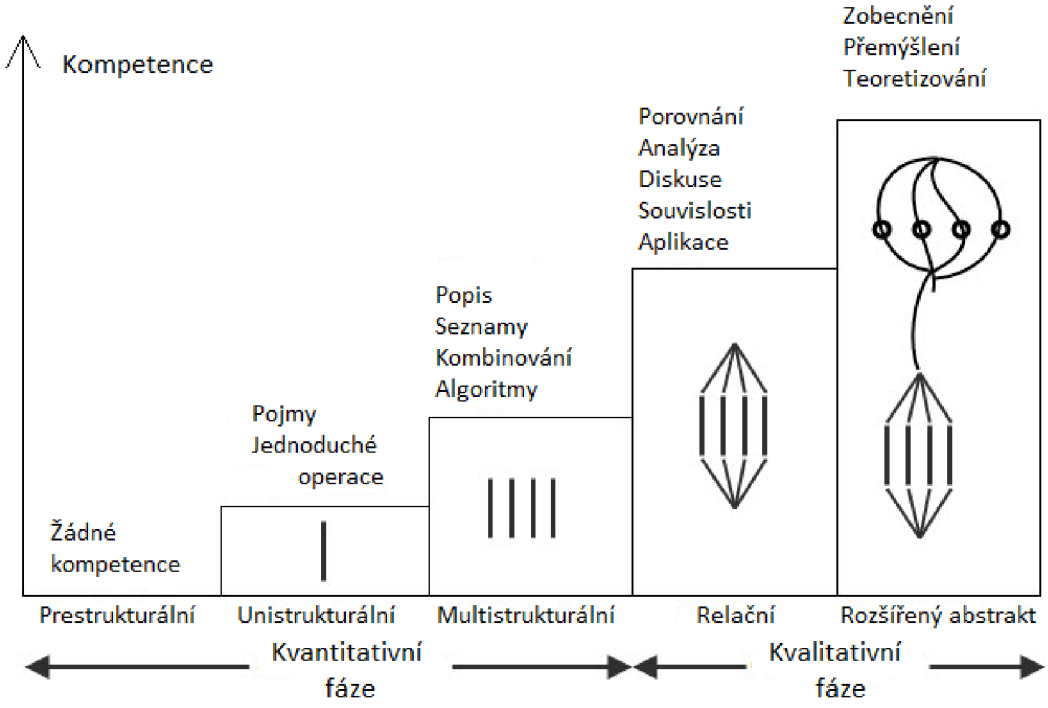
Alternativou k softwarovým metrikám je subjektivnější měření založené na vzdělávacích taxonomiích, jako jsou (revidovaná) Bloomova taxonomie (obr. 2.3; Bloom, 1956) a SOLO taxonomie (obr. 2.4; Biggs a Collis, 1982). Tento přístup byl zkoumán a popsán v literatuře s popisem klasifikace programovacích úkolů u začátečníků pomocí obou taxonomií (Whalley a kol., 2006). Výzkumníci zjistili, že vyučující jsou schopni spolehlivě klasifikovat problémy s programováním u začátečníků pomocí taxonomie SOLO (Lister a kol., 2009a, 2009b), a že kognitivní úroveň řešeného úkolu odráží skutečnou zjištěnou obtížnost.



Obrázek 2.3: Dimenze kognitivních procesů; Bloomova (vlevo) a Revidovaná Bloomova taxonomie (vpravo)

Zdroj: (Wilson, 2001)

Hodnocení softwarovými metrikami pro zjištění obtížnosti úlohy je čistě kvantitativní a vychází z pozitivistického vnímání. Přiřazení příslušné úrovně dle ta-



Obrázek 2.4: SOLO taxonomie

Zdroj: upraveno dle (Biggs a Tang, 2011; s. 91)

xonomie je subjektivnější, a tedy i kvalitativní. Korelace přiřazené úrovně s pozorovanou obtížností řešení úlohy probanda je kvantitativní. Požadovaná kognitivní úroveň probanda pro vyřešení úlohy a složitost kódu tak tvoří větší možnosti, jak změřit, popsat a pochopit obtížnost tvorby programu. V rámci výzkumu byla navržena řada programovacích úkolů tak, aby jednotlivé úlohy postupně stavěly na programovacích konceptech a byly patřičně obtížnější či složitější.

2.6 Teorie učení

Učení je aktivní proces, při kterém jedinec získává (osvojuje si, mění, přizpůsobuje, ...) něco, co je pro něho nové. Výsledkem učení jsou aktivně vytvářené osobní hypotézy a významy skutečnosti (Petty, 2013). Existuje řada teorií učení, které vysvětlují podstatu procesu učení a definují podmínky, za kterých učení probíhá. Současně přinášejí představu o tom, co je to učení, co jej umožňuje.

Základní rozlišení procesu výuky tkví v podstatě, jak si učitelé a výzkumníci představují své studenty. Studenti mohou být reprezentování prázdnou nádobou, do které jsou proudem znalosti nalévány, nebo aktivními tvůrci vlastních znalostí. V prvním případě jsou učitelé často popisováni jako „mudrci na jevišti“ a ve druhém jako „pobočníprůvodci“.

Učitelův názor na žáky závisí na jeho pohledu na znalosti. Je znalost souborem objektivních faktů o světě, které lze přenést? Pokud ano, pak jeho názor je objektivní. Pokud jsou chápány lidské znalosti jako něco dynamického, které lidé neustále, individuálně a společensky přizpůsobují a konstruují, pak je jeho pohled konstruktivistický. Odlišné pohledy na učení jsou cenné z různých důvodů a v různých kontextech. Žádná teorie není ideální pro každou situaci, ale s použitím různých perspektiv můžeme komplementárním způsobem dosáhnout toho nejlepšího ze všech metod.

2.6.1 Teorie učení a prahové koncepty

Bezprostředním kontextem, který ovlivnil vznik teorie prahových konceptů, bylo zkoumání způsobů, jak zlepšit výukové a studijní prostředí pro vysokoškoláky v době rychlého rozšiřování univerzit ve Velké Británii. Myšlenka prahových konceptů navazuje na jiné teorie učení zejména konstruktivismus, jak je uvedeno dále v této podkapitole.

Jak již bylo uvedeno, existuje řada teorií učení, řada pohledů na učení a lze je kategorizovat různými způsoby. Například Perkins (2008) má trinitární pohled na znalosti jako „přivlastňovací“, „performativní“ a „proaktivní“. To by mohlo odpovídat fázím učení, které vidí Entwistle (2008), ve kterých studenti začínají „pamatováním faktů“. Poté přecházejí k další fázi „schopnosti tyto znalosti aplikovat a využívat“ a nakonec pokročí k fázi „dát smysl myšlenkám pro sebe“.

Sfard (1998) rozděluje učení na učení jako „získávání“ versus učení jako „účast“. Jeho pohled je založen na posuzování znalostí buď jako něčeho, co se hromadí, nebo jako něco, co se dělá. Proto kategorizuje konstruktivistické pohledy na uče-

ní společně s transmisivními pohledy, protože oba zahrnují „učení jako získání vlastnictví nad nějakou komoditou“ (Sfard 1998). Alternativní kategorizace, rozdělující konstruktivismus od přenosu, vzniká, když je student považován za aktivního účastníka nebo pasivního příjemce učení. Watkins (2011) ve své studii sleduje tripartitní klasifikaci učení jako „být vyučován ... individuální vytváření smyslů ... budování znalostí jako součást jednání s ostatními“.

Transmisivní pedagogika

Transmisivní neboli předávací pojetí výuky, předpokládá, že „znalosti jsou někde„ tam venku“, vznášejí se v knihách nebo mozcích a učení je o tom, jak je dostat do hlavy“ (Watkins, 2011). Toto pojetí se někdy nazývá též transmisivně-instruktivní vyučování, jelikož učitel při výuce zároveň dává žákům instrukce a návod k řešení úloh. Nejčastěji je využívána tzv. frontální výuka, která se vyznačuje dominantním postavením učitele, který řídí a usměrňuje společnou práci žáků, jejím hlavním cílem je, aby si žáci osvojili co nejvíce poznatků. (Maňák a Švec, 2003). V této transmisivní pedagogice se předpokládá, že žáci budou v zásadě pasivní a budou věnovat pozornost výrokům autoritativního učitele, aby absorbovali fakta. Transmisivní učení je již dlouho hlavním pohledem na učení i v českém školství. Newton a kol. (2004) tvrdí, že v praxi ve třídě dominoval model přenosu, protože „pedagogika je v zásadě konzervativní činností“ (učitelé mají sklon učit způsobem, jakým byli vyučováni) a protože vnější „tlaky ... odpovědnosti a marketingu“ vzdělávání “vedlo učitele k tomu, aby se zaměřili na časově efektivní činnosti, které maximalizují skóre hodnocení. Umožňuje také nákladově efektivní model typu jedna ku mnoha. Gardner (1998) naznačuje, že dědictví „přeplněných tříd, fixního třídního nábytku a omezení v knihách a vybavení“ vedlo k pedagogice založené na „pouhém mluvení na ně“.

Existují aspekty myšlenky prahových konceptů, které odpovídají transmisivnímu pojetí. McCormick (2008) tvrdí, že myšlenky „nezávislého konceptuálního prostoru „a souboru znalostí“, které lze také nalézt v publikaci věnované teorii

prahových konceptů (Meyer a Land, 2003), charakterizují „znalost jako objekt, nezávislý na znalostech“. Dále ale má student tendenci být považován za pasivního. Felten (2016) konstatuje, že „práce vysokoškolských studentů většinou chyběla“ v práci na koncepcích prahových hodnot. „Učitel je ten, který by měl ‚pečlivě analyzovat, co je potřeba, aby se, obrazně řečeno, pomohlo studentům překročit konceptuální práh“. (Talanquer 2014). Selhání při překročení prahové hodnoty je považováno za vyhýbání se prahové hodnotě Savin-Badenem (2006, 164) nebo zapojení „mimikrů nebo nedostatku autenticity“ (Meyer a Land 2003), které odpovídají perspektivě „deficitu žáků“, která charakterizuje transmisivní pedagogiku. Dokonce i aspekt transformace prahových konceptů je často považován za něco, co se děje studentovi; žáci se někdy zdají být do značné míry pasivní, protože procházejí mezí.

Konstruktivistická pedagogika

Konstruktivistické teorie učení vycházejí z konstruktivistického vysvětlení procesu poznávání. Jedním ze zakladatelů konstruktivismu je švýcarský psycholog Jean Piaget (např. Piaget, 1999). Konstruktivismus pracuje s myšlenkou, že svět si vytvářejí lidé sami tím, že si ho aktivně konstruují ve své mysli. Základní tezi, která je východiskem konstruktivistické výuky, lze popsat asi takto: význam a smysl věcí nemůže být přenesen z jedné osoby na druhou, ale žák si sám (nebo za pomoci druhých) aktivně vytváří poznání ve své mysli. Z pohledu konstruktivismu není možné chápat učení jako mechanický přenos vědění mezi osobami (případně není efektivní). Proces učení je aktivním procesem učících se. To znamená, že se učí, jedinec pracuje s novými informacemi, spojuje si dosavadní poznání s novým na základě již vytvořených a relativně stabilních mentálních struktur. Žák si do procesu učení přináší své poznání světa (mnohdy je mylné) a toto poznání neopouští, dokud si podle ní může vysvětlit dění kolem sebe. K tomu, aby žák vytvářel nové poznání, potřebuje dostávat příležitosti k učení (Bertrand, 1998, Kalhous a kol., 2002).

„Základní požadavek konstruktivistické pedagogiky" podle Drivera, Asoko a kol. (1994) je, že „znalosti se nepřenášejí přímo z jednoho znalce na druhého, ale jsou aktivně vytvářeny žákem." U Watkinse (2011) je žák „aktivní při vytváření smyslu z prostředí".

Taylor (2006) spojuje konstrukci porozumění studenty s transformativním aspektem prahových konceptů. Tento pohled na učení ukazuje cestu studenta skrz liminality, která se „současně transformuje a je transformována studentem, když se v něm pohybuje".

Meyer a Land (2005) zkoumají problémy, se kterými se studenti setkávají, jako je zaseknutost, oscilace a mimikry, navrhují pedagogiku nejistoty, důraz je stále kladen na jednotlivce. Land a kol. (2014) hovoří o „prahovém stavu individuálního žáka". Je zde vidět, že koncept prahových hodnoty vychází z konstruktivistického modelu učení.

Sociální konstrukcionismus

Principem této teorie učení je myšlenka, kterou vyjadřuje například Watkins (2011). Tento autor zdůrazňuje, že „význam je konstruován společně v sociální aktivitě, nikoli individuálně v hlavách lidí. Lidské učení je nutně a zásadně sociální" (Watkins, 2011). Tento aspekt podpořili někteří výzkumníci zabývající se rozvíjením konceptů prahových hodnot. Například Kabo a Baillie (2010) navrhují, aby „část uchopení konceptu prahové hodnoty zahrnovala to, že student „se pohybuje ... od začínajícího myšlení k myšlení týmového odborníka". Uvádějí, že „vedle konceptuálního mistrovství, studenti musí zvládnout týmovou práci" (Kabo a Baillie, 2010).

Prahové koncepty a teorie učení

Kořeny teorie prahových konceptů navazuje na předchozí teorie učení. Kutsar a Karner (2010) píší: „perspektiva prahových konceptů ... integruje několik teoretických perspektiv kombinujících teorie učení se sociálním konstruktivismem".

To naznačuje důvod, proč byly prahové koncepty interpretovány odlišně lidmi z různých oborů, což by zase mohlo vysvětlit širokou přitažlivost prahových konceptů v celé řadě oborů od inženýrství po antropologii.

Webb (2016) tvrdí, že „teorie prahových konceptů poskytují nový teoretický rámec pro přehodnocení výzkumu a praxe ve vysokoškolském vzdělávání". Je nutno ale pracovat s prahovými koncepty velmi opatrně. Jedná se o teorii, která musí být dále dopracována o určitý filozofický základ. Zejména pojem liminálního prostoru zůstává relativně nedefinovaný.

Tyto popsané přístupy k učení a mnohé další, i když jejich koncept je třeba odlišný, mají jeden společný problém – mohou vést k přetížení studentů. Tento problém je společný snad všem učitelům a snaha vyhnout se mu, je velkým problémem. Jak předat studentovi vše důležité, ale stále tak, aby to dokázal absorbovat? Co v kurikulu ponechat, a co opomenout? Co je tou esenciální částí učiva, kterou je nutno studentovi předat za všech okolností?

2.7 Prahové koncepty

Britští výzkumníci Erik Meyer a Ray Land na počátku 21. století vedli rozsáhlý výzkum na britských univerzitách. Zjistili, že to, co studenty zejména trápí, je způsob, jak se naučit klíčové znalosti ze svého oboru. Studenti často namítali, že se i po ukončení vysokoškolského studia necítí být odborníky v daném studijním oboru.

Tyto rozhovory vedly Meyera s Landem (Mayer a Land, 2003) k pojmenování takzvaných prahových konceptů (anglicky Threshold Concepts, TC). Jak autoři uvádí v druhé ze svých stěžejních studií, „v určitých disciplínách existují ‚konceptuální brány' nebo ‚portály', které vedou k dříve nepřístupnému, a zpočátku možná ‚problematickému', způsobu myšlení o něčem." (Mayer a Land, 2003, str. 373). Překročením prahu takové „brány" student získává nový způsob porozumění a interpretování něčeho — jeho vnitřní pohled na téma, nebo dokonce na svět

jako takový je transformován. V ekonomii je takovým prahovým konceptem např. koncept nákladů obětovaných příležitosti, v biologii je jím evoluce a ve fyzice třeba gravitace. Každý obor má svůj unikátní prahový koncept (nebo koncepty), nicméně vztahují se k nim ty samé charakteristiky.

Meyer a Land (Mayer a Land, 2003) poznamenávají, že některé pojmy jsou prožívány jako určitá tajemství, která se pro nezasvěcené v dané disciplíně zdají zcela nemožné pochopit. Ve srovnání s přímočařejšími koncepty hrají tyto problematické koncepty často hlavní roli při učení předmětu. Abychom jim porozuměli, je často zapotřebí velkého úsilí a na chvíli může být student zaseknutý v přemýšlení o určitém pojmu, co vlastně znamená, než konečně najde „bránu (portál), která vede ze tmy". Pochopení těchto konceptů navždy změnilo studenta ve vidění světa novýma očima; jako kousky skládačky integrují další kousky znalostní skládačky do celku. Tento typ konceptu lze často nalézt mezi těmi studijními tématy, která tvoří hranice disciplíny, která ji odlišuje od ostatních věd.

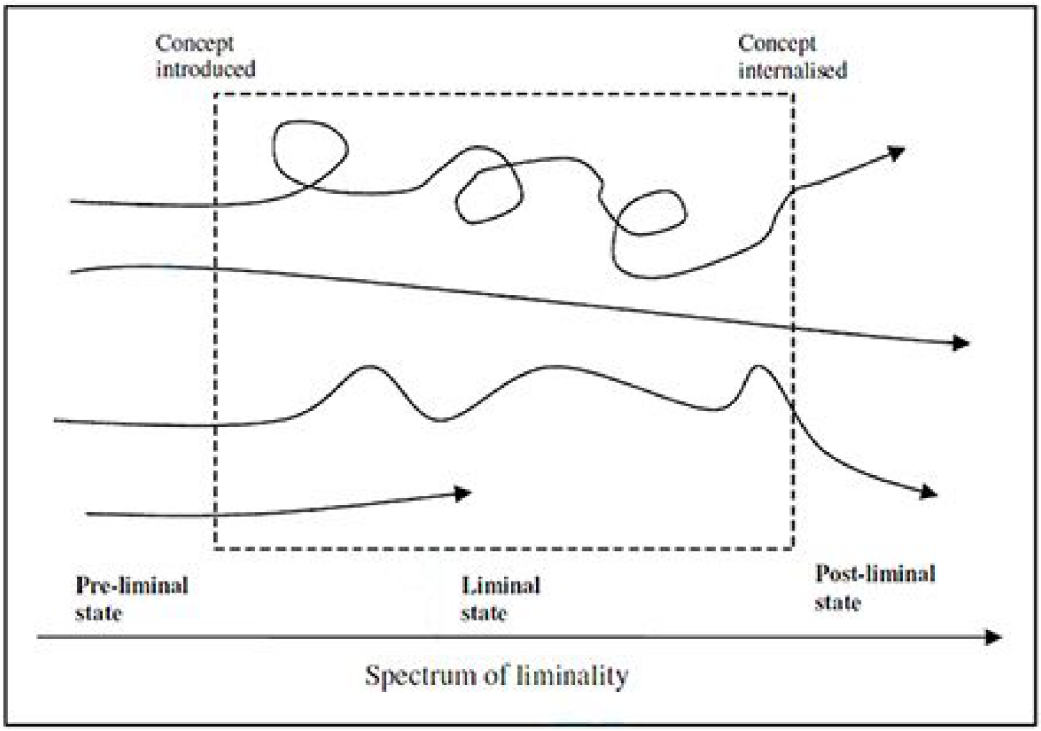
Fascinovaní těmito „portály", které jak zabraňují, tak umožňují učení, vytvořil Meyer a Land termín prahové koncepty (Mayer a Land, 2003) a začali s výzkumem tohoto tématu. Za účelem přesnějšího objasnění toho, co mají na mysli novým termínem, navrhli definici založenou na jeho rozlišovacích vlastnostech. Nové koncepty jsou: (1) transformativní, (2) pravděpodobně nezvratné, (3) integrativní, (4) pravděpodobně vázány na oborovou oblast a (5) potenciálně problematické (Mayer a Land, 2003).

Meyer a Land dále tvrdí, že je jednodušší identifikovat prahové koncepty v některých předmětech, jako je fyzika, a obtížnější v jiných, jako je historie. Tvrdí, že znalosti učitelů o prahových konceptech, které existují v dané disciplíně, hrají důležitou roli při hodnocení a navrhování efektivních učebních prostředí a jejich účelem je o tom vést debatu. Díky svým silným transformativním vlastnostem jsou prahové koncepty často obtížné se naučit. Existuje zjevné riziko, že ti, kdo těmto konceptům opravdu nerozumí, budou zadrženi ve stavu, kdy budou nuceni napodobovat a pamatovat si je nazpaměť, aniž by se s tímto konceptem sezná-

mili. Tito studenti se „zaseknou“ v tom, co Meyer a Land označují jako liminální prostor (Mayer a Land, 2005, s. 345).

Během svého studia se tedy student neučí jenom konkrétní oborové informace a koncepty. Proces učení na vysoké škole by se dal spíše charakterizovat jako osvojování si způsobů myšlení a jednání v daném oboru (Pabian, 2012). S tím může být spojený pocit ztráty, ztráty dosavadního způsobu myšlení, ztráty jisté představy o světě. Takovým stavem v našem případě vysokoškolského studia může být stav, kdy se student necítí ani jako informatik, ani jako laik. V některých publikacích se dokonce tento stav označuje, že student přišel studiem o svou „nevinnost“. Tento původně etnografický termín se tradičně pojí s transformativním stádiem, do kterého jedinec vstupuje při přechodu z jednoho stavu do jiného (obr. 2.5). Takový stav nazvali Meyer s Landem (Mayer a Land, 2003) liminální prostor (str. 375). Pro studenty může vstup do liminálního prostoru působit úzkost, nebo se v něm v určitém bodě „zasekne“. To může mít za následek předstírání „mimikry“ nebo až rezignaci a zanechání studia. Pro takový falešný stav, kdy student porozumění prahovému konceptu jenom předstírá nebo s ním „zápasí“, se užívá termínu mimikry (str. 383). Student užívající strategii mimikry se někdy brání hlubšímu porozumění, jindy se pokouší o aspoň omezené porozumění.

Land a kol. (2012) povzbuzují pedagogy v různých předmětech, aby na své předměty reagovali pomocí konceptů prahových hodnot jako analytického nástroje k lepšímu porozumění učení ve svém předmětu, a tím ke zlepšení vzdělávání. Prahové koncepty lze použít k definování potenciálně silných transformativních bodů ve výuce. Mohou sloužit k identifikaci bodů pro učitele, kteří tím studentům poskytnou příležitosti k získání důležitých konceptuálních znalostí. To znamená, že existuje potenciál zjistit rozhovorem se studenty, zda byla liminální hodnota překročena, a studovat, o čem studenti mluví a jak používají jazyk. Prahové koncepty jsou podmnožinou základních konceptů v dané disciplíně, kde základní koncepty jsou stavební kameny příslušné disciplíny, které je nutné pochopit. Prahový koncept umožňuje studentům pochopit a integrovat mnoho dří-



Obrázek 2.5: Liminální prostor

Zdroj: (Harlow a kol., 2011)

ve nesouvisejících myšlenek v určité disciplíně a také zlepšit komunikaci jednotlivců s užitím konkrétní terminologie.

Vlastnosti prahového konceptu podle Meyer a Land (2003):

transformativní — jakmile mu student porozumí, mění způsob, jakým se na disciplínu dívá; nový způsob pohledu umožňující studentovi pokročit na úroveň potřebnou pro pokročilé pochopení dané disciplíny;

problematický — zahrnuje soubor znalostí, které mohou být konceptuálně obtížné, kontraintuitivní (narušující běžné intuitivní ontologické představy o diciplíně) nebo naprosto cizí či zdánlivě nekoherentní.;

nezvratný — jelikož má transformativní potenciál, je těžké se jej „odučit“ (co nicméně nevylučuje pozdější zamítnutí nebo modifikování konceptu po rafinovanějším porozumění);

integrativní — pomáhá studentovi propojit různé aspekty z oboru, které předtím považoval za nesouvisející, nebo mu byly neznámé;

ohraničený — vymezuje určitý konceptuální prostor sloužící k specifickému a limitovanému účelu; označují hranice pojmové oblasti nebo disciplíny samotné;

diskurzivní — provádí ho změna v užívání jazyka, jeho rozšíření a vylepšení;

rekonstituční — může způsobit změnu v subjektivitě studenta, která může být zpočátku rozeznána zejména lidmi v jeho okolí;

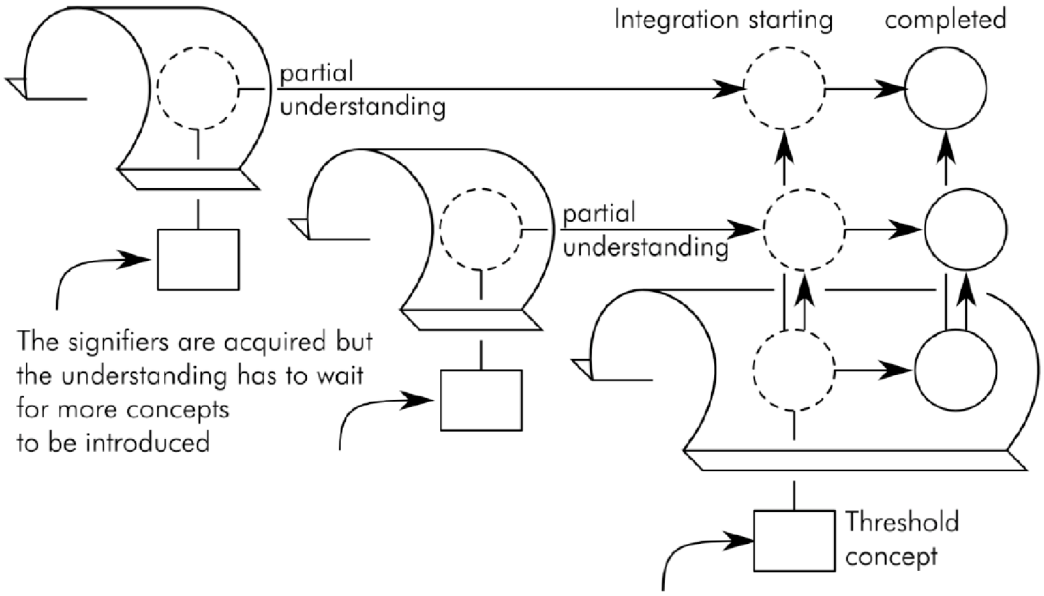
liminální — je svým způsobem „přechodovým rituálem", který není jen jednoduchým přechodem od lehkého k těžkému, ale často znamená spíše chaotickou spletitou cestu.

Autoři Land a kol. (2014) uvádějí, že nové pojmy je třeba získávat postupně. Významné koncepty ale nejsou navrženy zpravidla tak, aby byly chápány postupně, a svět funguje jako integrovaný celek. Určité množství holistického porozumění je vždy nutné. To nevyhnutelně znamená, že v učebních osnovách existuje (mezní) období nejistoty, ve kterém nejsou pochopeny důvody pro zavedení konceptu a možná koncept samotný. Jak poznamenal Dewey (1933):

„Reflektivní myšlení je vždy více či méně problematické, protože zahrnuje překonání setrvačnosti, která vede k přijímání návrhů v jejich nominální hodnotě; zahrnuje ochotu snášet stav duševního neklidu a narušení. Reflektivní myšlení ve zkratce znamená pozastavení přemýšlení během dalšího zkoumání; a napětí bude pravděpodobně poněkud bolestivé" (Dewey, 1933).

Scheja (2006) k tomu dále uvádí, že pedagogicky výzvou pro učitele je minimalizovat tato období a pro integraci těchto konceptů v pravidelných intervalech je zapotřebí konceptů prahových hodnot, jinak kurz zůstane konceptuálně fragmentovaný (obr. 2.6).

Pomáhá, pokud si studenti uvědomují své neúplné porozumění a chápou povahu svých myšlenkových procesů. To jim umožňuje být trpěliví, vytrvalí a neopouštět liminální prostor příliš brzy (Land a kol., 2014). Přesto existuje přirozená tendence studentů uspokojit se s konkrétními porozuměními dříve, než je odhalen konceptuální celek, což znamená, že by měli být připraveni později přehod-



Obrázek 2.6: Období konceptuální nejistoty a opožděného porozumění
Zdroj: (Land a kol., 2014)

notit své koncepce.

Určiým nedostatkem teorie prahových konceptů je, že v rámci této teorie chybí konkrétní metodologie, která by se dala využít při identifikaci prahových konceptů nebo při přiřazování příslušných charakteristik. Autoři Timmermans a Meyer ve své publikaci (Timmermans a Meyer, 2017) naznačili určité metody a směry pro určení prahových konceptů, ale tyto navržené metody zůstávají na poměrně obecné úrovni. Proto jako základní metodu pro zkoumání prahových konceptů jsme zvolili fenomenografický přístup, který se nám jevil jako přirozený způsob jak dané jevy zkoumat.

Přístup teorie prahových konceptů se stal v posledních letech důležitým nástrojem pro zkoumání přístupů učení studentů a navrhování a zlepšování osnov předmětů v různých oblastech. V našem výzkumu se proto tento výzkumný rámec stal základním pří zjišťování problémů a procesu výuky v programování se zaměřením na objektový přístup.

2.8 Fenomenografie

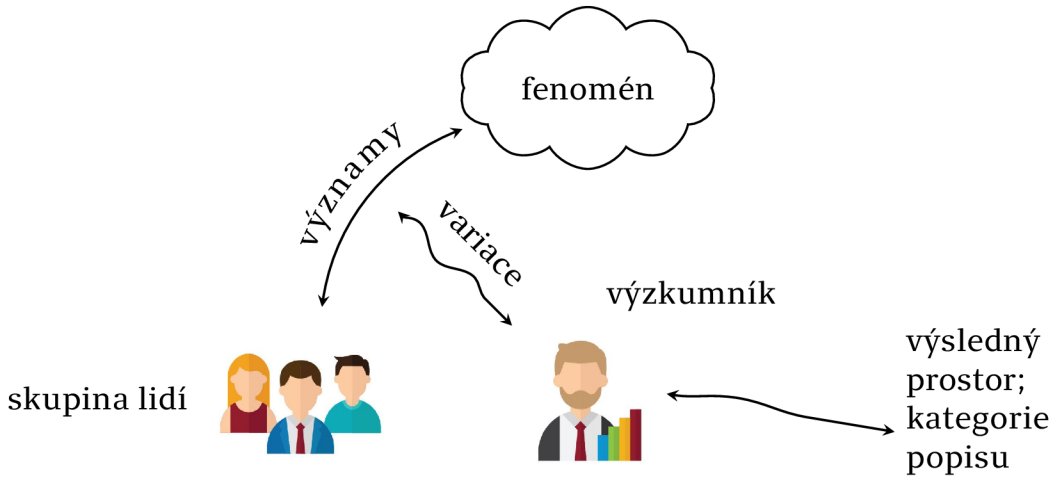
Je dalším teoretickým rámcem využívaným v této dizertační práci. Zakladatelem tohoto směru pedagogického výzkumu, jehož cílem je pochopení kvalitativně rozdílného prožívání zkušenosti a který zodpovídá otázky týkající se myšlení a učení, je Marton (1981), který tento přístup nazval fenomenografie. Postupně se vyvinul a dospěl k výzkumné tradici, která se týká toho, jak se u různých skupin lidí objevují různé aspekty světa. Studie v rámci tohoto přístupu jsou v zásadě explorativní a využívají empirická data a všechny berou perspektivu druhého řádu na některé jevy. To znamená, že fenomenograf nezkoumá jevy jako to, čím jsou (perspektiva prvního řádu), ale variace toho, jak jsou lidmi prožívány a vyjádřeny (perspektiva druhého řádu). Marton, jeden z průkopníků fenomenografie, definuje tuto výzkumnou specializaci takto (Marton, 1981):

„Fenomenografie je výzkumná metoda přizpůsobená pro mapování kvalitativně odlišných způsobů, jakými lidé zažívají, pojímají, vnímají a chápou různé aspekty a jevy ve světě kolem sebe. V důsledku toho je předmětem studia vztah mezi určitým fenoménem a skupinou lidí a variace vztahu. Není to ani fenomén, ani lidé, které se snaží vysvětlit; je to zkušenost skupiny s tímto fenoménem“ (Marton, 1981).

Fenomenografie bývá označována jako: přístup k výzkumu, metoda, orientace, paradigma, perspektiva či výzkumný program (Tight, 2015). Marton a Booth (1997) nepovažují fenomenografii za metodu samu o sobě, jde podle nich o celkový přístup se silným pedagogicko-výzkumným zájmem. Základní princip fenomenografie lze vidět na obrázku 2.7.

Ontologie fenomenografie je nedualistická, což znamená, že neodděluje pozorovatele od pozorovaného (objektu a subjektu). Marton (1986) to vysvětluje takto:

„Existuje pouze jeden svět, skutečně existující svět, který je lidskými bytostmi prožíván a chápán různými způsoby. Je to současně objektivní a subjektivní. Zážitek je vztah mezi objektem a subjektem, který zahrnuje obojí. Zkušenost je



Obrázek 2.7: Základní princip fenomenografie

Předmětem studia fenomenografického výzkumníka je variace ve vztahu mezi určitým fenoménem a skupinou lidí. Primárním výsledkem analýzy je omezená sada kvalitativně odlišných kategorií popisu. Kategorie a vztahy mezi nimi tvoří výsledný prostor.

Zdroj: upraveno dle (Boustedt, 2008); ikony (EEZY, 2021)

stejně tak aspektem předmětu, jako i předmětu" (Marton, 1986).

Zkušenosti z dřívějších studií ukázaly, že různí lidé popsali jevy jen několika různými způsoby, což vedlo k základnímu epistemologickému předpokladu, totiž že existuje pouze omezený soubor kvalitativně odlišných způsobů, jak zažít a popsat jev. Každý kvalitativně odlišný způsob prožitku tvoří kategorii popisu. Kromě toho mezi kategoriemi vždy existuje sada logických vztahů a logická struktura v kombinaci s kategoriemi popisu tvoří výstupní prostor. Tímto způsobem výsledný prostor obsahuje bohatou sadu informací o tom, jak je fenomén prožíván a jak tyto zkušenosti navzájem souvisejí.

Staudková (2016) uvádí, že lze identifikovat tři linie tematického vývoje fenomenografického výzkumu. První se soustředí na obecné aspekty učení, především na kvalitativní rozdíly v přístupu k učení a jeho výstupům. Druhá linie výzkumu se zaměřuje na učení a pojetí učení studentů uvnitř kontextu dané disciplíny. Třetí rovina se týká výzkumů každodennosti a individuálních představ ohledně různých aspektů života. Hlavní tematickou linií fenomenografického výzkumu je zkušenost učení. Fenomenografický výzkum se nesoustředí na učení

samo o sobě, ale sleduje vztah mezi jedincem a učením a umožňuje zachytit rozdílné způsoby prožívání tohoto vztahu. Zkušenost učení zahrnuje studentův přístup k učení a pojetí učení.

Učení tedy není determinováno pouze osobnostními předpoklady, ale závisí především na tom, jak studující k učení přistupují. Pojetí učení zahrnuje osobní názor na učení samotné, jímž se student řídí. Zahrnuje otázky proč, kdy, kde, jak a z čeho se učit, jaké prožitky by měly učení provázet, podle čeho člověk pozná, že je naučen, jaký výsledek je přijatelný a jakou roli hraje prostředí učícího se (Mareš, 1998).

Kromě toho neexistuje žádné výslovné spojení se zkušenostmi jakékoli jednotlivé osoby ve výsledném prostoru. Každá kategorie popisuje konkrétní způsob, jak zažít určitý fenomén, pozorovaný v kolektivu, a je tak tvořena sloučenými významovými fragmenty, které se nacházejí v popisu fenoménu jednotlivce. Kolektivní vyhlídka je kvalita, která odlišuje fenomenografii od kvalitativního výzkumu obecně, což se často popisuje jako perspektiva jednotlivce (Marton, 1986).

Fenomenografie se nesoustředí na individuální charakteristiky učícího se ani na motivaci (typické pro psychologické přístupy k učení) ani na rod, etnicitu nebo třídu (typické pro sociálně-vědní přístupy ke studiu vysokého školství). Namísto toho sleduje, jak studující rozumí tomu, co vlastně znamená učit se a naučit se (Pabian, 2012).

2.8.1 Základní koncepty fenomenografie

Fenomenografie je o různých způsobech porozumění. Obecný způsob porozumění představuje vztah mezi subjektem a jevem. Je to výsledek úmyslného přemýšlení člověka, interagující s jevem a snažící se vytvořit smysl. Tento jev lze teoreticky vnímat nekonečně mnoha způsoby, ale v procesu vytvoření smyslu, zůstane pouze omezený počet (obvykle 2 až 6) způsobů porozumění (Uljens, 1996). Toto bylo potvrzeno v četných empirických studiích (Marton, 1986). Různé způsoby

porozumění mají oba aspekty – „co“ i „jak“. Aspekt „co“ nám říká, co je v centru pozornosti subjektu, aspekt „jak“ nám popisuje, jak se vytváří význam.

Metody fenomenografického výzkumu sběru a analýzy dat lze použít ke studiu řady problémů, včetně přístupů k učení, přístupů k výuce, pochopení vědeckých jevů naučených ve škole nebo pochopení obecných problémů ve společnosti nesouvisejících se vzdělávacími systémy.

2.8.2 Proces výzkumu v rámci fenomenografie

Bowden (2000) nastiňuje fenomenografický proces výzkumu, který má čtyři fáze: plán, sběr dat, analýza a interpretace. Ve všech těchto fázích se musí výzkumník soustředit na účel studie. To je důležité vzít v úvahu pro získání důvěryhodných výsledků. Stejně jako ve všech výzkumech začíná plánem, který definuje účel a strategie. Samozřejmě to, co řídí výzkum, je základní otázka, na kterou se výzkumná činnost snaží odpovědět. Problémy studentů při zvládání fyziky dávaly Bowdenovi (2000) dobrý důvod, aby se pokusil pochopit pochopení důležitých pojmů ve fyzice ze strany studentů. Vstupní data pro fenomenografické vyšetřování jsou v zásadě výroky lidí o zkušenostech s jevem. Převládající metodou shromažďování těchto údajů jsou polostrukturované rozhovory s lidmi a výzkumník musí pečlivě vybírat osoby a zvažovat, proč jsou dobrou volbou. V polostrukturovaných rozhovorech tazatel klade otevřené otázky, které se týkají problémové oblasti, nebo žádá subjekty, aby hovořily o tom, co pro ně fenomén X znamená. V těchto rozhovorech jsou probandi povzbuzováni, aby mluvili o svých zkušenostech, sdělovali konkrétní příklady a vyhnuli se povrchním popisům toho, jak by věci měly být nebo jak by měly fungovat. Hloubkové rozhovory jsou nahrávány na pásku a doslovně přepisovány. Zkušenosti z velkého počtu fenomenografických studií ukazují, že údaje od dvaceti probandů jsou obvykle dostatečné, aby se objevily všechny různé způsoby pochopení daného fenoménu (Holmstrom a kol., 2003, Sandberg, 1994).

Kategorie popisu jsou abstrakcemi výzkumníka, jak byly ve výzkumu identi-

fikovány různé způsoby porozumění. Odkazují se na hromadnou úroveň a popisují různé způsoby, jak lze jev pochopit. Kategorie jsou často založeny na způsobech porozumění vyjádřených z více rozhovorů jednotlivých probandů. Tyto kategorie jsou vzájemně se vylučující. Všechny kategorie popisu, výsledný prostor, tvoří výsledek fenomenografické studie. V mnoha fenomenografických studiích, zejména staršího data, je prezentovan pouze soubor kategorií jako výsledek studie (Sjöstrom a Dahlgren, 2002). Avšak různé kategorie ve výsledném prostoru jsou obvykle vzájemně hierarchicky propojeny (Marton a Booth, 1997; Sandberg, 1994) a definování tohoto strukturálního vztahu mezi kategoriemi může být dalším krokem fenomenografické analýzy. Tato hierarchická struktura výsledného prostoru může být odvozena z dat nebo může být výsledkem teoretické analýzy kategorií.

Kategorie by měly mít vzájemné logické vztahy. Pokud ne, měl by výzkumník údaje znovu zvážit. Nakonec by výsledky měly být interpretovány podle účelu studie. V aplikované vývojové fenomenografii je interpretace přirozeným důsledkem nastolené výzkumné otázky. Pokud výsledek prozradí, jak žáci zažívají jevy ve vzdělávacím kontextu, mohou učitelé výsledky použít k ovlivnění jejich pedagogiky a výuky. Mohou přizpůsobit způsob, jakým představují nové koncepty, nebo lépe porozumět tomu, proč studenti nedělají určité úkoly.

Naproti tomu čistá fenomenografická studie by mohla mít pouze účel popsat zkušenost s fenoménem bez dalších implikací. Ve všech případech musí být výsledky studie vnímány ve světle jejího účelu, a pokud chce výzkumný pracovník použít výsledky v jiném kontextu, je třeba vzít v úvahu tuto otázku.

Při identifikaci prahových konceptů v určité disciplíně se musí výzkumník vypořádat s řadou problémů. Obvykle se používají kvalitativní metody (např. rozhovory); konkrétní metodika však dosud nebyla stanovena jako nejvhodnější metodika pro identifikaci prahových konceptů. Nejběžněji používanou metodou jsou rozhovory se studenty, při nichž se výzkumníci ptají na obtížné a náročné části osnov a na změny, ke kterým mohlo dojít v důsledku překonání těchto překážek.

Zatímco tento přístup je vhodný pro cíle takového výzkumu, je-li použit samostatně, může to vést ke kontroverzním výsledkům, protože každý účastník může uvažovat o jiném konceptu, a proto hloubka, kterou bude tento koncept zkoumat, musí být dostatečně propracovaná (Barradell, 2013).

3 Cíl disertační práce

V předchozí kapitole jsme předložili teoretická východiska výzkumného problému a z něj vycházející výzkumné otázky RQ1–RQ3 definované v kapitole 1.4.

Výzkumný problém lze stručně popsat následujícím způsobem:

V učení se objektově orintovanému programování je důležité pochopení hlavních konceptů objektově orientovaného přístupu začínajícími studenty. Bez tohoto pochopení studenti nebudou schopni úspěšně programovat zejména komplexnější aplikace. Pro učitele je proto důležité, aby pochopili, jak studenti rozumí konceptům objektově orientovaného přístupu, které koncepty jim dělají problémy (problematické koncepty), a které koncepty je významně posunou v jejich programátorských dovednostech (transformativní koncepty).

Výzkumný problém se týká zjištění, proč je tak obtížné naučit se programování a zvláště objektově orientovanému přístupu v programování. Cílem naší práce je tak identifikovat různé koncepty v dané oblasti, které studenti vnímají jako zvlášť výrazné, například tím, že jsou obtížné nebo obohacující. Cílem je také prozkoumat, jak studenti vnímají koncepty, zda vidí souvislosti mezi nimi. Na pomoc jsme si vzali teorii prahových konceptů Meyera a Landa (2005) s cílem prozkoumat prahové koncepty v oblasti objektově orientovaného programování, zda takové koncepty existují, jaké to konkrétně jsou a jak ovlivňují učení studentů.

S ohledem na pochopení tohoto problému jsme si stanovili základní cíl práce:

Porozumět, jak studenti chápají základní koncepty při procesu učení se objektově orientovanému přísupu v programování, jaké jsou pro ně problematické a transformativní koncepty objektově orientovaného přístupu, jak chápou souvislosti mezi nimi, a jaké strategie používají při jejich zvládání.

Tento výzkumný cíl odpovídá výzkumným otázkám RQ1 až RQ3, které jsme si stanovili a fomulovali v předchozí kapitole. Na základě těchto otázek můžeme základní cíl rozdělit do třech dílčích cílů:

1. Analyzovat, jaké koncepty objektově orientovaného přístupu k programování jsou pro studenty obtížné a zejména transformativní dle teorie prahových konceptů.

2. Identifikovat, jaké strategie používají studenti ve výuce programování při překonávání těžkostí u problematických konceptů.

3. Porozumět jak studenti při výuce programování chápou základní koncepty objektového programování a vztahy mezi nimi.

Studenti učící se objektově orientované programování hovoří o problémech souvisejících s výukou programování. Programové úkoly a řešení problémů jsou přirozenou součástí tradice výuky informatiky a jsou důležité pro porozumění příslušným konceptům. Naším záměrem je porozumění problémů, kdy se studenti během řešení různých úloh dostanou do slepé uličky, a zejména snaha o pochopení, jak lze tyto problémy identifikovat, a také jak je možno pomoci studentům zvládnout některé obtížné, ale klíčové koncepty na cestě naučení se dobře programovat. Použití teorie prahových konceptů a fenomenografického přístupu je pak přirozeným nástrojem při provádění našem výzkumu.

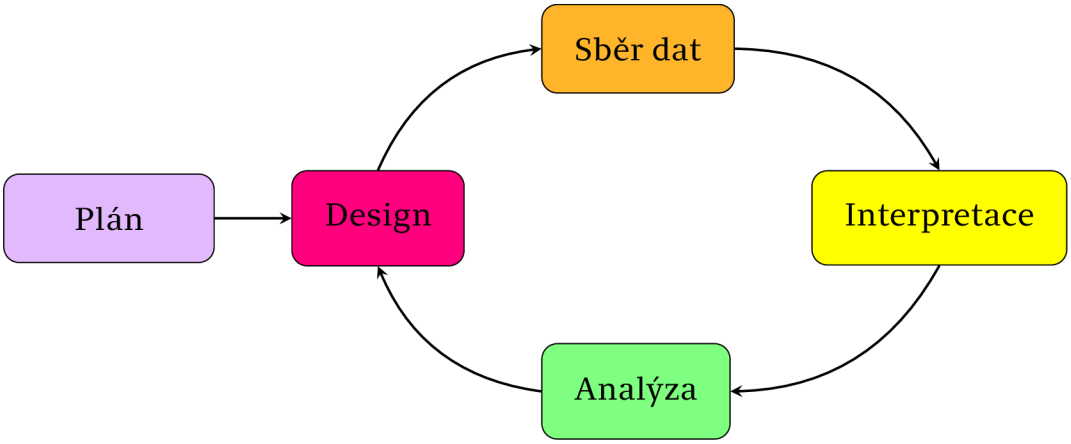
4 Metodologie práce

V mnoha oblastech v pedagogickém výzkumu je možné založit zkoumání na sběru a analýze kvantitativních dat, obvykle za použití statistických metod. Naproti tomu, Moström (2011) ve své dizertační práci uvádí, že výzkum v oblasti výuky programování je poněkud odlišný i od mnoha jiných oblastí výzkumu v informatice. Uvádí, že největším problémem ve výzkumu v oblasti výuky programování je, že statistická analýza sice může naznačovat, že existuje řada problémů, a řekne nám, jak časté jsou. Ale v oblasti výzkumu výuky programování nás spíše zajímá, kde mají studenti potíže, co je způsobuje, jakým způsobem se přes tyto chyby přenášejí, jak se s nimi vypořádají a jak jim v tom můžeme pomoct. Chceme pochopit, jak studenti přemýšlí při řešení problémů a proč dělají chyby, které dělají, nejen to, že dělají chyby, což bychom získali při omezení se na kvantitativní přístup. Je méně zajímavé vědět, kolik studentů zažívá určité problémy, ve srovnání s vědomím, jaké problémy mají a proč. Je samozřejmé, že je velmi obtížné objektivně měřit, jak někdo přemýšlí. Kvalitativní výzkum má za cíl shromáždit důkladné porozumění lidskému chování a důvodům tohoto chování. Kvalitativní metody jsou navrženy tak, aby vycházely z toho, proč a jak, na rozdíl od toho, co, kde a kdy. V této práci jsme zvolili jako primární výzkumnou metodu teorie prahových konceptů (Meyer a Land, 2003; Meyer a Land, 2005)

Jednotlivé fáze a aktivity prováděné během výzkumu názorně prezentuje diagram na obr. 4.1.

4.1 Plán studie

Na obr. 4.1 jsme si představili vizuální model našeho výzkumu. Nejprve jsme si definovali výzkumný problém a na něj navazující výzkumné otázky. Stanovili jsme si cíle práce. Dále následovalo studium literatury, kde jsme získali přehled o stavu poznání v dané oblasti a o možných metodách výzkumu dané oblasti. Na



Obrázek 4.1: Vizuální návrh průběhu výzkumu

Zdroj: autor

základě studia literatury jsme pak zvolili vhodné metody a postupy ke zkoumání daného jevu tak, abychom získali potřebná data a provedli analýzy a hodnocení těchto dat s cílem porozumět danému jevu a mohli udělat závěry týkající se zkoumané oblasti.

Pro naše účely jsme zvolili kvalitativní výzkum. Důvodem byla snaha porozumět, jak studenti přemýšlí při řešení problémů a proč dělají chyby a jaké strategie používají při překonávání obtíží v učení. Švaříček a Šeďová (2014) uvádějí, že v kvalitativním výzkumu se využívají v podstatě tři typy dat: data z rozhovorů, pozorování a dokumentů. Uvádějí, že se jedná o výzkum, kdy se pracuje se slovy a dokonce s vyjádřeními pocitů a myšlenek. Ve své publikaci (Švaříček a Šeďová, 2014) píší:

„Data kvalitativního výzkumu musíme analyzovat a interpretovat jinými postupy, než využívá kvantitativní přístup, a díky tomu získáme zcela jiné typy závěrů. “ (Švaříček a Šeďová, 2014)

4.1.1 Důvod pro výběr zvolené metodiky

V naší dizertační práci používáme kombinaci dvou teoretických rámců, a to fenomenografického přístupu a dále teorie prahových konceptů. Vycházíme z toho, že základní metodou sběru dat ve fenomenografickém výzkumu je hloubkový rozho-

vor, jak již bylo uvedeno. Takový rozhovor tematizuje aspekty zkušenosti a prožívání učení, přičemž se snaží o vyvolání hlubokých myšlenek respondenta (Yates a kol., 2012). To je i první fáze našeho výzkumu. Další krokem fenomenografického výzkumu je analýza, která může být provedena různými způsoby. Výstupem fenomenografie jsou kategorie popisu, které zachycují jednotlivé charakteristiky zkoumaného fenoménu. Každá kategorie by měla být v jasném vztahu k danému fenoménu, zároveň musí jednotlivé kategorie stát v logickém vztahu k sobě navzájem (Marton, 1981). V našem případě se jedná o zachycení jednotlivých prahových konceptů a zaměření se na významové složky těchto konceptů.

4.2 Design výzkumu

4.2.1 Metodika výzkumu konceptů prahových hodnot

V současné době neexistuje žádná dobře definovaná metoda, jak by bylo možné identifikovat prahové koncepty. Ve své práci se tímto problémem zabývá Cousin (2009). Identifikuje metody, které byly použity v různých výzkumech, a uvádí, že výzkum konceptů prahových hodnot je „dotazem na transakční kurikulum“ (Cousin, 2008), který zpravidla zahrnuje lektory, studenty a pedagogy. Davies (2006) naznačuje, že identifikace prahových konceptů vyžaduje metodu, která je „charakteristická a nezbytná vzhledem k charakteristikám prahových konceptů“. V původní publikaci Meyer a Land's (2003) demonstrují některé příklady prahových konceptů. Mnoho z těchto prahových konceptů bylo identifikováno tak, že autoři poskytli charakteristiky prahových konceptů výzkumníkům v oblasti pedagogiky a požádali tyto výzkumníky, aby pomocí těchto charakteristik tyto prahové koncepty identifikovali. Tímto způsobem získali příklady do zmíněné publikace. Tuto metodu používali i jiní autoři (Cousin, 2008; Davies a Mangan 2005). Tato metoda identifikace prahových konceptů však vytváří problémy, protože výzkumníci pedagogové mohou identifikovat, co oni považují za důležité koncepty v rámci své disciplíny (Davies 2008; Davies a Mangan 2005), a v některých příkla-

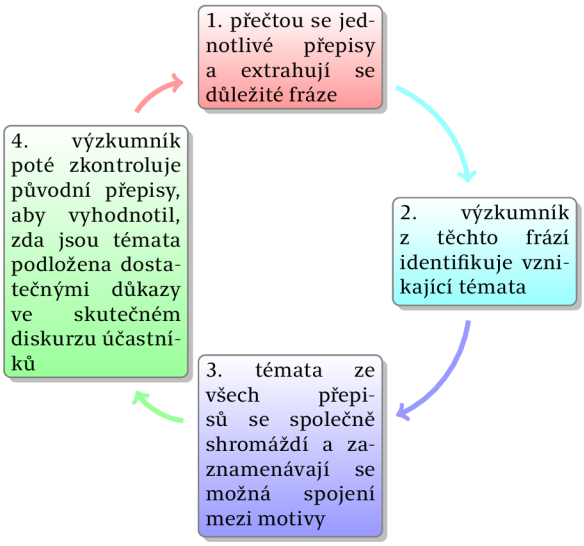
dech se jejich názory mohou lišit. Na druhé straně Eckerdal a kol. (2006) uvádějí, že přes tyto problémy může být takové uvedení základních konceptů „dobrým výchozím bodem“ pro další navazující výzkum. Lze konstatovat, že pro výzkum prahových konceptů bylo v literatuře použito mnoho metod sběru dat. Často však někteří výzkumníci vědci identifikovali prahové koncepty na základě svého osobního názoru (Quinnell a Thompson 2010). Další způsob bylo provedení rozhovorů a dotazníků se studenty, které byly dále použity k identifikaci potenciálních konceptů prahových hodnot v rámci různých oborů, např. Kabo a Baillie (2010). Jiní použili příklady prací studentů k identifikaci potenciálních konceptů prahů, např. Shanahan a kol. (2010), Taylor and Meyer (2010). Někteří autoři kromě hodnocení studentů a zpětné vazby (Quinnell a Thompson 2010) použili i předchozí výzkum a výsledky z literatury, např. Kutsar a Kärner (2010). Souhrnně lze říci, že v předchozím výzkumu prahových konceptů byla podrobně popsána celá řada různých dat, metod sběru dat a účastníků. Výzkum však trpí určitou nejednotností v metodologickém přístupu a procesu analýzy.

4.2.2 Fenomenologický analytický přístup

Pro náš výzkum jsme zvažovali několik různých metodologických a analytických přístupů. Nakonec jsme se rozhodli použít metodiku nazývanou interpretační fenomenologická analýza (Interpretive Phenomenological Analysis, IPA). Jedná se o přístup k psychologickému kvalitativnímu výzkumu s idiografickým zaměřením, klade si za cíl poskytnout vhled do toho, jak daná osoba v daném kontextu dává smysl danému jevu. Jedná se o standardní metodiku pro analýzu dat s postupy popsanými v různé literatuře.

Tento přístup byl původně vyvinut v oblasti psychologie zdraví a většina výzkumů využívajících interpretační fenomenologickou analýzu nebo přístup vycházející z interpretační fenomenologickou analýzu je v této oblasti. Jeho použití v jiných oblastech se postupně rozšířilo (Larkin a kol. 2006), a nyní se používá v jiných oblastech psychologie, stejně jako v humanitních, zdravotnických a so-

ciálních vědách (Smith a kol. 2009). Používá se také v rámci vysokoškolského výzkumu, viz například Roberts (2010), Walker a kol. (2008) a Kingston (2008). Ontologie interpretační fenomenologické analýzy je „široce realistická“ (Reid a kol. 2005) v tom, že „nemůžeme přímo poznávat svět, místo toho jej známe prostřednictvím našich interpretací založených na teorii, zkušenostech nebo myšlenkách“ (Ashwin 2009). Kromě celkového metodického přístupu má interpretační fenomenologická analýza také související přístup k analýze dat. Tento proces souvisí s filozofií interpretační fenomenologické analýza v tom, že je idiografická, se zaměřením na konkrétní, než přejde k obecnému, přičemž analýza je „důkladná a systematická“ (Smith a kol. 2009), viz obrázek 4.2, znázorňující procesy interpretační fenomenologické analýzy.



Obrázek 4.2: Procesy zahrnuté do interpretační fenomenologické analýzy
Výzkumník projde tolika iteracemi tohoto cyklu, kolik považuje za nutné k adekvátnímu zachycení témat a podtémat, která vycházejí z diskurzu účastníků.

Zdroj: autor

Interpretační fenomenologická analýza je ve svém přístupu interpretační, zkoumá výskyt fenoménu a klade důraz na pozici výzkumníka v „usnadňování a dodávání smyslu“ (Smith a kol. 2009). „Realita“ je konstruována výzkumníkem na základě interpretace dat, což jsou interpretace určitého jevu účastníky výzkumu (Patton, 2014). Tento aspekt je v rámci interpretační fenomenologické analýzy

uznáván a posouvá analýzu od popisu k interpretaci, přičemž říká, že účastníci popisují své zkušenosti a výzkumník tyto zkušenosti interpretuje (Reid a kol. 2005).

Velikost vzorku používaného v interpretační fenomenologické analýzy se pohybuje od jednoho účastníka po více než 30 účastníků (Brocki a Wearden, 2006). Je třeba poznamenat, že neexistuje žádný „správný" počet účastníků (Smith a kol., 2009). Pomocí různých pohledů může zkoumání určitého jevu se skupinami s různými počty účastníků umožnit „podrobnější a mnohostranný výzkum"(Smith a kol. 2009). Zároveň je možný tímto způsobem uskutečnit určitý typ triangulace, jak je doporučeno pro zajištění objektivity výzkumu (Švaříček, Šeďová, a kol., 2014).

To vše je důvodem, proč se domníváme, že pro výzkum v rámci této dizertační práci, který je zaměřený na identifikaci prahových konceptů, je metodika interpretační fenomenologické analýzy velmi vhodná. Výzkum v oblasti prahových konceptů směřuje od jednoduchého popisu zkušenosti k porozumění fenoménu. To je důležité, protože budeme implicitně i explicitně vytvářet a interpretovat významy zkušeností účastníků výzkumu (studentů) s jednotlivými koncepty objektově orientovaného programování včetně kontextu. Proto je vhodné zvolit přístup interpretační fenomenologické analýzy, v němž je výrazná role výzkumných pracovníků a jejich úvahy jsou vyjádřeny jako explicitní a legitimní součásti výzkumu. Idiografická povaha interpretační fenomenologické analýzy je také vhodná pro zkoumání pojmů prahové hodnoty, protože umožňuje výzkumníkovi soustředit se nejprve na sledování toho, co je prožíváno jako obtížné u jednotlivých účastníků než se přejde k analýze mezi zkoumanými případy, která umožní zohlednit více skutečností.

4.2.3 Analýza v rámci interpretační fenomenologické analýzy

Interpretační fenomenologická analýza nepředepisuje pro analýzu dat jedinou metodu pro práci s daty, což umožňuje flexibilitu při provádění analýzy (Patton, 2014). Analýza se soustředí na obecnost případné použité analytické metody

a věnuje pozornost interpretaci např. výsledků rozhovorů a zkušeností s účastníky výzkumu. V důsledku toho se vyvinula řada společných metodických procesů a principů (Smith a kol. 2009), které se pohybují mezi konkrétním a sdíleným, deskriptivním a interpretativním, včetně přísného zvažování pohledu účastníka výzkumu. Smith a kol. (2009) poskytují podrobný přístup k analýze, ale vysvětlují, že je to jen jedna cesta. Navrhují zpočátku psát poznámky ke všemu, co se jeví jako důležité, a poté přejít k vývoji naléhavých témat, snížit objem při zachování složitosti. Poté jsou prozkoumány souvislosti mezi nově vznikajícími tématy, které nakonec vyústí v rámec těchto témat. Výsledek má představovat společné rysy a zároveň má zachytit i určité rozdíly. Během analýzy si interpretační fenomenologická analýza klade za cíl „umožnit důkladné prozkoumání idiografických subjektivních zkušeností“ (Biggerstaff a Thompson, 2008), čímž se zaměří na jednotlivce a zároveň prozkoumá „fenomén sdílený specifickou skupinou“ umožňující pohledy na obecná témata, která je třeba zvážit (Smith a kol. 2009). Je třeba ale zdůraznit, že interpretační fenomenologická analýza je induktivní přístup, jehož analýza je založena na datech, spíše než „zavedení předem stanovené teorie“ (Clarke 2009). Analýza by měla být interpretační, transparentní a věrohodná a měla by vycházet z „podstatných doslovných výňatků“ (Reid a kol. 2005, str. 22).

4.2.4 Analýza v rámci fenomenografie

Stolz (2020) uvádí, že v obou fenomenologických i fenomenografických výzkumech jsou studovány jevy a objekty, jak se jeví lidem. Zatímco ve fenomenologickém výzkumu je zkoumán fenomén jako takový, ve fenomenografickém výzkumu výzkumník zkoumá, jak člověk (nebo skupina) na tento fenomén pohlíží nebo jej chápe.

Fenomenografie se zabývá především empirickými popisy učení, a proto sdílí mnoho podobností s Deskriptivní fenomenologickou metodou (Giorgi, 2009), která je založena na Husserlianské fenomenologii, byť speciálně upravené pro

účely psychologie. Je zajímavé, že Giorgi (1986) tvrdí, že mezi fenomenologií a fenomenografií existuje „silná konvergence“ a podle toho, co se zkoumá (např. povaha zkoumaného jevu), může ve skutečnosti obě výzkumné metody značně přiblížit. Z logického hlediska, protože fenomenologie předchází fenomenografii jako tradici, a s vědomím, že fenomenografie již vybírá z fenomenologie myšlenky, koncepty, metody atd., dává smysl, aby se fenomenografie konsolidovala v rámci širší výzkumné agendy fenomenologie. Stolz (2020) toto ve společenských vědách označuje jako „empirickou fenomenologii“, jak uvádí též ostatní autoři, např. Giorgi (2008) nebo Van Manen (2017).

Ve fenomenografii výzkumník hledá variace způsobů chápání nebo konceptualizace jevu, tj. různé způsoby, jak se jev s různými aspekty lidem jeví (Marton, 1996).

Ve fenomenografii jsou za hodnotné považovány také textové pasáže obsahující úvahy respondentů o jejich zkušenostech, na rozdíl od fenomenologických studií, kde je rozdíl mezi pre-reflexní a reflexní zkušeností zásadní. Marton tedy uvádí, že „strukturu a význam prožívaného jevu lze nalézt jak v pre-reflexních zkušenostech, tak v konceptuálním myšlení“ (Marton, 1996).

Uljens (1996) se vyjádřil k fenomenografickému pohybu a uvedl, že my lidé neustále provádíme nové interpretace. S odkazem na Gestaltovu psychologii tvrdí, že měníme své zaměření vědomí a máme tedy různé jevy jako postavu a pozadí. Dalo by se to chápat tak, že neustále měníme naše pojetí jevů v okolním světě. Uljens (1996) nadále tvrdí, že fenomenografické výsledky jsou neutrální jak s ohledem na jednotlivce, tak i na jejich souvislosti. Jeden jedinec tedy mohl během stejného rozhovoru vyjádřit dvě různé koncepce. Bere to jako argument, že kontext hraje roli při vyjadřování koncepcí během rozhovorů. Tyto dva problémy jsou podle Uljense problematické a pokračuje, že na člověka i svět se při vývoji fenomenografického výzkumu zapomnělo. Zkušenosti se vždy vyskytují v kontextu a zažívá je konkrétní jedinec. Upozorňují na to také Friberg a kol. (2000). V souladu s jejich kritikou je důležité vymezit a definovat studovaný fenomén.

4.3 Sběr dat

Za účelem zodpovězení výzkumných otázek je nezbytné získat dostatečné a relevantní údaje prostřednictvím vhodných výzkumných metod. Bell (2005) zdůrazňuje využít efektivního sběru dat, aby byl problém vyřešen. Švaříček a Šeďová (2014) uvádí, že při výběru metod sběru dat je nutné znát, jaké výsledky mohou mít jednotlivé metody. Švaříček a Šeďová (2014) dále ve své práci píší, že volba metod sběru dat musí odpovídat s výzkumnými otázkami a musí být také v souladu s navrženým cílem daného výzkumu. Yin (2013) zdůrazňuje potřebu triangulace, která vyplývá z etické potřeby potvrdit platnost procesů. U případových studií lze triangulaci provést pomocí více zdrojů dat (Yin, 2013). Švaříček a Šeďová (2014) popisují podrobně typy triangulace: triangulaci metod, triangulaci zdrojů dat, multiperspektivní triangulaci a kombinaci předchozích metod. O triangulaci píší, že její použití vede ke zvýšení validity měření. V našem případě byl sběr dat prováděn pomocí široké škály technik včetně dokumentace, pozorování, polostrukturovaných rozhovorů a dotazníků, aby bylo možné identifikovat a umožnit podrobné porozumění tématu výzkumu. Bell (2005) navrhuje, aby síla výzkumu kvalitativních studií zahrnovala schopnost využívat všechny metodiky v rámci procesu sběru dat a jeho schopnost porovnávat v rámci případu a mezi případy za účelem platnosti výzkumu.

4.3.1 Rozhovory

Švaříček a Šeďová (2014) ve své publikaci píší, že rozhovor je nejčastější metodou sběru dat v kvalitativním výzkumu. Rozhovory pro účely výzkumu se liší od běžných rozhovorů na základě toho, že rozhovory jsou plánované, předem dohodnuté, strukturované, kontrolované tazatelem, mají předem stanovený účel a probíhají mezi dvěma nebo více lidmi. Levy a Powell (2003) uznávají, že rozhovory jsou klíčovým rysem úspěšných případů, protože poskytují nejlepší přístup k interpretacím a názorům účastníků ohledně akcí a událostí, ke kterým došlo. Levy

a Powell (2003) dále zdůrazňují, že polostrukturované rozhovory jsou vysoce flexibilní a jsou také považovány za „nejdůležitější formu rozhovoru v kvalitativním výzkumu“.

Otázky k rozhovoru jsou obsaženy v příloze této práce (B). Rozhovory začaly otázkami na typy softwarových aplikací, které respondenti používají, dále následovaly otázky na pochopení zadaného problému v oblasti objektově orientovaného programování, způsob řešení, případné problémy, jejich překonávání, pocity spojené s řešením úlohy a další.

Otázky rozhovoru byly navíc otevřené, a proto dostali respondenti příležitost nabídnout své názory v jejich vlastním smyslu. Polostrukturované rozhovory s otevřenými otázkami také pomáhají při sběru údajů. Účastníci byli vždy vyzváni, aby své odpovědi podrobněji rozvedli a uvedli příklady, kde to bude možné. Yin (2013) uvádí, že v případové studii je důležité mít otevřené otázky, které respondentům umožní vysvětlit jejich pohled na situaci. Použití polostrukturovaných rozhovorů pomohlo nejen prezentovat účastníky vnímání zkoumaných problémů, ale také poskytl tazateli příležitost požádat o další objasnění a vypracování odpovědí. Cohen a kol. (2000) potvrzují, že použití polostrukturovaných rozhovorů umožňuje sběr bohatých dat, protože jsou považována za užitečnou metodu povzbuzení diskuse o problémech, které by jinak v dotaznících nebyly identifikovány. Podobné doporučení podává i Švaříček a Šeďová (2014).

Průměrná doba rozhovorů byla kolem třiceti minut. Účastníkům byl poskytnut čas na zodpovězení otázek. Ke konci každého rozhovoru výzkumník podporoval otevřenou diskusi, což umožnilo dotazovanému klást otázky a přidávat případné komentáře, které by si přál zahrnout. Yin (2013) doporučuje, aby rozhovory byly spíše řízenými konverzacemi než strukturovanými dotazy. Jinými slovy, i když výzkumník sleduje důslednou linii dotazování, skutečný proud otázek v rozhovoru pro případovou studii by měl být spíše „tekutý než rigidní“ (Yin, 2013).

Rozhovory byly zaznamenány a přepsány do písemné podoby. Polostrukturo-

vané rozhovory jsou často založeny na několika připravených tématech a otázkách. V této technice rozhovoru je zásadní, aby tazatel byl citlivý k dotazovanému a přišel s následnými otázkami v reakci na odpovědi dotazovaného, a není možné pro to vytvořit úplně podrobný plán Cohen a kol. (2000). Tazatel musí být připraven uspořádat pořadí otázek a přinést nové následné otázky, aby zachytil spontánní reakce studenta. Vzhledem k dynamické povaze těchto rozhovorů je důležité zajistit, aby rozhovor pokrýval všechny připravené otázky a témata.

Zaznamenávání rozhovorů nepředstavovalo pro účastníky žádné výzvy, protože byli požádáni, aby projevili ochotu účastnit se cvičení podepsáním a vrácením informovaného souhlasu s výzkumem (příloha A). Nahrávání rozhovorů na pásku se často navrhuje jako prostředek k poskytnutí úplného popisu rozhovorů, odpovědí a komentářů. Shromážděný materiál je rozsáhlý a obsahuje bohatá data o názorech informátorů na jevy, jimiž se tato studie zabývala.

4.3.2 Dotazníky

Dotazníky jsou široce přijímané a používané v oblasti kvantitativního, ale i kvalitativního výzkumu. Mají být navrženy tak, aby poskytovaly přehled o tom, jak se věci mají v určitou dobu (Kelly a kol., 2003). Autor dále tvrdí, že průzkumné dotazníky jsou dobrým prostředkem k pohledu na mnohem větší počet proměnných, než je možné u experimentálních přístupů. Z tohoto důvodu mohou dotazníky poskytnout přiměřeně přesný popis situací v reálném světě z různých hledisek. S výzkumem pomocí dotazníků však souvisejí určité nevýhody; například se obvykle získá malý přehled o příčině nebo procesech za studovaným fenoménem. Plánování a provádění výzkumu pomocí dotazníků lze rozdělit do šesti různých aktivit, které určil Oates (2006): požadavky na data, generování dat, rámec vzorkování, technika vzorkování, míra odezvy a ne-odpovědi a velikost vzorku.

V dotazníku si respondenti přečetli otázky, interpretovali, co se od nich očekává, a poté zapsali své odpovědi. Dotazník byl vyvinut na základě přehledu literatury a byl cílem v rámci našeho výzkumu na otázky týkající se prahových

konceptů (příloha C). Cílem bylo získat další informace, které pak mohly být doplněny více zaměřenými dotazy v rozhovorech.

4.3.3 Pozorování

Pozorování je metodika skládající se ze sledování toho, co lidé dělají, poslechu toho, co říkají, a někdy jsou požádáni, aby objasnili určité problémy. Cohem a kol. (2000) identifikují výhody zapojení do pozorování, které zahrnují sledování toho, co lidé skutečně dělají, spíše než to, co říkají, že dělají, nebo proč a jak by to měli dělat. Data byla zachycena pečlivým sledováním aktivit studentů, kteří byli požádáni, aby říkali nahlas, jak postupují v řešení programovacích úloh, co dělají, co jim dělá potíže a jak se s tím vypořádají. Zároveň mají i uvést, jak se přitom cítí. Získané informace byly dále porovnány s informacemi poskytnutými účastníky během rozhovorů. Adler a Adler (1994) tvrdí, že hlavní výhodou přímého pozorování je skutečnost, že je nenápadné a nevyžaduje přímou interakci s účastníky. Pozorování je důsledné, když je kombinováno s jinými metodami, uvádí Myers (2009), a může osvětlit rozdíly mezi tím, co lidé říkají v rozhovorech, příležitostnými rozhovory a tím, co vlastně dělají. Pomáhá také pozorovat věci, které mohou běžně uniknout vědomému vědomí mezi účastníky. Zdůrazňuje, že „jedním přístupem k porozumění používání technologií je pečlivé sledování studentů v práci“. Yin (2013) zdůrazňuje, že přístup případových studií kombinuje metody sběru dat, jako jsou archivy, rozhovory, dotazníky a pozorování, a dále doporučuje použití protokolu případových studií jako součást pečlivě navrženého výzkumného projektu.

4.3.4 Analýza dokumentace

Tato dokumentace (slovem dokumentace zde označujeme návrh programu zapsaný buď rukou nebo nějakým softwarovým nástrojem, vytištěný kód odpovídající danému návrhu nebo jiný výstup získaný studenty při programování) vytvářená

studenty na základě zadání různých příkladů použita jako základ pro pochopení výsledků studentů. Dokumentace sloužila také k odhalení "mimiker", tj. zda studenti rozumějí danému konceptu a umějí ho použít nebo to jen předstírají. Kromě toho byly informace získané z těchto dokumentů použity k potvrzení a jako doplněk k důkazům shromážděným z jiných zdrojů. Analýza takových dokumentů vytvářených studenty nám umožnila zkoumat některé podrobnosti, čímž se vyhnul rozporům. Závěry byly rovněž získány z dokumentů, které později posloužily jako podněty pro další vyšetřování formou rozhovoru apod.

To také pomohlo poskytnout další důkazy k dalším údajům shromážděným prostřednictvím rozhovorů, ačkoli Yin (2013) uvádí, že výzkumní pracovníci nesmí považovat dokumenty a záznamy za pouhý popis skutečností, které se staly. Myers (2009) však uvádí, že použití dokumentů je důležité, protože mohou být použity jako vstupy při plánování a provádění polostrukturovaných rozhovorů. Analýza dokumentů pomohla porozumět reakcím a pocitům zachyceným v průzkumu a rozhovorech a zajistila, že výsledky budou umístěny do správného kontextu (Grainger a Tolhurst, 2005). Bryman (2006) dále poznamenává, že analýzy dokumentů a záznamů pomáhají zkoumat platnost informací získaných jinými metodami a mohou také poskytnout další informace o problémech, Dokumenty byly analyzovány s ohledem na cíl výzkumu. Toho bylo dosaženo pečlivou analýzou dokumentů se zaměřením na klíčové informace, které byly relevantní pro náš prováděný výzkum. Hlavním důvodem pro zkoumání dokumentů byla podpora faktů již získaných z rozhovorů.

Součásti analýzy „dokumentů" byla i analýza konceptuálních map. Po jejich zakódování do grafů, které provedli podle našich pokynů studenti, jsme se rozhodli analyzovat konceptuální mapy. To znamenalo, že každá konceptuální mapa byla transformována do grafu s uzly a hranami. Poté jsme grafy analyzovali mnoha různými způsoby. Vypočítali jsme číselné charakteristiky, jako je maximální hloubka a vzdálenost mezi určitými uzly. Přirozeně jsme zkoumali, kolik různých konceptů studenti, a jak vnímají souvislost těchto konceptů.

Konceptuální mapy jsou založeny na teorii „že si lidé myslí na koncepty a že tyto konceptuální mapy slouží k externalizaci těchto konceptů a zlepšení jejich myšlení.“ (Novak a Gowin, 1984). Byly použity k tomu, aby pomohly studentům učit se, získat statický obraz o tom, co vědí, a měřit změny v porozumění studentům (Nash a kol., 2006; Steyvers a Tenenbaum, 2005).

Bylo několik návrhů, jak vyhodnotit konceptuální mapy. Některé techniky se spoléhají na kvantitativní analýzu, kde se počítá a porovnává počet uzlů, hran atd. Někteří se dívají na strukturu map. Kinchin a kol. (2000) například navrhují klasifikaci konceptuálních map, ve kterých identifikují tři základní typy: síť, paprsky a řetěz. Jiné techniky využívají kvalitativnější přístup, přičemž se zaměřují na významy štítků uzlů a hran, podobu s předem definovanou „hlavní“ konceptuální mapou nebo kvalitu map hodnocených „hodnotiteli“ (McCure, 1999). Výsledná agregovaná konceptuální mapa je uvedena též v příloze D.

4.4 Analýza získaných dat

Literatura týkající se prahových konceptů neposkytuje příliš mnoho podrobností o metodologickém přístupu, který zvolili příslušní výzkumníci. Proto jsme hledali přístup, který by nám umožnil identifikovat rozdíly a podobnosti mezi zkušenostmi jednotlivých účastníků (studentů). Interpretační fenomenologická analýza poskytuje pro analýzu dat vhodnou metodiku. Co je ale také důležité, tato metodika, podle naší znalosti literatury, vykazuje také podobnost s metodami sběru dat použitými v předchozím výzkumu zabývajících se prahovými koncepty.

Spencer a kol. (2003) uvádějí, že až do druhé poloviny 20. století byla kvalitativní analýza dat z velké části opomíjena. Vysvětlují, že z tohoto důvodu byly kvalitativní metodologické procesy obtížné určit a formulovat. Tvrdí však, že proces „analýzy je náročnou a vzrušující fází procesu kvalitativního výzkumu. Vyžaduje směs kreativity a systematického hledání, směs inspirace a pečlivé detekce“ (Spencer a kol., 2003). To se liší od kvantitativní analýzy dat, která je založe-

na na myšlence převzít velké množství dat pomocí předem určených indikátorů a poté je analyzovat statistickými metodami. Mareš (2015) ve své publikaci píše, že u kvalitativního výzkumu není situace tak přímočará, jako u kvantitativního, kde se předpokládá využití statistických metod na datech, které jsou k takovému zpracování vhodná.

Kvantitativní analýza se zabývá převážně dekontextualizací dat, zatímco kvalitativní analýza dat funguje prostřednictvím dat, protože jsou situována v přirozeném kontextu. V souladu s tím „cílem všeho kvalitativního výzkumu je porozumět fenoménu, spíše než zobecňovat studijní vzorek na populaci na základě statistické inference“ (Forman a Damschroder, 2008). Creswell (1998) vysvětluje, že proces analýzy dat v kvalitativním výzkumu je spirálovitý proces, „který se pohybuje spíše v analytických kruzích než pomocí lineárního přístupu. Zdroje dat mohou být texty, obrázky (např. fotografie, videonahrávky) nebo vyprávění a příběh. Sandelowski (2000) uvádí, že metodický přístup by měl být „dobře zvážený“, protože by měl existovat logický a teoretický smysl pro spojení mezi všemi částmi designu výzkumu. To popisují i Švaříček a Šeďová (2014), kteří píší, že způsob analýzy dat musí odpovídat definovaným výzkumným otázkám.

V souladu s tím jsme shromáždili řadu empirických dat. Při analýze jsme použili zejména induktivní přístup, pokud jde o pokus porozumět konceptu objektově orientovaného programování a funkčním dostupnostem, jak je studenti vnímali. Níže popisujeme stručně použitou obsahovou analýzu. Pozorování a rozhovory, které byly učiněny během několika prvních sezení, hrály důležitou roli, protože pomohly v následujícím kroku vylepšit způsob rozhovorů, pozorování a analýz. Začali jsme uvažovat o tom, které pojmy, procesy a části jsou relevantní, abychom získali představu o širším obrazu.

Proces analýzy zahrnuje rozebrání dat a hledání naléhavých charakteristik a témat. Interpretace dat je potom proces, který z nich extrahuje smysl a sdruží je dohromady. Marshall a Rossman (2006) varují, že vědci však musí být opatrní a dávat pozor, aby kriticky nezpochybnili vznikající vzorce, protože vždy existu-

jí „jiná věrohodná vysvětlení těchto údajů a vazeb mezi nimi. „Alternativní vysvětlení vždy existují a výzkumník je musí identifikovat a popsat a poté prokázat, jaké je vysvětlení, které výzkumník nabízí, „nejpravděpodobnější“ (Marshall a Rossman, 2006).

Forman a Damschroder (2008) uvádějí, že v literatuře neexistuje jasná shoda, jak přesně definovat a provádět obsahovou analýzu. Protože „cílem veškerého kvalitativního výzkumu je porozumět fenoménu, než zobecňovat studijní vzorek na populaci na základě statistické inference,“ (Forman a Damschroder, 2008). Kvalitativní obsahová analýza obvykle zkoumá data, která byla shromážděna, abychom lépe porozuměli nebo popsali určitý jev do hloubky, spíše než používá tabelační proměnné nebo jiná kritéria, která se mají měřit. Výhodou tohoto typu analýzy je, že poskytuje vhled do procesů (Forman a Damschroder, 2008).

Elo a Kyngäs (2008) poznamenávají, že u kvalitativní analýzy se nejedná o lineární proces, protože je méně standardizovaný než kvantitativní metodologie, a je tedy méně formální. Uvádějí, že každá analýza je jedinečná, protože každý dotaz má své vlastní odlišné cíle a motivaci a výsledek závisí na analytických dovednostech a poznatcích výzkumníka. Ten se tedy musí vědomě rozhodnout, které přístupy jsou pro jeho situaci nejlepší. Hsieh a Shannon (2005) popisují dva typy kvalitativní obsahové analýzy. První je to, co označují jako konvenční analýza obsahu, která je induktivní s primárním cílem popsat jev. Druhý, cílenější přístup, může nastat, když výzkumný pracovník použije stávající teoretický rámec, který je již v literatuře vytvořen, jako způsob dalšího porozumění jevu. Jedná se o deduktivní přístup. Náš přístup použitý v rámci této dizertační práci je spíše induktivní, přehledně jsou metody uvedeny v tab. 4.1.

Jak jsme již uvedli, základem pro nás byla fenomenografická analýza. Ve fenomenografické analýze se zpravidla upřesňuje primární zdroj dat přepisem zaznamenaných rozhovorů do textové podoby, kde jsou citace účastníků anonymní. Stále je však možné oddělit jednotlivce pomocí pseudonymů. Dalším krokem je hledání textů pro různé významové výrazy, které se vztahují k určitému jevu. Je

Tabulka 4.1: Přehled použitých metod při analýze dat

Typ dat	Použitá metoda
Rozhovory	Induktivní obsahová analýza – interpretační fenomenologická analýza Deduktivní obsahová analýza – teorie prahových konceptů
Pozorování	Induktivní obsahová analýza – interpretační fenomenologická analýza
Dokumenty (analýza návrhů a kódů studentů při řešení úloh OOP)	Induktivní obsahová analýza – interpretační fenomenologická analýza
Dotazníky	Induktivní obsahová analýza – interpretační fenomenologická analýza Deduktivní obsahová analýza – teorie prahových konceptů

Zdroj: autor

třeba poznamenat, že fragmenty významu byly syntetizovány do samostatných kategorií, které představují různé kvality významu, nicméně, na rozdíl od standardní fenomenografické analýzy, kategorie, které vycházely z analýzy obsahu, nemusí nutně navzájem souviset. To souvisí s použitím návazného výzkumného rámce teorie prahových konceptů.

4.4.1 Induktivní kvalitativní analýza obsahu

Protože jak deduktivní, tak indukční kvalitativní analýza obsahu zahrnují podobné procesy, začínám obecným popisem, který vede ke specifikům o analýze induktivního obsahu a o tom, jak byla aplikována na tuto studii. Chcete-li provést analýzu obsahu, musíte nejprve vytvořit určité kódovací schéma. Kódy jsou klasifikačním systémem pro kvalitativní analýzu dat. Forman a Damschroder (2008) vysvětlují:

Kódy mohou představovat témata, koncepty nebo kategorie událostí, procesů, postojů nebo přesvědčení, které představují lidskou činnost a myšlení. Kódy používá výzkumný pracovník k reorganizaci dat způsobem, který usnadňuje interpretaci a umožňuje výzkumnému pracovníkovi organizovat a načítat data podle kategorií, které jsou pro studii analyticky užitečné, a tím napomáhají interpre-

taci. Promyšlený a uvážlivý vývoj kódů zajišťuje přísnost analytického procesu. Kódy vytvářejí prostředky, pomocí nichž lze důkladně identifikovat a načíst data z datové sady a zároveň umožnit badateli zobrazit obraz dat, který není snadno rozpoznatelný ve formě přepisu ... Kódy mohou být deduktivní nebo induktivní. Deduktivní kódy existují a priori a jsou identifikovány nebo konstruovány z teoretických rámců, relevantní empirické práce, výzkumných otázek, kategorií sběru dat (např. otázek rozhovorů nebo kategorií pozorování) nebo analytické jednotky (např. pohlaví, venkov versus město atd.). Induktivní kódy pocházejí ze samotných dat: analytické vhledy, které se objeví během ponoření do dat a během takzvaného „předběžného kódování". Ačkoli existují studie využívající kódy vyvinuté buď deduktivně, nebo indukčně, analytici obsahu nejčastěji používají kombinaci obou přístupů (Forman a Damschroder, 2008)

Je důležité ujasnit, že při analýze nejde o třídění koncepcí subjektů do předem definované struktury. Jedním ze základních epistemologických předpokladů v rámci fenomenografie jsou vztahy mezi kategoriemi popisu. Různé způsoby, jakými lze fenomén zažít, jsou logicky propojeny navzájem prostřednictvím samotného fenoménu a struktura logických vztahů je obvykle hierarchicky inkluzivní.

Další vlastností výsledného prostoru je kolektivní úroveň popisů vytvořených v kategoriích. Není pravda, že všichni jednotlivci nebo konkrétní jednotlivci mají určitou strukturu způsobu, jak zažít. Analytik se spíše snaží vytvořit kategorie na kolektivní úrovni, a pokud budou úspěšné, mohou být opodstatněné kategorie strukturovány a vzájemně propojeny. To je to, čeho se fenomenografičtí vědci snaží dosáhnout.

Marton a Booth (1997) popisují tři hlavní kritéria pro očekávané vlastnosti výsledného prostoru složeného z kategorií popisu. Prvním kritériem je, že každá kategorie by měla mít zřetelný a jedinečný vztah k jevu podle odlišného způsobu jeho prožívání. To je motivováno skutečností, že fenomenografie je pedagogická výzkumná specializace zaměřená na učení s cílem získat jasný obraz o kvalita-

tivně odlišných způsobech prožívání jevů, které mají vztah k učení.

Druhým kritériem je, že kategorie musí mít k sobě logický vztah, který je často hierarchický a často také inkluzivní. Z pedagogického hlediska existuje norma, která definuje, jaké způsoby chápání (prožívání) určitého jevu jsou lepší než ostatní. Pedagogickým cílem je často to, že student by měl být schopen zažít jevy rozsáhlejším, komplexnějším nebo specializovanějším způsobem, a proto je hledána hierarchická struktura kategorií, která tomuto cíli odpovídá.

Třetím kritériem je, že systém kategorií by měl být co nejkompaktnější. To znamená, že by nemělo existovat více kategorií, než je nutné k vyjádření kritických zkušeností a rozdílů mezi nimi.

Analýza byla cyklickým procesem pohybujícím se mezi fázemi výzkumného procesu (viz obrázek 4.1). Některé počáteční analýzy začaly během sběru dat a během procesu transkripce získaných rozhovorů nebo záznamů z myšlení nahlas. Vždy jsme každý přepis nebo záznam prošli, abychom si připomněli rozsah shromážděného materiálu a nápady vzniklé během rozhovoru. Cílem bylo získat určitý holistický pohled. Poté jsme si zaznamenávali naše počáteční myšlenky na to, abychom zjistili, co bylo významné nebo zajímavé. Při dalším čtení byla poté zdokumentována počáteční nově se objevující témata, čímž se zahájil posun směrem k abstrakci. Jak jsme pokračovali v analýze dalších rozhovorů, vznikající témata byla zaznamenávána do poznámek. Vytvořili jsme si určité kategorie. Poté jsme témata upřesňovali a uspořádávali do klastrů s nadřazenými a podřízenými tématy. Jak jsme pokračovali v analýze, témata byla přejmenována, pozměněna, sloučena nebo rozdělena. Analyzovali jsme přepisy jednotlivě a také jsme se snažili číst mezi řádky, identifikovat jakékoli implikované významy a identifikovat případné mezery (hledat např. kde studenti používají „mimikry“. Této induktivní analýzy spočívala v napsání struktury pohybujícího se mezi popisem a interpretací údajů (Smith a kol. 2009). V rámci výzkumu bude navržena řada programovacích úkolů tak, aby jednotlivé úlohy postupně stavěly na programovacích konceptech a byli patřičně obtížnější či složitější. Zajímavá byla metoda inter-

view založená na myšlení nahlas, během kterého budou studenti pozorováni při řešení úloh. Analýza verbálních protokolů byla použita k získání vzorců chování, které byly následně podle kódů kategorizovány k vyvození závěrů o vztazích mezi kognitivními procesy (verbalizace) a kvalitou vyřešeného programu (řešení úloh) (Atman a Bursic, 1998).

Tyto významové projevy bylo možno identifikovat na několika místech a v různých formách textu. Byly nalezeny významy, kdy respondent výslovně popisuje své zkušenosti s tímto fenoménem jako takovým. Implicitní popisy však mohou také odhalit významy, jako v popisech toho, jak tento fenomén používá, nebo jaké účely, výhody nebo nevýhody tento jev přináší.

Fenomenografický výsledný prostor se vyznačuje kategoriemi popisu a jejich vzájemnými logickými vztahy, obvykle hierarchickou inkluzivitou, což znamená, že význam kategorií se navzájem zahrnuje v tom smyslu, že určité chápání zahrnuje nebo implikuje podobný, elementárnější porozumění. Jelikož fenomenografie pocházela ze studií, které se tak či onak zaměřovaly na pochopení nebo zlepšení formálního učení, bylo rozumné roztřídit výsledný prostor v hierarchii, kde je kvalita každé kategorie oceňována určitým měřítkem v souladu s cíli studie.

Hierarchickou strukturu výsledného prostoru lze potom vysvětlit inkluzivností koncepcí. Popis inkluzivnosti je však trochu vágní, používali jsme následující definici inkluzivity. Vzhledem k tomu, že existuje kategorie A, která kóduje konkrétní způsob prožívání a popisu jevu. Pak je kategorie A zařazena do jiné kategorie B, pokud jejich vztah splňuje následující podmínky:

- Mezi kategorií A a B existuje a není rozporuplný vztah
- Vztah je typu B, který se skládá z A, nebo B je rozšířením A, nebo
- Něco v B předpokládá A.

Během analýzy dat jsme tuto definici použili ke studiu a stanovení vnitřních vztahů a věrohodnosti kategorií souvisejících s ostatními kategoriemi v prostoru

výsledků, prahových konceptů.

Pomocí induktivní analýzy jsme také zkoumali, jaké strategie studenti používají při překonávání problémů, v případě že u nějaké OOP konceptů „uvíznou“. Identifikovali a pojmenovali 28 různých typů přístupů, ty jsme agregovali a seskupili je skupiny 12 kategorií. Výsledné tabulky s uvedením těchto typů kategoií jsou uvedeny v kapitole Zjištění.

4.4.2 Deduktivní kvalitativní analýza obsahu

V rámci této deduktivní analýzy obsahu jsme využili některé přístupy a definované pojmy popsané v literatuře, které se zabývají aplikací teorie prahových konceptů v různých oblastech. Je třeba připomenout, jak jsme již psali v kapitole 2, že teorie prahových konceptů jako taková, nemá zavedený standardizovaný kódovací protokol. Využili jsme proto přístupů popsaných např. autory Yeomans a kol. (2019). To nám posloužilo jako určité kódovací schéma, které tvořilo základ naší deduktivní analýzy s cílem poskytnout popisné chápání toho, jak studenti rozumí a používají základní koncepty objektově orientovaného programování.

Analyzovali jsme tedy přepisy a audiozáznamy rozhovorů a myšlení nahlas na základě teorie prahových koncepcí. Předpokládali jsme, že existují prahové koncepty související s učením objektově orientovaného programování. Hledali jsme důkazy ve shromážděných textech hledáním stop nejdůležitějších klíčových charakteristik, které definují prahové koncepty: transformativní, integrační, nevratné a že jsou problematické. Na druhou stranu jsme pomocí indukční analýzy provedli zajímavé objevy v našich datech, například abychom našli a prozkoumali koncepty a další aspekty, které se objevily v souvislosti s transformacemi, například to, jak studenti reagují a jak se identifikují s objektem jako základem objektově orientovaného programování.

V závěrečné interpretační fázi obsahové analýzy bere výzkumník zprávy o kódu, poznámky, poznámky nebo cokoli jiného; a další analýzy a interpretace (Forman a Damschroder, 2008). Kódy aplikované na data umožňují výzkumnému pra-

covníkovi znovu sestavit data takovým způsobem, který podporuje „koherentní a revidované porozumění nebo vysvětlení“ (Forman a Damschroder, 2008). Klíčem je, že po rozsáhlé analýze a vytvoření struktury v datech je na výzkumníkovi, aby interpretoval, co data znamenají.

Celkový proces, který v této kapitole popisujeme jako lineární, se v této lineární formě nevyskytoval. Během provádění dalších rozhovorů došlo k přepisu předchozích rozhovorů. Analýza některých rozhovorů proběhla, zatímco jiné byly přepsány. Takže obecně byl skutečný proces více „chaotický“ než prezentovaný v této kapitole.

4.5 Interpretace získaných dat

V našem výzkumu používáme pro interpretaci získaných výsledků teorii prahových konceptů. Tuto teorii, jak jsme již uvedli, vytvořili Meyer a Land (2003). Tito autoři definují koncept prahové hodnoty jako „podobný portálu, který otevírá nový a dříve nepřístupný způsob uvažování o něčem“. Mají za to, že student musí koncept pochopit nebo prohlédnout, než bude schopen v tomto předmětu pokročit v porozumění. Změna v chápání nebo pohledu může souviset s předmětem, zastřešujícím pohledem na předmět nebo světovým pohledem. Ve své práci Meyer a Land (2003) píší, že koncept prahové hodnoty má pět charakteristik: transformativní, pravděpodobně nevratný, problematický, integrativní a omezený. Můžeme to označit jako „model kritérií“, který je v našem výzkumu základní a je to východisko i pro interpretaci informací získaných analýzou dat.

Interpretace v rámci teorie prahových konceptů není jednoduchá. Literatura zabývající se prahovými koncepty říká, že stejný koncept může být prahovou hodnotou v rámci jedné disciplíny, například ekonomie, ale nikoli jiné (Cousin 2008). Například prahové hodnoty v jednom oboru, který má různé specializace, ale stejný základ, například mikroekonomie a makroekonomie, mohou být prahové hodnoty chápány stejně.

Zajímavá je charakteristika nazvaná „mimikry“, kterou jsme také často identifikovali a interpretovali. Spočívá v tom, že během období liminality existují situace, kdy se zdá, že si student osvojil tento koncept, ale ve skutečnosti tomu tak nebylo. Meyer a Land (2005) nazval toto nedorozumění „mimikry“. Tato mimikry může zahrnovat spíše laický pohled na koncept. Žák může být schopen přednést definici pojmu, ale plně jí nerozumí. Mohou také být schopni použít koncept v rámci svého předmětu, ale neaplikovat ho na svůj širší svět. Cesta studenta při získávání konceptu prahové hodnoty může zahrnovat pokusy o porozumění i nedorozumění nebo o „kompenzační mimikry“ (Meyer a Land, 2005).

Při analýze a v následné interpretaci bylo také důležité prozkoumat, jak předchozí znalosti a zkušenosti studentů ovlivňují jejich učení. Přednášející si mohou být vědomi toho, že student nerozumí nikoliv konceptu, ale tomu, jak vše souvisí se „správným“ chápáním tohoto konceptu (Laurillard 1993). K tomu dochází v programování, kdy studenti mísí pojmy objekt a třída a lektoři nechápou, jak a proč k této mylné představě dospějí. Analýza toho nebyla jednoduchá, ale mohla být provedena na základě polostrukturovaných rozhovorů.

4.6 Validita a reabilita výzkumu

Loh (2013) diskutuje o důvěryhodnosti v rámci kvalitativního výzkumu a tvrdí, že tento výzkum je na rozdíl od pozitivistických tradic nevyhnutelně spojen se subjektivními hodnotami. V kvalitativním výzkumu je zásadní ukázat, že vybrané výzkumné metody vhodným způsobem odrážejí cíle výzkumu a také ukázat, jak výsledky využít. Booth a kol. (2001) pojednávají o těchto věcech ze své vlastní dlouholeté zkušenosti s programováním a ze své znalosti a dobrých vztahů se studenty, když se podíleli na jejich studiu. V tomto výzkumu neexistují absolutní pravdy, a proto tvrdí, že výzkumník musí přesvědčivě argumentovat pro zvolené metody, výsledky a interpretace.

Yin (2013) vysvětluje, že „protože výzkumný návrh má představovat logickou

sadu tvrzení, můžete podle určitých logických testů také posoudit kvalitu jakéhokoli daného návrhu“. Pokud jde o empirický sociální výzkum (který tato studie představuje), Yin (2013) uvádí, že existují čtyři testy v rámci kvalitativního výzkumu. Těmito testy jsou platnost konstrukce, vnitřní platnost, vnější platnost a spolehlivost. V průběhu studie se používají taktiky zajišťující, že případné studie obstojí proti těmto testům, počínaje návrhem výzkumu. Totéž také uvádí u nás Mareš (2015). Uvádí, že výzkumné projekty kvalitativního výzkumu by měly tyto testy zahrnout. V našem případě bylo ověření našich výzkumů důležité, protože fenomenografická analýza má subjektivní povahu. Odráží způsob, jakým výzkumník objevuje a analyzuje významy v získaném materiálu (nejčastěji textovém).

4.6.1 Konstruktová validita

Konstrukční validita ukazuje, jak dobře test nebo experiment odpovídá svým nárokům. Odkazuje na to, zda pracovní definice proměnné skutečně odráží skutečný teoretický význam pojmu. Platnost konstruktu si můžeme představit jako problém „označování“. Pokud měříme to, co nazveme např. „mimikry“, je to, co jsme skutečně měřili? Podle Yin (2013) se platnost konstruktu týká zajištění identifikace správných pracovních opatření pro zkoumané koncepty. Snažili jsme se proto zajistit platnost a spolehlivost prostřednictvím jasných vazeb mezi výzkumnou otázkou, důkazy a závěry, které mají být v rámci studie prokázány. Yin (2013) zdůrazňuje, že hlavním kritériem pro stanovení platnosti konstruktu je stanovení klíčových konceptů a volba vhodných indikátorů zkoumaného jevu.

Ve své publikaci se problémem validity a reliability zabývá Mareš (2015). Popisuje postupy pro ověření konstruktové validity. Podobně jako Švaříček a Šeďová (2014) doporučuje po získání dat a zformování předběžných pracovních konstruktů, aby respondenti okomentovali navržené konstrukty, a případně je upravili. Punch (2013) to nazývá také jako „členské kontroly“, uvádí je také jako strategii pro zajištění vnitřní platnosti.

Abychom tedy zajistili přesnost našich konstruktů (jak analyzujeme a interpretujeme jejich porozumění a činnost u řešení jednotlivých programových úloh), požádali jsme účastníky našeho výzkumu, aby se podívali na koncept našich zjištění. Na základě pozitivní zpětné vazby od studentů respondentů jsme dospěli k názoru, že naše konstrukty týkající se konceptů ve výuce objektově orientovaného programování jsou vnitřně validní.

4.6.2 Vnitřní validita

Yin (2013) píše, že vnitřní validita souvisí s výzkumem, jehož cílem je identifikovat kauzální vztahy, kde „pokud vyšetřovatel nesprávně dojde k závěru, že existuje kauzální vztah mezi x a y, aniž by věděl, že některý třetí faktor, z, mohl ve skutečnosti způsobit y, návrh výzkumu se nepodařilo vypořádat s určitou hrozbou pro vnitřní platnost ". Jiným konceptem vnitřní validity je posílení přesnosti kauzálních vztahů mezi konstrukty. Zdůrazňuje proto, že vnitřní platnosti se v experimentálních a kvazi experimentálních studií věnuje největší pozornost. Podle našeho názoru je praktičtější přístup, jak jej pojal Punch (2013). Autor tvrdí, že vnitřní validita je obvykle nejjasněji chápána v kvantitativním kontextu, přičemž „to znamená, do jaké míry jsou vztahy mezi proměnnými správně interpretovány". Protože interní validita souvisí s přesností, Punch (2013) uvádí, že podobný, ale širší pohled na interní validitu je aplikován na kvalitativní výzkum. Proto se „interní validita zabývá otázkou, jak výsledky výzkumu odpovídají realitě. Jak shodná jsou zjištění s realitou? Zachycují nálezy, co tam opravdu je? Pozorují nebo měří vyšetřovatelé to, co si myslí, že měří? " (Punch, 2013).

Druhá otázka, kterou Punch představuje, byla řešena Yinovou (2013) představou platnosti konstruktu. Punch dále objasňuje, že pojem platnosti v kvalitativním výzkumu se také někdy označuje jako důvěryhodnost, aby se odlišil od pozitivistických perspektiv. Punch (2013) vysvětluje:

„Jedním z předpokladů kvalitativního výzkumu je skutečnost, že realita je holistická, multidimenzionální a neustále se měnící; nejedná se o jediný, fixovaný,

objektivní jev čekající na objevení, pozorování a měření jako v kvantitativním výzkumu. Hodnocení izomorfismu mezi shromážděnými údaji a „realitou“, ze které byly odvozeny, je tedy nevhodným determinantem platnosti“. Punch (2013)

Existují postupy, které by měly být použity k prokázání přesnosti (tj. vnitřní validity nebo generalizace) studie. Patří mezi ně přiměřené zapojení do sběru dat, triangulace, flexibilita výzkumných pracovníků a kontroly členů.

Vnitřní validitu jsme ověřovali opět členským ověřováním. Pokud jde o tuto taktiku, Punch (2013) vysvětluje, že „účastníci by měli být schopni rozpoznat své zkušenosti ve vaší interpretaci nebo navrhnout jemné doladění, aby lépe zachytili jejich perspektivy“. Studenti dostali návrh našich zjištění. Konkrétně jsme je požádali, aby řešili jakékoli nesrovnalosti ohledně těchto zjištění. Naše členské kontroly většinou potvrdily, že naše zjištění vystihovala realitu, bylo jen třeba doladit drobné detaily (například správné vystižení pocitů, nebo opravení porozumění některým konstruktům). Dále jsme použili metodu triangulace, (Švaříček, Šeďová, a kol., 2007) pro úplnost a porozumění, s daty shromážděnými z různých pohledů (rozhovory, dotazníky, pozorování, dokumentace). Konstatujeme, že zjištění diskutovaná v následujících kapitolách jsou interně validní.

4.6.3 Vnější validita

Pojmy vnější validita a zobecnitelnost jsou v kvantitativním výzkumu v zásadě synonyma (Punch, 2013). Švaříček a Šeďová (2014) popisují vnější validitu jako schopnost zobecnění výsledků dané kvalitativní studie na širší populaci. Yin (2013) poukazuje na to, že test, zda je možné zobecnit ukazatele mimo rámec kvalitativní studie, může být hlavní překážkou kvalitativního výzkumu zejména u případových studií.

Punch (2013) uvádí, že pokud jde o kvalitativní výzkum, neměli bychom konceptualizovat zobecnitelnost stejným způsobem, jako by to bylo u kvantitativního výzkumu. Tyto metody jsou navrženy tak, aby zajistily, že výsledky budou zobecnitelné pro širokou populaci. V tomto smyslu Eisenhart (2009) tvrdí, že po-

kud si chtějí kvalitativní výzkumníci přát přistupovat ke generalizaci ze stejné perspektivy jako kvantitativní výzkumník, musí velmi pečlivě navrhnout studii předem tak, aby byla „typickým“ případem, nebo výběrem více stránek pro srovnání témat mezi nimi. V kvalitativním výzkumu je vnější validita označována také jako přenositelnost nebo aplikovatelnost. Punch (2013) uvádí, že “Výzkumník musí poskytnout„ dostatečná popisná data “, aby byla přenositelnost možná“. Švaříček a Šeďová (2014) uvádějí podobně, že zpravidla není možné zobecnit závěry kvalitativního výzkumu ve všech prostředí. Nicméně výzkumník by měl popsat provedení výzkumu, výchozí konstrukty a závěry tak, aby bylo možné posoudit přenositelnost závěrů na jiná prostředí. Také např. Eisenhart (2009, s. 60) zdůrazňuje:

„Při snaze o teoretické zobecnění se výběr skupiny nebo místa ke studiu provádí na základě pravděpodobnosti, že případ odhalí něco nového a jiného a že jakmile bude tento nový fenomén teoretizován, další případy odhalí rozdíly nebo variace, které testují jeho zobecnitelnost. Kritériem pro výběr případů, z nichž se bude zobecňovat, není náhodný nebo reprezentativní výběr vzorků, ale míra, do jaké je pravděpodobné, že vybrané případy vytvoří, obnoví nebo vyvrátí teorii“. Eisenhart (2009, s. 60)

Náš výzkum se týká výuky objektově orientovaného programování. Naši analytickou jednotku tvořili studenti, kteří jsou převážně studenti oboru Ekonomická informatika. Jedná se o standardní obor v zahraničí označovaný jako Business Informatics. Výuka objektového programování v podstatě probíhá způsobem, který odpovídá zhruba doporučeným standardům ACM (JTFCC, 2013). Takové složení „výzkumné skupiny“ nám umožnilo smysluplný vzhled do problematiky a porozumění problematice učení se objektovému programování. Vnější validitu jsme se také snažili zajistit způsobem, který doporučuje Merriam (2015). Jedná se o poskytnutí bohatých a podrobných popisů nastavení výzkumné studie, účastníků a zjištění. To umožňuje čtenářům „určit, do jaké míry se jejich situace shoduje s kontextem výzkumu, a tedy zda lze zjištění přenést“ (Merriam,

2015). Vnější validitu také vymezujeme popisem limitů našeho výzkumu.

4.6.4 Reliabilita

Závěrečným testem kvality výzkumné studie je spolehlivost. Yin (2013) vysvětluje, že cílem spolehlivosti je:

„Chcete-li si být jisti, že pokud pozdější vyšetřovatel použil stejné postupy, jaké popsal dřívější vyšetřovatel, a provedl stejnou případovou studii znovu, měl by dospět ke stejným zjištěním a závěrům". (Yin, 2013).

Merriam (2015) poznamenává, že tento koncept je zvláště problematický ve společenských vědách z jednoduchého důvodu, že „lidské chování není nikdy statické". Podobně píší o reliabilitě i Švaříček a Šeďová (2014). Píší, že reliabilitu (spolehlivost) lze zajistit konzistencí otázek, přepisem nahrávek rozhovorů, správnou činností při kódování. Uvádějí také, že v kvalitativním výzkumu často spolehlivost nemusí být vysoká, důvodem je, že ne všechny metody jsou v takovém typu výzkum standardizované. Píší také, že výzkumník musí být pečlivý a pracovat jen se získanými informacemi, nesmí si informace domýšlet nebo doplňovat.

V našem výzkumu jsme pro zajištění reliability (spolehlivosti) využili postup, který doporučuje Merriam (2015). Zaměřujeme se na pečlivou dokumentaci jednotlivých kroků našeho výzkumu, na podrobný popis toho, jak byl náš výzkum proveden a jak byla data analyzována.

4.7 Shrnutí metodologie

Metodika v oblasti výzkumu prahových konceptů není v současné době standardizovaná. Teorie prahových konceptů hodnot podrobně nerozhoduje o použitém metodickém přístupu. Podrobně popisuje různé metody sběru dat a výběr účastníků, přičemž převažují rozhovory se studenty. Některé články pojednávající o prahových hodnotách používá jako účastníky zaměstnance i studenty. V naší

práci jsme jako metodický a analytický přístup použili interpretační fenomenologickou analýzu. Interpretační fenomenologická analýza se zaměřuje na zkušenosti lidí se zkoumaným jevem a jejich interpretaci. Uznává roli výzkumníka, který musí následně interpretovat zkušenosti účastníků. Přechod výzkumu od jednotlivce k obecnému je rysem výzkumu interpretační fenomenologické analýzy. Domníváme se, že tyto vlastnosti znamenají, že interpretační fenomenologická analýza je vhodným metodickým přístupem k výzkumu obtížných a potenciálně prahových konceptů. V této kapitole jsme popsali plán a design našeho výzkumu. Dále jsme popsali způsoby sběru dat, způsoby rozhovorů, pozorování a analýzy dokumentace. Poté jsme představili použité metody analýzy dat, induktivní a deduktivní obsahovou analýzu. Kapitola je zakončena popisem postupů pro splnění požadavků na validitu a reliabilitu našeho výzkumu.

5 Zjištění výsledků

V této kapitole se věnujeme výsledkům zjištění, která výplývají z našeho výzkumu definovanému výzkumnými otázkami RQ1 až RQ3. Zaměřili jsem se na to, jak studenti chápou, jak rozumí základním konceptů OOP přístupu k programování, které jim dělají případně potíže a jak jsou schopni tyto koncepty využívat při řešení svých úloh.

Při tomto výzkumu jsme použili induktivní a deduktivní analýzu obsahu, využili fenomenograficou analýzu. Základem pro nás pak byla teorie prahových konceptů formulovaná Meyerem a Landem (2003, 2005). U jednotlivých konceptů jsme přiřazovali případně jednotlivé charakteristiky definované touto teorií. Jednotlivé charakteristiky konceptů byly posuzovány na základě vyjádření studentů, jejich reakcí při řešení jednotlivých úkolů, na základě strukturovaných rozhovorů vedených se studenty a na základě detailní analýzy získaných dokumentací od studentů (návrhů a kódu jejich programů).

5.1 Osnovy bakalářského studia informatiky na Jihočeské univerzitě Ekonomické fakulty v Českých Budějovicích

Účastníky výzkumu byli studenti oboru Ekonomická informatika na Ekonomické fakultě Jihočeské univerzity. Výzkum proběhl v akademickém roku 2019/2020 a 2020/2021 v rámci dvou semestrového předmětu s náplní výuky programování. Pozorování a nestrukturované rozhovory s vysokoškolskými studenty prvního ročníku probíhaly po dobu dvou semestrů v průběhu akademického roku 2019/2020, kdy studentům byly kladeny otázky týkající se jejich zkušeností s programováním a byly analyzovány výsledky jejich programových úloh. I když tato data nebyla při analýze rozsáhle využívána, vytvořila zásadní krok při informo-

vání otázek kladených pro další fázi sběru dat a také, v praxi, ke zlepšení míry účasti studentů, když byli pozváni k účasti v cílové skupině navržené tak, aby bylo možné cíleněji zkoumat potenciální koncepty prahových hodnot.

Tento cílený výzkum byl proveden v průběhu dvou semestů v průběhu akademického roku 2020/2021. Ve snaze dosáhnout určité shody ohledně koncepcí prahových hodnot pro začínající programátory studentů byla shromážděna data od studentů prvního ročníku na Ekonomické fakultě Jihočeské univerzity oboru Ekonomická informatika. Obsah předmětu Programování, který je dvousemestrový, u těchto studentů odpovídá doporučením organizace ACM a IEEE (JTFCC, 2013) a odpovídá obsahu výuky předmětu programování na vysokých školách s informatickým zaměřením. Účastníci byli požádáni zejména o popis aspektů programování, které považovali za náročné, ale byli také požádáni, aby diskutovali o všech koncepcích, které změnily jejich vnímání samotného programování. Při těchto rozhovorech byly již používány polostrukturované rozhovory a byly využity i konceptuální mapy pro získání přehledu, jak studenti chápou základní koncepty objektového přístupu v programování a vztahy mezi nimi.

Prvním kurzem obsahujícím objektově orientované koncepty je „Základy programování“. Zde jsou představeny mimo jiné objekty, třídy, metody, dědičnost, abstrakce a polymorfismus. Účastníkům je umožněno řešit algoritmické problémy a jednoduché programy s použitím objektově orientovaného přístupu v jazyku C♯.

Kromě kurzu Základů programování existuje navazující kurz „Objektově orientované programování“, který je založen na koncepcích OOP. Účastníkům je dále umožněno porozumět jazyku C♯. Mohou sami vyvíjet malé aplikace pomocí základních konceptů počítačové vědy. Výuka na EF JU je tedy typu object-first, zavádí objektově orientované pojmy hned na začátku.

V dalších odstavcích uvádíme základní koncepty OOP a charakteristiky podle teorie prahových konceptů, tak jak jsme je přiřadili na základě analýz. Je třeba poznamenat, že přiřazení bylo provedeno autorem této dizertační práce, v případě pochyb pak bylo příslušné přiřazení diskutováno i s druhým učitelem, který

má zkušenosti s výukou programování.

Zároveň jsme sledovali strategie, které studenti použili, pokud se „zasekli“ (nemohli pochopit a využít určitý koncept při řešení svých úloh, nevěděli co s tím, mohli bychom to také označit, že byly v určitém liminálním prostoru. V našem výzkumu jsme tedy zjišťovali, jaké strategie studenti používají při pokusu osvojit si nové počítačové koncepty a dovednosti. Tyto strategie studenti použili, pokud nepochopili dané pojmy a postupy již v rámci prvotní výuky. Jsou to tedy určité strategie „nad rámec výuky“. Tyto strategie jsme kategorizovali. Analyzovali jsme širokou škálu strategií, které jsme zařadili v této kapitole u jednotlivých konceptů do řady rozpoznatelně odlišných kategorií.

5.2 Koncepty OOP a charakteristiky dle teorie prahových konceptů

V následujících podkapitolách uvádíme základní koncepty OOP. U každého konceptu popisujeme základní problémy, které daný koncept řeší, přiřazujeme charakteristiky příslušného konceptu a uvádíme přístupy studentů k ovládnutí popisovaného konceptu. Rovněž pro každý koncept popisujeme strategie, které studenti používali pro překonání problémů s uvedeným konceptem.

5.2.1 Koncept Syntaktické elementy

V každém programovacím jazyku je nutné dodržovat pravidla, která jsou konkrétním jazykem určena. To zahrnuje například způsoby deklarace proměnných, používání předdefinovaných datových typů, práce s operátory, způsob práce s řídícími strukturami (např. cykly, větvení), ale též způsob práce s programovými strukturami, a v případě OO jazyka veškeré koncepty týkající se objektově orientovaného programování. Aby vše fungovalo a bylo možné výsledný program ze zdrojového kódu sestavit, je nutné dodržet syntaktická pravidla jazyka.

Jaké problémy koncept Syntaktické elementy řeší

Tento koncept je zásadní pro správný a funkční zdrojový kód. Ovládnutí syntaxe konkrétního programovacího jazyka je základní predispozice pro schopnost tvorby základních i pokročilejších programů a používání ostatních konceptů programátorem. Pokud zdrojový kód obsahuje syntaktické chyby, nepodaří se kompilátoru kód přeložit a následně nebude možné sestavit z přeloženého kódu spustitelný program (aplikaci). V pokročilých vývojových prostředích (např. MS Visual Studio) je již v prostředí editoru možné zdrojový kód přeložit a při neúspěchu informovat programátora o syntaktických chybách, jejich umístění ve zdrojovém kódu a možných způsobech jejich nápravy.

Přiřazení charakteristik a přístup studentů konceptu Syntaktické elementy

Na základě rozhovorů, vyjádření studentů, jejich reakcí při řešení úkolů a analýzy získané dokumentace přiřazujeme tomuto konceptu charasteristiky podle teorie TC:

- nezvratný,
- integrativní,
- ohraničený a
- diskurzivní.

Tento koncept zahrnuje jednoduché pojmy a operace, proto jej dle taxonomie SOLO řadíme do úrovně *unistrukturální*.

U získané dokumentace od studentů je patrné, že jejich úlohy, které odpovídají dle taxonomie SOLO *unistruturální* úrovni, jsou studenti schopni snadno pochopit a vyřešit.

Během výzkumu jsme často zaznamenali, že probandi se zastavili při vytváření výsledného modelu programu a zjišťovali, jak příslušný syntaktický element správně použít ve svém programovém návrhu. Jednalo se nejčastěji o případy již

předpřipravených datových struktur, např. kolekce typu List, apod., kdy probandi opomněli drobné skutečnosti, na které kompilátor často správně upozornil. Následující ukázka tento případ demonstruje (proband F).

> *„Budu vycházet z návrhu, který jsme si udělali na posledním cvičení, který jsem si prostudoval a srovnal jsem si myšlenky, jak pokračovat dál.*
>
> *Začnu tím, že si vytvořím... další objekt (pozn.: myšleno novou třídu) Display.*
>
> *Chtěl bych zkusit jednotlivé displaye ukládat do Listu.*
>
> *Založím si ho... Proč mi to svítí červeně?*
>
> *Jo... Using*
>
> *Dobrý opraveno "* (proband F)

Celkově jsme nezaznamenali závažnější problémy u studentů při řešení zadaných úloh. Z tohoto důvodu tento koncept nepovažujeme za problematický. Syntaktické elementy umožňují studentů řešit základní úlohy algoritmického typu, používat základní struktury jazyka. Tento koncept považujeme proto za *diskurzivní*, protože studenti zvládnou základní terminologie programování, a *ohraničený*, protože koncept Syntaktické elementy může být spojen s určitým programovým prostředím nebo jazykem. Dále jej charakterizujeme jako *nezvratný*, neboť naučený koncept nadále bez větších potíží studenti využívali i v následujících řešených úkolech.

Strategie studentů pro překonání problémů s konceptem Syntaktické elementy

Studenti si poměrně rychle osvojili tento koncept (základní programové syntaxe) již z výuky. Použití dalších strategií (viz tabulka 5.1) pro překonávání problému s tímto konceptem proto nebylo příliš rozšířené.

Dílčí problémy, které ale nesouvisely s porozuměním konceptu, spočívaly v tom, že si často studenti nevzpoměli na správný tvar daného programového prvku. Studenti však měli v paměti, jak to asi má být (proto také přiřazení charakteristiky *nezvratný*). Potom využívali trasování (to znamená, že využívali výhod vývojového prostředí, kde mohli sledovat průběh vykonávání kódu krok za krokem a zjistit, kde udělali chybu, kde napsali špatně syntaxy určitého prvku, atd.).

Tabulka 5.1: Přístup a strategie studentů pro překonání problému s konceptem Syntaktické elementy

Použitá strategie	Přístup	Procento studentů využívajících danou strategii
Učí se od ostatních lidí	Učí se od spolužáků	5,3
	Získá pomoc od učitele	15,8
Učí se z psaných materiálů	Čte instrukce z knihy	10,5
	Čte instrukce z internetu	0
Učí se z příkladů	Používá vlastní příklady	0
	Hledá příklady na internetu	5,3
Užití jiného přístupu	Pokus a omyl	0
	Vyhnutí se problému	0
	Udělá si přestávku a zkusí řešit po přestávce	0

Zdroj: autor

5.2.2 Koncept Třída

Třída je uživatelský datový typ. Jedná se o předlohu (šablonu), podle které bude následně objekt vytvořen. Základními členy třídy jsou, shodně s objektem, data (atributy) a metody (funkce definované v kontextu třídy). Následující tabulka 5.2 představuje všechny členy třídy (a tedy i následného objektu) s jejich popisem.

Tabulka 5.2: Základní vlastnoti třídy – členy třídy

Členy třídy	
Člen	Popis
Pole	Pole jsou proměnné deklarované v rozsahu třídy. Pole může být předdefinovaný číselný typ nebo instance jiné třídy. Například třída kalendáře může mít pole, které obsahuje aktuální datum.
Konstanty	Konstanty jsou datová pole, jejichž hodnota je nastavena v době kompilace a nelze je změnit.
Vlastnosti	Vlastnosti jsou metody ve třídě, ke kterým se přistupuje, jako by šlo o datové pole (atribut) v této třídě. Vlastnost může poskytnout ochranu pro atributy třídy, aby nedošlo ke změně bez znalosti objektu.
Metody	Metody definují akce, které může třída provádět. Metody mohou brát parametry, které poskytují vstupní data, a mohou vracet výstupní data prostřednictvím parametrů. Metody mohou také vrátit hodnotu přímo, bez použití parametru.
Události	Události poskytují oznámení jiným objektům. Události jsou např. kliknutí na tlačítka myši nebo úspěšné dokončení metody. Události jsou definovány a spouštěny pomocí delegátů.
Operátory	Přetížené operátory jsou považovány za členy typu. Při přetížení operátor definujeme jako veřejnou statickou metodu.
Indexery	Indexery umožňují, aby byl objekt indexován a zpřístupněn podobným způsobem jako typ pole.

Konstruktory	Konstruktory jsou metody, které se volají při prvním vytvoření objektu. Často se používají k inicializaci dat objektu.
Destruktory	Finalizátory se v C♯ používají velmi zřídka. Jsou to metody, které jsou volány modulem spuštění modulu runtime, když má být objekt odstraněn z paměti. Obvykle se používají k zajištění toho, aby se všemi prostředky, které musí být uvolněny, bylo zacházeno odpovídajícím způsobem.
Vnořené typy	Vnořené typy jsou typy deklarované v rámci jiného typu. Vnořené typy se často používají k deklaraci objektů, které dané typy používají a které je obsahují.

Zdroj: upraveno dle (Microsoft, 2021a)

Jaké problémy koncept Třída řeší

Celé paradigma objektově orientovaného programování se opírá o koncepty objekt a třída.

Používají se k rozdělení počítačového programu do jednoduchých a znovu použitelných částí – tříd, ze kterých se následně vytváří objekty a konkrétní instance těchto objektů.

Z podstaty tohoto konceptu je tak zřejmé, bez třídy bychom nemohli vytvářet objekty a jejich instance. Třída je základním elementem a ostatní koncepty jsou s ní určitými procesy pevně propojeny.

Přiřazení charakteristik a přístup studentů konceptu Třída

Na základě rozhovorů, vyjádření studentů, jejich reakcí při řešení úkolů a analýzy získané dokumentace přiřazujeme tomuto konceptu charakteristiky:

- integrativní,
- ohraničený,
- diskurzivní.

Na následující ukázce je vidět, že studenti měli obtíže rozeznat mezi elementy atribut (vlastnost) a metoda (schopnost). Dělalo jim problémy porozumět vlastnosti a metodě u konceptu Třída, nicméně, ve velmi krátké době tento koncept ovládli. Proto ho nepovažujeme za problematický.

„M: Tak, to jsou teda zbraně jako takový

J: Takže co má?

M: Momentálně nic, máme zaseklý VisualStudio

J: Ahhh

J: Co máme umět, máme umět... Zabojovat se zbraní

M: Takže to budou –

J: Zbraň bude mít vlastnost ... metodu

M: Útok

J: Teoreticky by jich mohlo být víc, kdybychom měli sekání sekerou ručně nebo házení sekerou

M: Hmm

J: Zatím bych to neřešil

M: Nebudeme to komplikovat.. Je to stejný princip

J: Necháme metodu pro útok.. Tady bude hlavní... kdyžtak pak přidáme –, každá bude mít –

M: Specifický útok "

Koncept Třída (spolu s konceptem Objekt) je základním kamenem objektově orientovaného programování. Třída tvoří základ celého paradigmatu. Jakmile student pochopí tento koncept, je schopen základním způsobem objektově programovat. To je vidět z následujícícho, kdy proband byl schopen vytvořit určitý model reálné situace: (J1 – proband, V – výzkumník).

„J1: Vytvořím si v class diagramu novou třídu. Pojmenuju jí.

V: Proč J1 vytváříte novou třídu?

J1: Protože chci vytvořit toho bojovníka tak potřebuju si ho nějak.

Potřebuju nějakou šablonu podle, který bude vznikat. Potom budu potřebovat nějakou třídu zbraně. Vytvořím si další třídu zbraně.

Vytvořím si další dvě třídy a ty budou dědit ze třídy zbraně a vytvořím si třeba dvě. Třeba meč, sekera. "

Koncept Třída umožňuje propojit další koncepty, zejména předchozí koncept Syntaktické elementy. Porozumění tomuto konceptu znamená, že student musí integrovat další pojmy a postupy z oblasti programování, jako jsou metody (resp.

funkce), datové typy a další. Proto tomuto konceptu přiřazujeme charakteristiku *integrativní*.

Proband M2:

> *„Udělám si metodu ve třídě WEATHERDATA a pojmenuji Display_Name*
>
> *Mám zde hotové metody, jako GetHumidity, GetPressure atd...*
>
> *Dále potřebuji metodu, kterou nastavím jednotlivé hodnoty Takže si udělám novou metodu ve třídě WEATHERDATE , bude to metoda SETMEASSUREMENTS a ude mít parametry ty naše hodnoty Nadefinuju si jí... Takže rovná se.. Tady musí být THIS u všeho Teď jsem v mainu a vytvořím si instanci, přes kterou budu volat jednotlivé metody - ještě že umím syntaxy, hm ...*
>
> *Takže si jí založím....(wstation) “*

Charakteristiku *nezvratný* jsme přiřadili, protože studenti byli schopni používat koncept Třídy a posunuli se alespoň do určité úrovně do nového paradigmatu – objektově orientovaného programování. Např. proband M1 to okomentoval při řešení úlohy:

> *„To fakt jinak nejde než přes třídy a objekty, já už to používám všude, tak udělám si třídu WeatherData*
>
> *Metody GetHumidity, GetPressure a GetTemperature už tam jsou daný ...*
>
> *Pak tam musím mít metodu, kterou nastavím jednotlivé hodnoty*
>
> *To bude ... udělám si novou metodu do WeatherData, nazvu to SetValues*
>
> *Tak ... Jo, ještě this k těm datům...*
>
> *Teď už jen v mainu vytvořím instanci, přes kterou budu volat ty moje metody*
>
> *A mám to připravený....(WeatherMeteo) “*

Konceptu jsme přiřadili charakteristiku *diskurzivní*, studenti používají jazyk OOP. Koncept Třída, jak jsme uvedli, je základním kamenem objektově orientovaného přístupu. Poměrně dobře to bylo vidět z konceptuálních map, které studenti měli za úkol udělat (popis je uveden v kap. 5.3).

Strategie studentů pro překonání problémů s konceptem Třída

Nejčastější strategií při překonávání problémů s konceptem Třída byla konzultace se spolužáky, kteří již chápali tento pojem.

Proband J1 uvedl:

„Třída mi za začátku dělala trochu problémy, já vím, že to není složité, seděl jsem nad tím, potom jsem se zeptal souseda, který to fakt dobře vládl. Docela mi to vysvětlil, ale potm jsem šel do knih a musel si to dát z příkladů "

Proband K1 se zmínil, že mu pomohl internet:

„Měl jsem knihu, ale z ní jsem to moc nechápal. Já jsem na netu našel jakýsi serál o OOP a o UML a tam bylo dost příkladů, tak jsem to nasál a dobrý. "

Tabulka 5.3: Přístup a strategie studentů pro překonání problému s konceptem Třída

Použitá strategie	Přístup	Procento studentů využívajících danou strategii
Učí se od ostatních lidí	Učí se od spolužáků	21,1
	Získá pomoc od učitele	5,3
Učí se z psaných materiálů	Čte instrukce z knihy	5,3
	Čte instrukce z internetu	0
Učí se z příkladů	Používá vlastní příklady	10,5
	Hledá příklady na internetu	5,3
Užití jiného přístupu	Pokus a omyl	5,3
	Vyhnutí se problému	0
	Udělá si přestávku a zkusí řešit po přestávce	0

Zdroj: autor

Požití dalších strategií při překonání problémů s konceptem Třída uvádí tab. 5.3. Tabulka uvádí úspšné strategie, které studenti použili, aby porozuměli danému

konceptu a byli ho schopni používat v programových úlohách.

5.2.3 Koncepty Objekt a Instance objektu

Objekt je základní stavební entitou v objektově orientovaného programování. Modelovaná realita do programového prostředí je tvořena jednotlivými prvky, které jsou označovány pojmem objekt. V objektu jsou obsažena data a jejich související funkčnost modelované reality. Data jsou označována jako atributy nebo vlastnosti, jejich funkčnost označujeme jako operace nebo metody.

Instance je objekt, který již má alokovaný prostor v paměti. Instance se vytváří pomocí zavolání konstruktoru třídy. Objekt, který již má vytvořenou instanci, můžeme používat běžným způsobem a přistupovat ke všem členům, ke kterým máme při jejich deklaraci povolen přístup.

Protože i v mnohých ostatních publikacích se striktně nerozlišuje mezi objektem a instancí objektu a z důvodu nejednoznačně vyhraněného konceptu ve sledované dokumentaci probandů, budeme nadále tento koncept uvažovat jednotně pod označením Objekt.

Jaké problémy koncepty Objekt a Instance objektu řeší

Jak již bylo uvedeno, celé paradigma objektově orientovaného programování se opírá o koncepty objekt a třída.

Používají se k rozdělení počítačového programu do jednoduchých a znovu použitelných částí (tříd), ze kterých se vytváří objekty a jejich konkrétní instance.

Pomocí objektů můžeme modelovat složité situace jako jednoduché a reprodukovatelné struktury. Objekty je možné použít i v ostatních částech našeho programu nebo též v cizích programech.

S použitím objektů jsou aplikace přehlednější, neboť části objekty obsahují všechny příslušné informace, a tím se také snadněji ladí případné chyby v kódu. Mezi další nesporné výhody patří možnost využívat další koncepty OOP (např.

zapouzdření, kompozice), které z použití tohoto programovacího paradigmatu vycházejí.

Přiřazení charakteristik a přístup studentů konceptů Objekt a Instance objektu

Tento koncept nezahrnuje jen jednoduché pojmy a operace, ale též složitější kombinování členů a využívá pro následnou aplikaci také např. seznamy. Proto jej dle taxonomie SOLO řadíme do úrovně *multistrukturální*.

V průběhu našeho výzkumu jsme žádné zásadní obtíže s tímto konceptem nezaznamenali. Studenti jsou schopni objekt navrhnout a používat.

> *„ M: Nejdřív bych se dohodnul, máme tři hrdiny*
>
> *J: Máme definitivně dané zadání. Jedná se o objektový návrh, potřebujeme nějaké objekty.*
>
> *Co máme za objekty? Aspoň ze začátku…*
>
> *M: Uděláme postavičku a zbraň.*
>
> *J: Jo! Minimálně*
>
> *M: Já to rovnou budu dělat, budeme psát v češtině?*
>
> *J: Ano, můžeš*
>
> *M: Myslíš ne jako konkrétně?*
>
> *J: Zatím dejme tomu nějakou postavu…*
>
> *M: Jasan*
>
> *J: Minimálně potřebujeme dva objekty, protože řešíme zbraň*
>
> *M: A postavu… To takhle překopíruji rovnou “*

Koncept Objekt se nám z vyjádření studentů nejevil jako problematický. To souviselo s předchozím konceptem. Jakmile byli studenti schopni pochopit koncept Třída jako model určitého reálného světa, pak velmi rychle s tím chápali

i koncept Objekt. Porozuměli tomu, že Třída je určitý předpis a Objekt je jen jeho konkrétní obraz. Proto jsme konceptu Objekt nepřiřadili charakteristiku problematický.

U získaných dokumentací od studentů je patrné, že tyto úlohy, které odpovídají dle taxonomie SOLO *multistruturální* úrovni, jsou studenti rovněž schopni zadané problémy vyřešit (viz např. ukázka výstupu práce probanda L1 na obr. 5.1 a výpisu 5.1).

Tedy konceptu Objekt jsme přiřadili charakteristiky

- nezvratný,
- integrativní,
- diskurzivní.

Přiřazené charakteristiky jsou podobné těm, které jsme přiřadili konceptu Třída. Opět jsme nepřiřadili charakteristiku problematický, protože studenti jej po krátké době opět byli schopni zvládnout a následně používat ve svých úlohách, jak je demonstrováno probandem M2:

> *„ Docela mě to všechno zapadlo, třída objekt, a teď to všechno používám.*
>
> *Někdy se mi to zplete, ale dobrý, jsem rád, že třídu a objekt chápu, zas tak těžký to nebylo. Teď to tam vždycky nacpu, no snad ke spokojenosti. “*

Použití odborných výrazů, slengů, jeho rozšíření a vylepšení, charakteristika *diskurzivní*, je vidět např. u probanda M1:

> *„Začnu v class diagramu, kde si vytvořím třídu člen rodiny, pak si vytvořím šatník a do toho člena rodiny přidáme vlastnost a to bude teda jméno (string) a přidáme tam potom ještě KusObleceni (jako vlastnost z toho co si teda vzal), a to co teda bude moci ten člen rodiny udělat, bude*

to, že si bude moci vzít oblečení a bude tam asi třeba potřeba metoda ToString a to je teda asi zatím všechno.

Do šatníku přidáme metodu – VytvořitŠatník –, my chceme, aby tam ty členové mohli vracet ty věci, takže tam budeme mít metodu – VraťKusOblečení –, potom budeme chtít metodu pro vypsání obsahu šatníku. Jméno šatníku, bude Šatník, což zní divně, ale nechala bych to. Potom tam dáme umsítění, obojí bude string a musíme tam přidat pole, protože do toho pole budeme potom ukládat kusy oblečení, co jsou v šatníku.

Potom teda musíme ještě vytvořit jednu třídu a to budou ty kusy oblečení, co tam budeme přidávat, a to bude teda třída Oblečení, tam bude jméno, a metoda ToString.

Takže, jsem zpátky, teď jsem řešila něco jiného, takže jsem měla pauzu, ale pokračujeme teda, jsme v class diagramu,

To, co musíme přidat, o čem jsem mluvila teda předtím, tak teda musíme do šatníku přidat ToString. Na co jsem předtím zapomněla, když na to teď koukám, tak to ještě přidat všem třídám konstruktor. A půjdu do kódu.

Začnu s ČlenRodiny, s tím že konstruktor bude mít teda v parametrech jméno a jméno se rovná Jméno, tím určíme, co bude jméno při vytváření, upravíme metodu get; set; a to samé uděláme teda s KusOblečení, jenom to nebude mít v parametru.

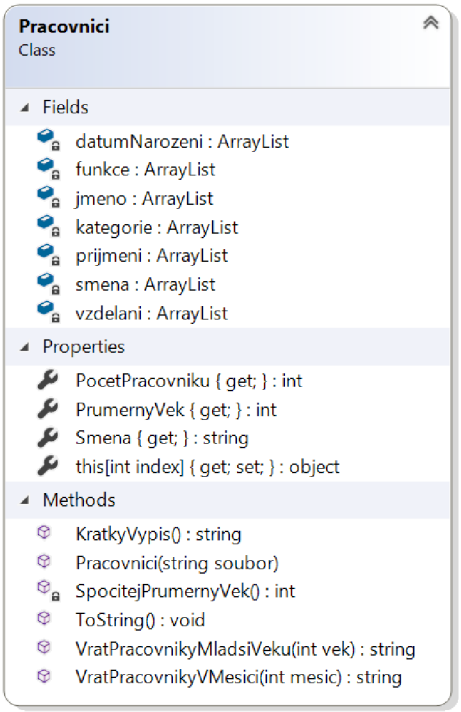
Mě to teď teda nefungovalo, tak jsem nakoukla do kódu, co jsme dělali na hodině, a zjistila jsem, že KusOblečení je ekvivaletní k objektu Potravina. Tak ten jsem si musela vytvořit. Takže to pojmenuju jako _kusobleceni. Ale stejně mi to nefunguje, protože... (tady se studentka na to dlouho rozmýšlela, koukala na diagram a zasekla se) my teda nechceme, aby byl kusObleceni, jako _kusobleceni ale chceme, aby to bylo jako Oblečení. Pořád to nefunguje, stejně. Tam možná musím ještě něco při-

dat do oblečení. Jo, už jsem pochopila, proč mi to nefungovalo. My totiž musíme nastavit v class diagramu, že kusObleceni bude datového typu Obleceni. Ale stejně to nefunguje.

Ještě jsem měla jeden problém, přiznám se, že jsem nakoukla do kódu, ale nakonec jsem zjistila, že mi jen chyběl vytvořit konstruktor u oblečení, nicméně už to funguje, ještě když už jsem teda u toho oblečení, tak upravím metodu get; set; a udělám ToString (ten studentka okopírovala ze své minulé práce).

A ta metoda bude teda jednoduchá, tak, že to jenom vrátí jméno oblečení. A tím bychom měli mít třídu oblečení hotovou. "

Následující ukázka diagramu tříd (obr. 5.1) a výpisu kódu (výpis 5.1) probanda L1 prezentuje propojení konceptu Třída s ostatními rovněž již ovládnutými koncepty (charakteristika *integrativní* dle teorie TC).



Obrázek 5.1: Diagram tříd pro ovládnuté koncepty Objekt a Instance objektu

Zdroj: (proband L1)

Výpis 5.1: Výpis ukázky kódu pro ovládnuté koncepty Objekt a Instance objektu

```
class Program
{
    static void Main(string[] args)
    {
        DateTime pracovnik1DatumNarozeni = new DateTime(1999, 01, 01);
        Pracovnik pracovnik1 = new Pracovnik("Nový", "Jan", pracovnik1DatumNarozeni,
            "ŠS", "vedoucí"); ;
        Console.WriteLine(pracovnik1.ToString());
        DateTime pracovnik2DatumNarozeni = new DateTime(2000, 12, 24);
        Pracovnik pracovnik2 = new Pracovnik("Stará", "Jana", pracovnik2DatumNarozeni,
            "ŠS", "vedoucí");
        Console.WriteLine(pracovnik2.ToString());
        Console.WriteLine("ěMsíc narození: " + pracovnik2.VratMesicNarozeni());
    }
}

public class Pracovnik
{
    private string prijmeni;
    private string jmeno;
    private DateTime datumNarozeni;
    private string vzdelani;
    private string funkce;

    public Pracovnik(string prijmeni, string jmeno,
        DateTime datumNarozeni, string vzdelani, string funkce)
    {
        this.prijmeni = prijmeni;
        this.jmeno = jmeno;
        this.datumNarozeni = datumNarozeni;
        this.vzdelani = vzdelani;
        this.funkce = funkce;
    }

    public int Jmeno
    {
        get => default;
        set
        {
        }
    }

    public int Prijmeni
```

```
44      {
45          get => default;
46          set
47          {
48          }
49      }
50
51      public int DatumNarozeni
52      {
53          get => default;
54          set
55          {
56          }
57      }
58
59      public override string ToString()
60      {
61          return $"{prijmeni} {jmeno} {datumNarozeni.ToShortDateString()} {funkce}";
62      }
63
64      public int VratMesicNarozeni()
65      {
66          return datumNarozeni.Month;
67      }
68
69  }
```

Zdroj: (proband L1

Strategie studentů pro překonání problémů s koncepty Objekt a Instance objektu

Tabulka 5.4 ukazuje způsob, který studenti využili, pokud měli problém s daným konceptem při řešení programových úloh. Jak již jsme uvedli, OOP koncept studenti poměrně rychle zvládli díky pochopení jiného konceptu Třída. Koncept zvládli velmi rychle v rámci výuky, proto jen několik z nich pro úplné pochopení použilo určitou strategii uvedenou v tabulce 5.4. Studenti tedy zvládli tento koncept a byli schopni samostatně vytvářet třídy a objekty (viz Proband M1 v předchozí ukázce).

Tabulka 5.4: Přístup a strategie studentů pro překonání problému s koncepty Objekt a Instance objektu

Použitá strategie	Přístup	Procento studentů využívajících danou strategii
Učí se od ostatních lidí	Učí se od spolužáků	10,5
	Získá pomoc od učitele	5,3
Učí se z psaných materiálů	Čte instrukce z knihy	0
	Čte instrukce z internetu	0
Učí se z příkladů	Používá vlastní příklady	0
	Hledá příklady na internetu	5,3
Užití jiného přístupu	Pokus a omyl	0
	Vyhnutí se problému	0
	Udělá si přestávku a zkusí řešit po přestávce	0

Zdroj: autor

5.2.4 Koncept Kompozice

Kompozice je jedním ze základních konceptů v objektově orientovaném programování. Popisuje třídu, která odkazuje na jeden nebo více objektů jiných tříd v proměnných instance. To nám umožní modelovat asociaci *HAS-A* (kompozici) mezi objekty.

V reálném světě můžeme takové vztahy najít docela pravidelně. Například auto má motor, moderní kávovary mají často integrovaný mlýnek a varnou jednotku.

Jaké problémy koncept Kompozice řeší

Vzhledem k jeho širokému použití v reálném světě není překvapením, že se kompozice běžně používá i v pečlivě navržených softwarových komponentách. Když použijeme tento koncept, můžeme:

- znovu použít stávající kód
- navrhovat čistá API

- změnit implementaci třídy používané ve skladbě bez přizpůsobení jakýchkoli externích klientů

Znovu použít stávající kód Hlavním důvodem použití kompozice je to, že nám umožňuje znovu použít kód bez modelování přidružení *IS-A*, jak to děláme pomocí dědičnosti. To umožňuje silnější zapouzdření a usnadňuje údržbu našeho kódu, jak vysvětluje Bloch (2002).

Koncept kompozice se často používá v reálném světě a měl by být stejný i při vývoji softwaru. Auto má motor, kávovar má mlýnek a varnou jednotku (přidružení *HAS-A*), ale není to žádný z uvedených objektů (nejedná se o přidružení *IS-A*). Automobil a kávovar integrují motor, mlýnek a varnou jednotku prostřednictvím svých externích API, aby vytvořili vyšší úroveň abstrakce a poskytli svým uživatelům významnější hodnotu.

Totéž můžeme udělat při vývoji softwaru, když navrhujeme třídu, abychom zachovali odkaz na objekt a použili jej v jedné nebo více jeho metod.

Navrhněme čisté API To nám také umožňuje navrhovat čistá a snadno použitelná rozhraní API. Když vytváříme třídu, můžeme se rozhodnout, zda se odkazované třídy stanou součástí API, nebo zda je chceme skrýt.

Objektově orientovaný jazyk C♯ podporuje různé modifikátory přístupu. Obvyklým osvědčeným postupem je použít soukromý modifikátor pro všechny atributy, včetně těch, které odkazují na jiné objekty, takže k němu lze přistupovat pouze v rámci stejného objektu. Chceme-li povolit externí přístup k atributu, musíme pro něj implementovat metodu getter nebo setter, případně použít vlastnost.

Ale to není jediná věc, kterou můžeme udělat pro vytvoření čistého API. Pokud pro třídu nepoužíváme žádné modifikátory přístupu, stane se soukromou třídou. K této třídě nelze přistupovat mimo její jmenný prostor a není součástí API. Externí klienti našeho jmenného prostoru o této třídě nevědí. Mohou ji použít pouze prostřednictvím veřejné třídy.

Skrýt změny implementovaného interního kódu Použití kompozice a zapouzdření nám nejen umožňuje vytvářet lepší rozhraní API, ale můžeme jej také použít ke snazší údržbě a úpravám kódu. Dokud bude třída využívána pouze naším vlastním kódem, můžeme ji snadno změnit a v případě potřeby upravit libovolný kód klienta.

Přiřazení charakteristik a přístup studentů konceptu Kompozice

Na základě rozhovorů, vyjádření studentů, jejich reakcí při řešení úkolů a analýzy získané dokumentace přiřazujeme tomuto konceptu charakteristiky

- problematický,
- nezvratný,
- integrativní,
- diskurzivní,
- liminální.

Tento koncept vyžaduje zvládnutí pochopení základních pojmů, ale též schopnosti kombinování, pochopení souvislostí, porovnávání a aplikaci. Proto jej dle taxonomie SOLO řadíme do úrovně *relační*.

Častý příklad, který je následně popsán, nám demonstruje chybný způsob aplikace konceptu Kompozice. Tento příklad nám demonstruje charakteristiku podle teorie TC *liminální*, protože se u studenta projevuje zmatenost a nejistota při řešení zadaného problému. Tato byla u zmiňovaného studenta pozorována v delším časovém úseku a trvala do doby, kdy vyzkoušel strategii *učení se od ostatních lidí* (konkrétně přístup *konzultace s učitelem*), která mu pomohla při zvládnutí konceptu. To jsme pozorovali v rozhovorech a popisech při řešení úkolu, kde studenti měli za úkol použít při řešení koncept Kompozice.

„ Ty, jo, tak už už mi to docvaklo. Fakt mi to trvalo dlouho, vůbec jsem nevěděl co s tím. Ale pořádně mě to mátlo, protože mi to nějak unikalo a já, děkuji, že jste mi to vysvětlil. “ (proband V2)

Ukázka práce ve dvojicích (P1 i M1 – probandi) nám rovněž představuje důvod zvolení charakteristiky konceptu podle teorie TC jako *problémový*.

> „ *P: Takže ja som nad tým premýšlal a našou úlohou, mali sme teda vytvoriť postavu, ktorá je schopná si zobrať zbraň a dať ju spať? M:Vzít tu zbraň dát ji dolů a má mít jméno, chceme víc zbraní a víc postav. P: Ja by som teda začal klasicky s class diagramom a, vlastne my sme sa o tom předtím rozprávali a ja by som teda vytvoril dve triedy, takže tam dávam tie dve triedy.* “ (proband V2)

Charakteristiky *integrativní* a *diskurzivní* jsme přiřadili tomuto konceptu opět na základě rozhovorů se studenty. Z rozhovorů bylo zřejmé, že si studenti uvědomují, že OOP koncept kompozice integruje předchozí koncepty a bylo též vidět, že si dále rozšiřují svůj odborný jazyk jazyk používaný v rámci objektově orientovaného přístupu.

Strategie studentů pro překonání problémů s konceptem Kompozice

Tabulka 5.5 ukazuje způsob, který studenti využili, pokud měli problém s daným konceptem při řešení programových úloh. Koncept kompozice studentům činil časté problémy, bylo nutné věnovat studentům dostatek času, aby tento koncept zvládli. Několik z nich pro úplné pochopení použilo i několik ze strategií, které jsou uvedeny v tabulce 5.5. Studenti tedy zvládli tento koncept a byli schopni samostatně vytvářet třídy a objekty (viz proband M1 v předchozí ukázce).

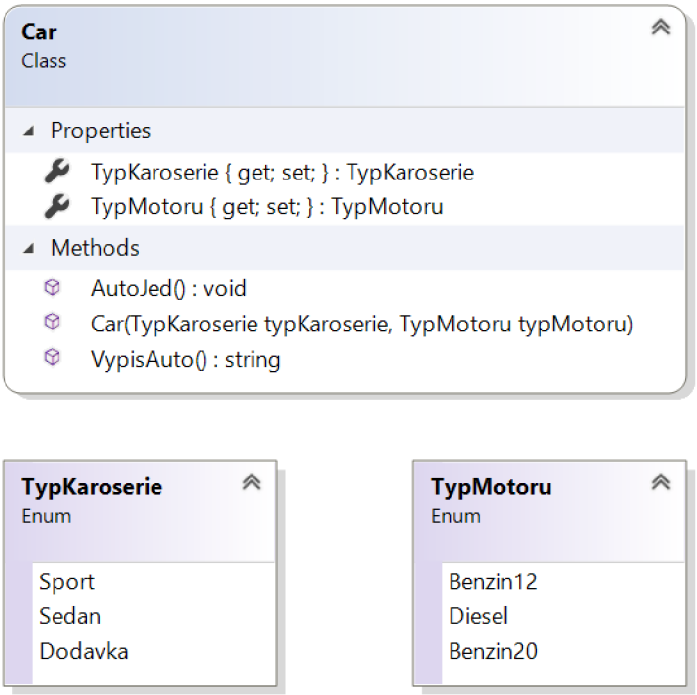
Na následujícím přepisu proband M2 používá kombinaci přístupů *pokus-omyl* a *učení se od spolužáků* (kombinace strategií *užití jiného přístupu* a *učí se od ostatních lidí*).

> „ *Nejdřív jsem si to zkusil, vyzkoušel jsem to sám, myslel jsem, že to tam dokážu poskládat, ale nějak to nešlo a házelo mi to chybu. Tak jsem se zeptal proband J1, a ten mi to vysvětlil a pomohl mi najít chybu. Pak*

Tabulka 5.5: Přístup a strategie studentů pro překonání problému s konceptem Kompozice

Použitá strategie	Přístup	Procento studentů využívajících danou strategii
Učí se od ostatních lidí	Učí se od spolužáků	26,3
	Získá pomoc od učitele	31,6
Učí se z psaných materiálů	Čte instrukce z knihy	0
	Čte instrukce z internetu	10,5
Učí se z příkladů	Používá vlastní příklady	5,3
	Hledá příklady na internetu	36,8
Užití jiného přístupu	Pokus a omyl	10,5
	Vyhnutí se problému	31,6
	Udělá si přestávku a zkusí řešit po přestávce	0

Zdroj: autor



Obrázek 5.2: Příklad diagramu tříd pro řešení úkolu pro koncept Kompozice s využitím výčtového typu Enum

Zdroj: (proband P3)

jsem to zkusil vyřešit sám a fungovalo mi to, takže jsem rád, že už tomu konečně rozumím. “

Častými případy, kdy studenti se snažili obejít problematiku konceptu Kompozice, byly situace s nahrazením správného řešení, nejčastěji výčtem (obr. 5.2 a výpis 5.2). Uplatnili tedy přístup *vyhnutí se problému* strategie *užití jiného přístupu*. Takovéto řešení je pro další využití programu samozřejmě naprosto nevyhovující, neboť nám neumožňuje uchovat pro jednotlivé hodnoty výčtu další atributy a aplikovat další metody.

Výpis 5.2: Příklad zdrojového kódu řešení úkolu pro koncept Kompozice s využitím výčtového typu Enum

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace car_factory
{
    class Program
    {
        static void Main(string[] args)
        {
           carFactory car1 = new carFactory(carName.Tesla, engineTypes.engineA);
           car1.Print();
        }
    }

    public class carFactory
    {
        private engineTypes _engineTypes;
        private carName _carName;

        public carFactory(carName _carname, engineTypes _enginetypes)
        {
            _engineTypes = _enginetypes;
            _carName = _carname;

        }

```

```
30          public void Print()
31          {
32              Console.WriteLine("{0} has this type of engine: {1}",_carName, _engineTypes);
33          }
34      }
35
36      public enum engineTypes
37      {
38          engineA,
39          engineB,
40          engineC
41      }
42
43      public enum carName
44      {Š
45          koda,
46          Audi,
47          Tesla
48      }
49  }
```

Zdroj: (proband P3

Následné zpracování již odporovalo využití kompozičního konceptu. V jiném úkolu měli studenti vyřešit možnost pracovat s více zbraněmi, což probandi nevhodně vyřešili pomocí kolekce (listu) (M1 – proband), použili přístup *vyhnutí se problému*.

„M: Dej tam teda trídu character a weapon a ja by som do toho weapon dala nejdřív jména jako vlastnost a asi bych potom dala weapon na list, že sa bude vyberať .. “

Zde je patrno, že pouze studenti zkouší, co a jak. Snaha vytvořit třídu pro výběr zbraně je v rozporu zejména s principem otevřenosti-uzavřenosti. Součástí jejich třídy chtěli mít datový člen zbraň typu enum, ze kterého chtěli vybírat požadovanou zbraň (P1 – proband).

„P1: Ešte mi napadlo, že by sme mohli urobiť weaponlist, to bude class, enum, a ten by sme mohli využívať při výberu zbrane. “

Pro doplnění uvádíme deklaraci použitého typu enum pro výběr zbraně (výpis 5.3).

Výpis 5.3: Deklarace výčtového typu enum pro výběr zbraně

```
public enum weapon_list
{
    sword,
    knife,
    gun,
    stick
}
```

Zdroj: (proband P1

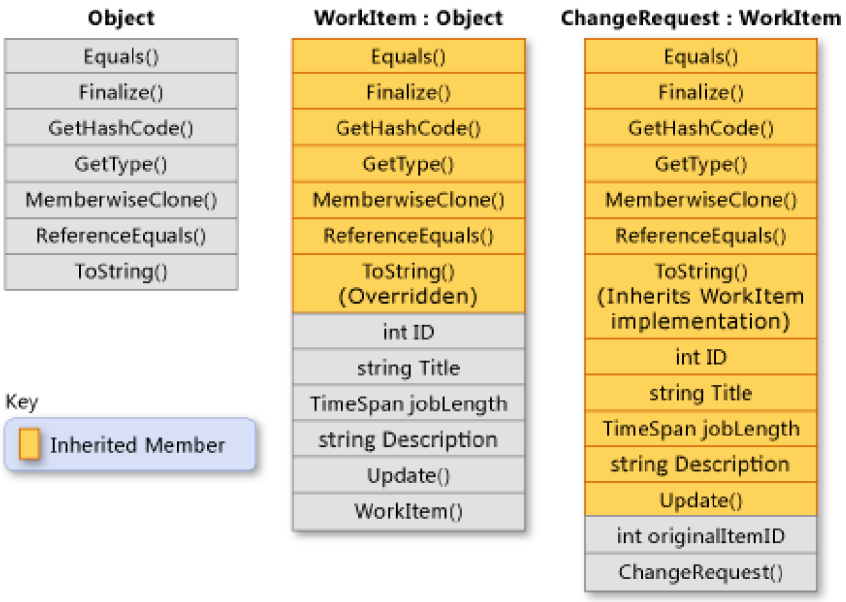
Strategie, kterou probandi použili, odpovídá přístupu *trasování kódu*. Vzápětí, po neúspěšných pokusech, přechází na další přístup *pokus-omyl*.

> *„P1: Skúsim to spustit... Jo to vypouští charakter jak toje možné? Tak můžeme si to prejsť tým krokovaním, takže, teraz to nastavuje, name je adam, teraz name je null... Tak dobre, character name je prázdne, skúsme zadať opačne. Jak to máš tam vlastne? “*

5.2.5 Koncept Dědičnost

Dědičnost umožňuje vytvořit nové třídy, které znovu používají, rozšiřují a mění chování definované v ostatních třídách. Třída, jejíž členové jsou zděděni, se nazývá základní třídou a třída, která dědí tyto členy, se nazývá odvozená třída. Odvozená třída může mít pouze jednu přímou základní třídu. Dědičnost je však tranzitivní. Pokud třída ClassC je odvozena ze třídy ClassB, třída ClassB je odvozena ze třídy ClassA , tak třída ClassC dědí (viditelné) členy deklarované ve třídách ClassB a ClassA (viz obr. 5.3).

V obecném případě je odvozená třída specializací základní třídy. Při definování třídy pro odvození z jiné třídy odvozená třída implicitně získá všechny členy základní třídy, s výjimkou jejích konstruktorů a finalizační metody. Odvozená třída znovu používá kód v základní třídě, aniž by bylo nutné ho znovu implementovat. V odvozené třídě lze přidat další členy, odvozená třída tak rozšiřuje funkčnost třídy základní.



Obrázek 5.3: Příklad dědičnosti třídy WorkItem

Zdroj: (Microsoft, 2021b)

Jaké problémy koncept Dědičnost řeší

Dědičnost je jedna ze základních vlastností OOP a slouží k tvoření nových datových struktur na základě již existujících.

Přiřazení charakteristik a přístup studentů konceptu Dědičnost

Na základě rozhovorů, vyjádření studentů, jejich reakcí při řešení úkolů a analýzy získané dokumentace přiřazujeme tomuto konceptu charakteristiky:

- integrativní,
- ohraničený,
- diskurzivní.

Konceptu jsme přiřadili tyto charakteristiky, *integrativní* a *diskurzivní*:

Probant M2:

„ Najednou se mi to dalo dohromady, pochopil jsem, proč jsme se učili třídu a objekt, dědičnost mi moc problémů nedělala s pochopením, je to něco co bych řekl, že je přirozené. “ (proband M2)

V průběhu našeho výzkumu jsme žádné zásadní obtíže s tímto konceptem nezaznamenali. Studenti jsou schopni aplikovat vztahy dědičnosti ve svých programech, probandi zadané úkoly zdárně vyřešili.

„ M1: Teď musíme přidat dědičnost třídy dědičnost k beverage a dědičnost třídy milk k condimentdecorator a caramel bude taky dědit od condimentdecorator a condimentdecorator musí dědit od beverage. “ (proband M1)

Strategie studentů pro překonání problémů s konceptem Dědičnost

Tabulka 5.6 ukazuje způsoby přístupů, které studenti využili, pokud měli problém s daným konceptem při řešení programových úloh. Jak již jsme uvedli, koncept dědičnosti studentům nečinil závažnější problémy a poměrně rychle jej zvládli. Koncept zvládli velmi rychle v rámci výuky, proto jen několik z nich pro úplné pochopení použilo určitou strategii uvedenou v tabulce 5.6. Studenti tedy zvládli tento koncept a byli schopni samostatně vytvářet třídy a objekty (viz Proband M1 v předchozí ukázce).

Tabulka 5.6: Přístup a strategie studentů pro překonání problému s konceptem Dědičnost

Použitá strategie	Přístup	Procento studentů využívajících danou strategii
Učí se od ostatních lidí	Učí se od spolužáků	21,1
	Získá pomoc od učitele	5,3
Učí se z psaných materiálů	Čte instrukce z knihy	5,3
	Čte instrukce z internetu	21,1
Učí se z příkladů	Používá vlastní příklady	10,5
	Hledá příklady na internetu	21,1
Užití jiného přístupu	Pokus a omyl	5,3
	Vyhnutí se problému	0
	Udělá si přestávku a zkusí řešit po přestávce	0

Zdroj: autor

Problémy, které probandi měli s konceptem Dědičnost, velmi rychle zvládli,

např. proband J1 (V – výzkumník) použil za začátku místo vztahu dědičnosti asociační vztah.

„ *J1: No tady jsem to špatně udělal.*

V: Co jste špatně udělal?

J1: Špatně jsem to nadefinoval tu dědičnost v class diagramu.

V: Co jste vlastně nadefinoval v tom class diagramu?

J1: Že třída zbraně má jako vlastnost sekeru a meč.

V: To jste nechtěl?

J1: To jsem nechtěl. “

Velmi rychle ale přišel na chybu za pomoci učitele (strategie *učí se od ostatních lidí*).

V dalším případě byla situace velice obdobná. Probandi pracovali ve dvojicích, takže na chybný krok velice rychle přišli (strategie *učí se od spolužáků*)

„ *M2: Ale potřebuješ zbraň, protože každá zbraň má svůj způsob boje*

J2: Každá zbraň má své způsoby boje

M2: Takže udělám další objekty... potomky

J2: Který budou dědit

M2: Dohodneme se na třech zbraních

J2: Nutně by asi dědit nemusely, ale ... jo, celkově by mohly mít nějakou vlastnost, třeba jméno

J2: Stačí asi dvě

M2: Dáme tři

J2: Dědičnost, né asociace “

5.2.6 Koncept Polymorfismus

Polymorfismus je řeckého původu, znamená „mnoho tvarů“, v češtině lze tedy přeložit jako mnohotvárnost. Má dva odlišné aspekty:

- V době běhu lze objekty odvozené třídy považovat za objekty základní třídy v místech, jako jsou parametry metody a kolekce nebo pole. Pokud tato polymorfismua nastane, deklarovaný typ objektu již není totožný s jeho typem za běhu.
- Základní třídy mohou definovat a implementovat virtuální metody a odvozené třídy je mohou přepsat, což znamená, že poskytují svou vlastní definici a implementaci. V době běhu, když kód klienta volá metodu, modul kompilátoru (CLR) vyhledá typ za běhu objektu a vyvolá přepsání virtuální metody. Ve zdrojovém kódu můžeme zavolat metodu pro základní třídu a tím provést spuštění metody z odvozené třídy.

Virtuální metody umožňují pracovat se skupinami souvisejících objektů jednotným způsobem. Předpokládejme například, že máme aplikaci pro kreslení, která umožňuje uživateli vytvářet různé druhy tvarů na kreslicí ploše. V době kompilace neznáme, které konkrétní typy tvarů uživatel vytvoří. Aplikace však musí sledovat všechny různé typy tvarů, které byly vytvořeny, a musí je aktualizovat v reakci na akce myši uživatele. K vyřešení tohoto problému můžeme použít polymorfismus ve dvou základních krocích:

1. Vytvoříme hierarchii třídy, ve které jsou konkrétní třídy pro jednotlivé tvary odvozeny ze společné základní třídy.
2. Použijeme virtuální metodu k vyvolání vhodné metody pro jakoukoli odvozenou třídu prostřednictvím jediného volání metody základní třídy.

Následující zdrojový kód (výpis 5.4) demonstruje použití konceptu polymorfismu s využitím virtuálních metod.

Výpis 5.4: Příklad zdrojového kódu s virtuálními metodami pro koncept Polymorfismus

```
public class Shape
{
    public int X { get; private set; }
    public int Y { get; private set; }
    public int Height { get; set; }
    public int Width { get; set; }

    public virtual void Draw()
    {
        Console.WriteLine("Performing base class drawing tasks");
    }
}

public class Circle : Shape
{
    public override void Draw()
    {
        Console.WriteLine("Drawing a circle");
        base.Draw();
    }
}
public class Rectangle : Shape
{
    public override void Draw()
    {
        Console.WriteLine("Drawing a rectangle");
        base.Draw();
    }
}
public class Triangle : Shape
{
    public override void Draw()
    {
        Console.WriteLine("Drawing a triangle");
        base.Draw();
    }
}
```

Zdroj: (proband P3

Jaké problémy koncept Polymorfismus řeší

Polymorfismus vyplývá ze skutečnosti, že každá třída žije ve svém vlastním jmenném prostoru. Jména přiřazená v definici třídy nejsou v rozporu s názvy přiřazenými kdekoli mimo ni. To platí jak pro proměnné instance v datové struktuře objektu, tak pro metody objektu.

Proměnné instance objektu jsou v chráněném oboru názvů. Názvy metod jsou také chráněny. Na rozdíl od názvů funkcí jazyka C nejsou názvy metod globálními symboly. Název metody v jedné třídě nemůže být v rozporu s názvy metod v jiných třídách; dvě velmi odlišné třídy mohou implementovat identicky pojmenované metody.

Názvy metod jsou součástí rozhraní objektu. Když je odeslána zpráva požadující, aby nějaký objekt něco udělal, zpráva pojmenuje metodu, kterou by měl objekt provádět. Protože různé objekty mohou mít metody se stejným názvem, je třeba chápat význam zprávy ve vztahu ke konkrétnímu objektu, který zprávu přijímá. Stejná zpráva odeslaná dvěma různým objektům může vyvolat dvě odlišné metody.

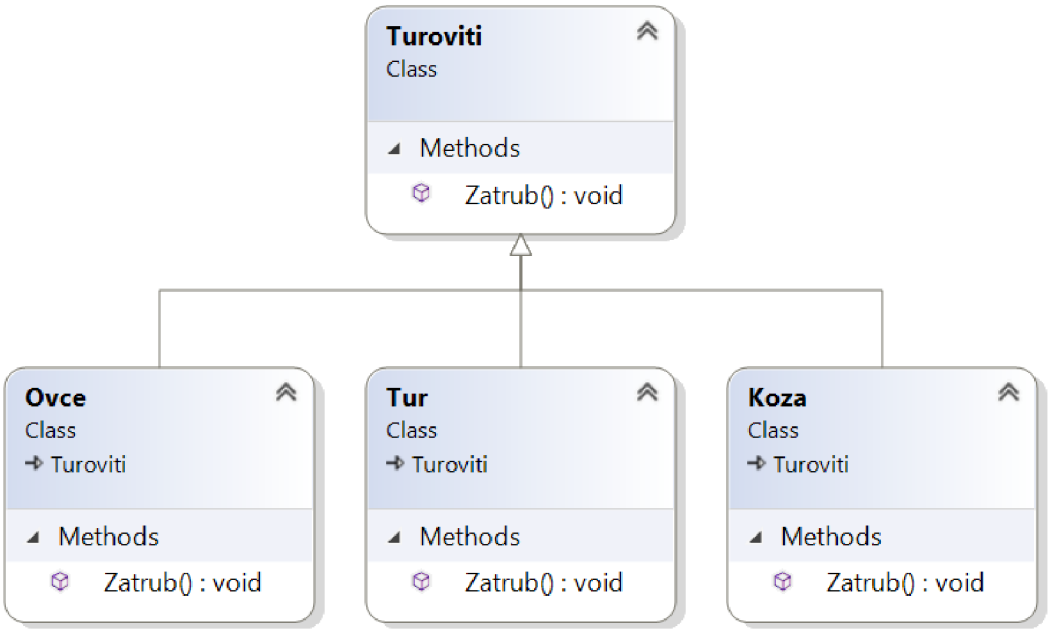
Hlavní výhodou polymorfismu je, že zjednodušuje programovací rozhraní. Umožňuje vytvořit konvence, které lze znovu použít v dceřinné třídě. Virtuální metody a vlastnosti umožňují odvozeným třídám rozšířit základní třídu bez nutnosti použití implementace metody v základní třídě.

Přiřazení charakteristik a přístup studentů konceptu Polymorfismus

Na základě rozhovorů, vyjádření studentů, jejich reakcí při řešení úkolů a analýzy získané dokumentace přiřazujeme tomuto konceptu charakteristiky

- problematický,
- integrativní,
- diskurzivní.

Z rozhovorů se studenti jsme zjistili, že někteří studenti byli zprvu přesvědčeni o svém brzkém překonání problémů s konceptem, avšak analýza jejich dokumentace tomuto stavu nenasvědčovala. Dále je demonstrováno na ukázce probanda P2 (obr. 5.4 a výpis 5.5). Konceptu polymorfismus jsme proto přiřadili charakteristiku *problematický*.



Obrázek 5.4: Příklad diagramu tříd pro řešení úkolu pro nepochopený koncept Polymorfismus

Zdroj: (proband P2)

Výpis 5.5: Příklad zdrojového kódu řešení úkolu pro nepochopený koncept Polymorfismus

```
1  class Program
2  {
3      static void Main(string[] args)
4      {
5          Turoviti turoviti = new Turoviti();
6          turoviti.Zatrub();
7          Tur tur = new Tur();
8          tur.Zatrub();
9          Ovce ovce = new Ovce();
10         ovce.Zatrub();
11         Koza koza = new Koza();
12         koza.Zatrub();
```

```
13      }
14  }
15
16  public class Turoviti
17  {
18      public virtual void Zatrub()
19      {
20          Console.WriteLine("ůůůůůůůůůůůT");
21      }
22  }
23
24  public class Tur : Turoviti
25  {
26      public override void Zatrub()
27      {
28          Console.WriteLine("ůůůůůůůůůůůB");
29      }
30  }
31
32  public class Ovce : Turoviti
33  {
34      public override void Zatrub()
35      {
36          Console.WriteLine("Béééééééééé");
37      }
38  }
39
40  public class Koza : Turoviti
41  {
42      public override void Zatrub()
43      {
44          Console.WriteLine("Méééééééééé");
45      }
46  }
```

Zdroj: (proband P2

Tento koncept vyžaduje zvládnutí pochopení základních pojmů, ale též schopnosti porovnávání, anlýzy, pochopení souvislostí, kombinování a aplikaci. Proto jej dle taxonomie SOLO řadíme do úrovně *relační*.

Podobně, jako u předchozích OOP konceptů, lze na základě rozhovorů se studentu konstatovat, že tento koncept integruje další OOP koncepty, studenti musí

znát a kombinovat předchozí postupy a pojmy (je to vidět i na návrhu, obr. 5.4) a zároveň se rozšiřuje i odborná slovní zásoba studentů, tedy tomuto konceptu jsme přiřadili další charakteristiky *integrativní* a *diskurzivní*.

Strategie studentů pro překonání problémů s konceptem Polymorfismus

Tabulka 5.7 ukazuje způsob, který studenti využili, pokud měli problém s daným konceptem při řešení programových úloh. Studenti s konceptem měli značné obtíže, pro úplné pochopení studenti použili i několik strategií uvedených v tabulce 5.7.

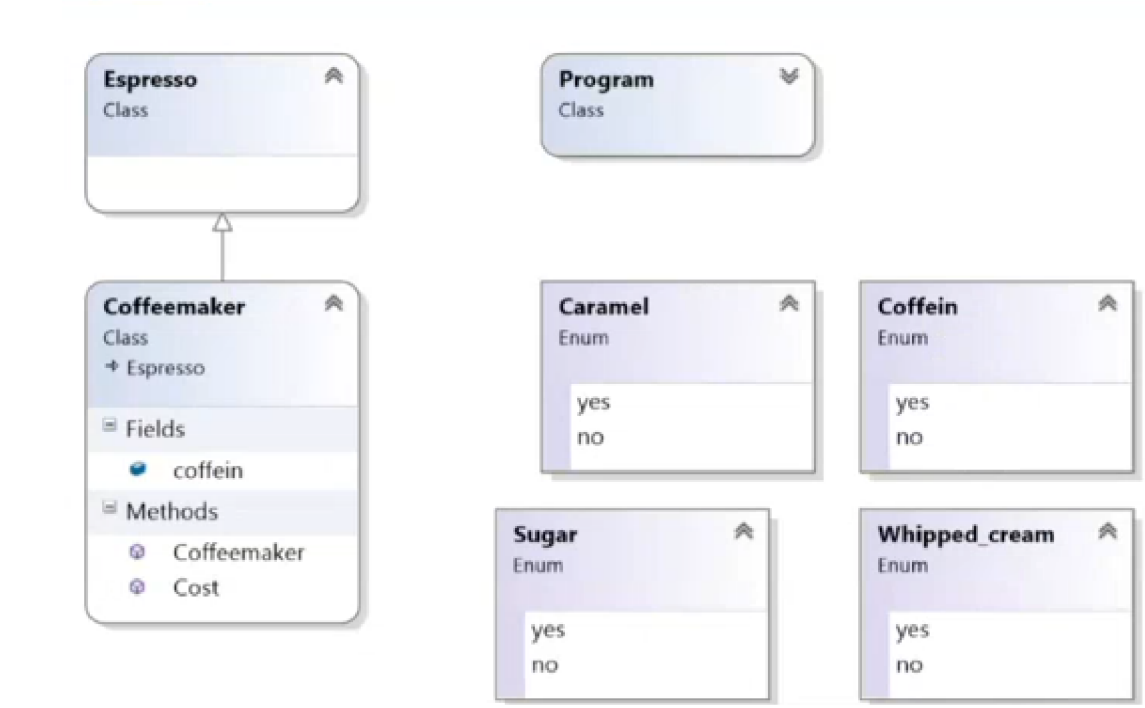
Tabulka 5.7: Přístup a strategie studentů pro překonání problému s konceptem Polymorfismus

Použitá strategie	Přístup	Procento studentů využívajících danou strategii
Učí se od ostatních lidí	Učí se od spolužáků	21,1
	Získá pomoc od učitele	26,3
Učí se z psaných materiálů	Čte instrukce z knihy	21,1
	Čte instrukce z internetu	10,5
Učí se z příkladů	Používá vlastní příklady	21,1
	Hledá příklady na internetu	26,3
Užití jiného přístupu	Pokus a omyl	10,5
	Vyhnutí se problému	10,5
	Udělá si přestávku a zkusí řešit po přestávce	0

Zdroj: autor

Při řešení úloh, ve kterých má student použít koncept Polymorfismus, se někdy objevuje nápad uchovávat rozličné hodnoty v jednom seznamu. Problém tohoto řešení je, že studenti volí pro prvek seznamu typ string nebo některý z primitivních typů. Pokud potřebujeme uchovávat seznam, volíme pro typ prvku některou z bázových tříd objektu, který popisuje rozličné hodnoty, resp. situace. Jedná se o přístup *vyhnutí se problému*, studenti při řešení nepoužijí daný koncept. Vymyslí jiné řešení, které ale není správné, např. v první verzi programu

Kavárna deklarovala probandka M1 typ enum pro všechny přísady, vždy stejným způsobem, s hodnotami *yes* a *no* (viz obr. 5.5). Následující výpis 5.6 zobrazuje použitý kód pro určení ceny nápoje dle zvolené přísady.



Obrázek 5.5: Příklad diagramu tříd pro řešení úkolu pro nepochopený koncept Polymorfismus

Zdroj: (proband P2)

Výpis 5.6: Příklad zdrojového kódu řešení úkolu pro nepochopený koncept Polymorfismus

```
1  class Program
2  {
3      static void Main(string[] args)
4      {
5          CofeeShop coffeShop = new coffeShop();
6          int price = coffeShop.Cost(Coffein.yes, Whipped_cream.no, Sugar.no, Caramel.yes);
7          Console.WriteLine(price);
8      }
9  }
10
11 public class Espresso
12 {
13     public Espresso()
14     {
```

```
    }
}

public class Coffemaker : Espresso
{
    private int coffein;

    public Cofeemaker()
    {
    }

    public int Cost(Coffein caffein, Whipped_cream cream, Sugar sugar, Caramel caramel)
    {
        int cost = 0;
        if (caffein == Coffein.yes)
        {
            cost = cost + 15;
        }
        if (cream == Whipped_cream.yes)
        {
            cost = cost + 5;
        }
        if (sugar == Sugar.yes)
        {
            cost = cost + 2;
        }
        if (caramel == Caramel.yes)
        {
            cost = cost + 20;
        }
        return cost;
    }
}

public enum Coffein
{
    yes,
    no
}

public enum Whipped_cream
{
    yes,
    no
```

```
59 }
60
61 public enum Sugar
62 {
63     yes,
64     no
65 }
66
67 public enum Caramel
68 {
69     yes,
70     no
71 }
```

Zdroj: (proband P2

5.2.7 Koncept Zapouzdření

Zapouzdření umožňuje programátorovi implementovat požadovanou úroveň abstrakce. Jedná se o proces skrývání všech vnitřních detailů objektu před vnějším světem. Skrýváme data, která je činí soukromými, a vystavujeme veřejné vlastnictví pro přístup k těmto datům z vnější.

Jaké problémy koncept Zapouzdření řeší

Řeší dva problémy, které sužovaly rané programování:

- neregulovaný sdílený přístup k datům,
- obtíže při změně kódu, když vztahy mezi částmi nejsou dobře strukturované.

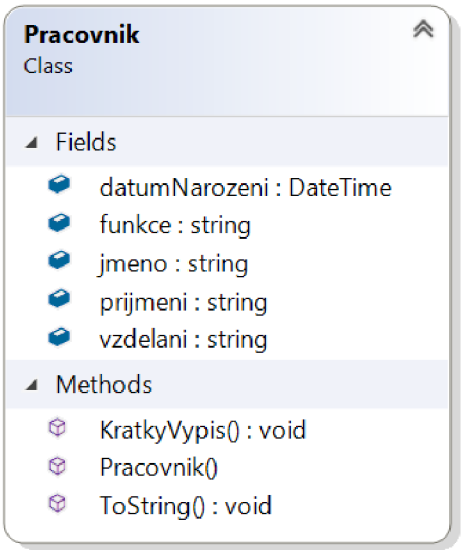
Zapouzdření chrání data před nežádoucím přístupem nebo změnami. Je to mechanismus, kde je implementována abstrakce. Řeší problém na úrovni implementace.

Zapouzdřená třída nám místo toho neumožňuje přímý přístup k datovým polím, k přístupu k funkcím pro čtení a zápis dat na základě požadavků musíme použít getter a setter (nebo vlastnosti).

Přiřazení charakteristik a přístup studentů konceptu Zapouzdření

Na základě rozhovorů, vyjádření studentů, jejich reakcí při řešení úkolů a analýzy získané dokumentace přiřazujeme tomuto konceptu charakteristiky

- nezvratný,
- integrativní,
- ohraničený,
- diskurzivní.



Obrázek 5.6: Příklad diagramu tříd pro řešení úkolu pro nepochopený koncept Zapouzdreni

Zdroj: (proband L2)

Častým řešením studentů, jak zajistit přístup k datům svého objektu, před ovládnutím konceptu Zapouzdření, bylo použitím veřejných deklarací datových polí (atributů). Tato řešení se sice objevovala zpočátku velice často, ale poměrně brzy studenti koncept ovládli a tuto chybu již neopakovali. Proto také tomuto konceptu přisuzujeme charakteriky *nezvratný* a *diskurzivní*. Ukázka použití chybného přístupu k atributům je demonstrována na diagramu tříd a výpisu k tomuto kódu probanda L2 (obr. 5.6 a výpis 5.7).

Výpis 5.7: Příklad zdrojového kódu řešení úkolu pro nepochopený koncept Zapouzdření

```
class Program
{
    static void Main(string[] args)
    {
        Pracovnik pracovnik = new Pracovnik();
        pracovnik.jmeno = "Adam";
        pracovnik.vzdelani = "ŠV";
        pracovnik.prijmeni = "Zelený";
        pracovnik.funkce = "CEO";
        pracovnik.KratkyVypis();
    }
}

public class Pracovnik
{
    public string prijmeni;
    public string jmeno;
    public DateTime datumNarozeni;
    public string vzdelani;
    public string funkce;

    public Pracovnik()
    {
    }

    public void KratkyVypis()
    {
        Console.WriteLine(prijmeni +" "+ jmeno + " " + funkce + "/n");
    }

    public void ToString()
    {
        throw new System.NotImplementedException();
    }
}
```

Zdroj: (proband L2)

Strategie studentů pro překonání problémů s konceptem Zapouzdření

Tabulka 5.8 ukazuje způsob, který studenti využili, pokud měli problém s daným konceptem při řešení programových úloh. Studenti s konceptem neměli značné

obtíže, nicméně, i zde pro úplné pochopení studenti použili i několik strategií uvedených v tab. 5.8.

Tabulka 5.8: Přístup a strategie studentů pro překonání problému s konceptem Zapouzdření

Použitá strategie	Přístup	Procento studentů využívajících danou strategii
Učí se od ostatních lidí	Učí se od spolužáků	31,6
	Získá pomoc od učitele	10,5
Učí se z psaných materiálů	Čte instrukce z knihy	5,3
	Čte instrukce z internetu	21,1
Učí se z příkladů	Používá vlastní příklady	10,5
	Hledá příklady na internetu	21,1
Užití jiného přístupu	Pokus a omyl	15,8
	Vyhnutí se problému	0
	Udělá si přestávku a zkusí řešit po přestávce	0

Zdroj: autor

Proband L2 o příkladu k tomuto konceptu (obr. 5.6 a výpis 5.7) později při rozhovorech poznamenal:

„ Pochopení zapouzdření nebylo těžké, něco mi na začátku nešlo, ale sousedi to měli v pořádku, a tak jsem se to od nich dozvěděl, no a potom jsem to udělal správně.“

Proband L2 tedy použil strategii *učí se od ostatních lidí*, konkrétně přístup *učí se od spolužáků*.

5.2.8 Koncept Abstrakce

Abstrakce je proces skrývání stylu práce s objektem a zobrazování užitečných informací, které jsou nutné k pochopení objektu. Abstrakce staví všechny proměnné a metody do třídy, která je nezbytná. Jedná se o akt identifikace chování objektu, který má být ovládnut. Řeší problém na úrovni designu.

Při navrhování aplikací je důležité vědět, kdy použít abstraktní třídu a kdy použít rozhraní. Ačkoli se abstraktní třídy a rozhraní v některých ohledech zdají podobné, existují klíčové rozdíly, které určí, které kosntrukty je nejlépe použít pro to, čeho se snažíme dosáhnout.

Abstraktní třída je speciální typ třídy, z které nelze vytvořit instanci. Abstraktní třída je navržena tak, aby z ní dědily podtřídy, které buď implementují, nebo přepíší její metody. Jinými slovy, abstraktní třídy jsou buď částečně implementovány, nebo nejsou implementovány vůbec. Ve své abstraktní třídě můžeme mít metody, které mohou být abstraktní i konkrétní. Abstraktní třída může mít konstruktory, to je jeden ze zásadních rozdílů mezi abstraktní třídou a rozhraním.

Abstraktní třída nám umožňuje vytvořit funkce, které mohou podtřídy implementovat nebo přepsat. Rozhraní nám umožňuje pouze definovat funkce, nikoliv jejich implementaci. Třída může rozšiřovat pouze jednu abstraktní třídu, kdežto rozhraní můžeme využívat více.

Pochopení rozdílů v použití mezi abstraktní třídou a rozhraním je klíčové pro navrhování aplikací s nízkou provázaností a snadnou rozšiřitelností.

Jaké problémy koncept Abstrakce řeší

Abstrakce umožňuje návrhářům programů oddělit princip od konkrétních instancí, které implementují detaily. To znamená, že kód programu lze zapsat tak, že tento nemusí záviset na konkrétních podrobnostech spolupracujících aplikací, softwaru operačního systému nebo hardwaru, ale výlučně na konceptu řešení. Řešení problému pak lze integrovat do frameworku systému s minimem dalších nutných prací. To nám umožňuje při řešení svých problémů využívat práce jiných programátorů, přičemž se od nás vyžaduje pouze obecného porozumění implementace cizích kódů.

Abstrakce lze dosáhnout abstraktními třídami. C♯ nám umožňuje vytvářet abstraktní třídy, které se používají k zajištění částečné implementace, a finální implementaci lze provést až ve zděděné odvozené třídě. Abstraktní třída obsahuje

abstraktní metody, které musí být implementovány v odvozených třídách.

Abstraktních tříd můžeme využít k návrhu komponent a určit tak úroveň společných schopností (metod), které musí zděděné třídy implementovat.

Abstraktní třídy mají několik klíčových bodů:

- Nemůžeme vytvořit instanci abstraktní třídy.
- Nemůžeme deklarovat abstraktní metodu mimo abstraktní třídu.
- Abstraktní třída nemůže být zapečetěná (sealed). Když vytvoříme sealed třídu, pak ji nemůžeme zdědit, ale abstraktní třída může obsahovat sealed třídy.
- Podtřídu abstraktní třídy lze vytvořit pouze v případě, že implementuje všechny abstraktní metody své nadtřídy.
- Nemůžeme deklarovat abstraktní metody jako soukromé (private).

Kompilátor nám neumožní vytvořit objekt (instanci) abstraktní třídy, protože nemůžeme vyvolat abstraktní metody pomocí objektu, které runtime modulem nemohou být spuštěny.

Přiřazení charakteristik a přístup studentů konceptu Abstrakce

Tento koncept vyžaduje zvládnutí pochopení základních pojmů, ale též schopnosti kombinování, pochopení souvislostí, porovnávání a aplikaci. Proto jej dle taxonomie SOLO řadíme do úrovně *relační*.

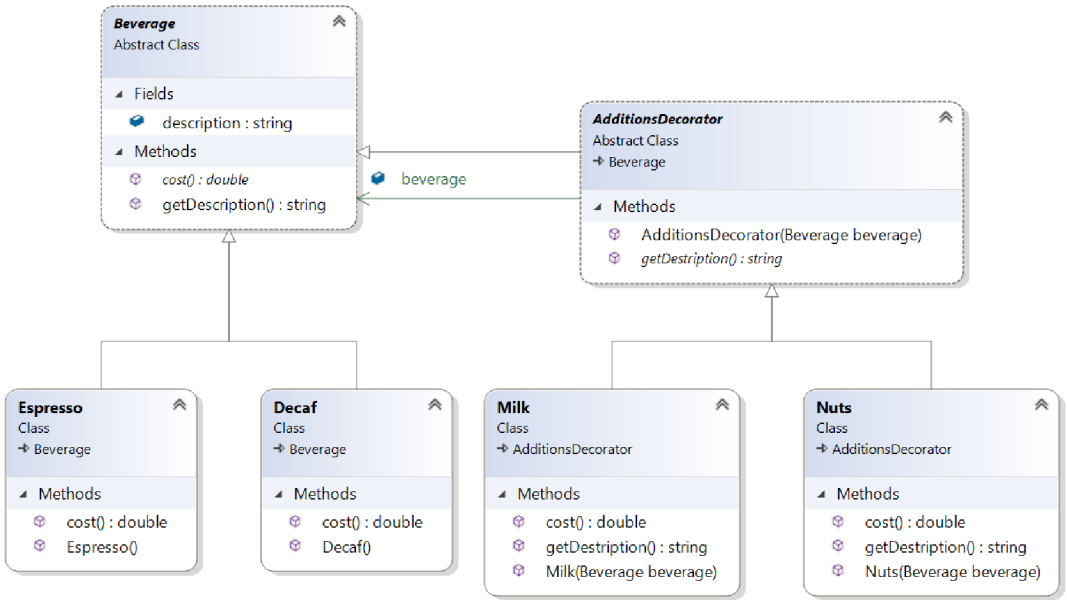
Na základě rozhovorů, vyjádření studentů, jejich reakcí při řešení úkolů a analýzy získané dokumentace přiřazujeme tomuto konceptu charakteristiky

- problémový,
- integrativní,
- diskurzivní.

Tento koncept dělal studentům větší problémy než předchozí koncepty. Proband J2 popsal tento problém:

> „ *Dalo mi problém pochopit smysl abstraktní třídy, no a potom hlavně použít to v mých příkladech to bylo ještě o level dál. Harjí tam roli rozné vztahy a je to na začátku strašně nepřehledné. Dost mi ale daly různé příklady, které jsem si našel na Internetu a pročetl. Pak jsem to pobral.* “ (proband J2)

Ukázka, jak po pochopení tohoto konceptu řešil své příklady proband J2, je na obr. 5.7 a výpisu 5.8 Z uvedeného diagramu tříd a zdrojového kódu bylo pak zřejmé, že daný koncept zvládl.



Obrázek 5.7: Příklad diagramu tříd pro řešení úkolu pro pochopený koncept Abstrakce
Zdroj: (proband J2)

Výpis 5.8: Příklad zdrojového kódu řešení úkolu pro pochopený koncept Abstrakce

```
1 abstract class Beverage
2 {
3     public string description = "Unknown beverage";
4     public virtual string getDescription()
5     {
6         return description;
```

```
    }

    public abstract double cost();
}

class Espresso : Beverage
{
    public Espresso()
    {
        description = "Espresso";
    }

    public override double cost()
    {
        return 50;
    }
}

class Decaf : Beverage
{
    public Decaf()
    {
        description = "CopyOfEspresso";
    }

    public override double cost()
    {
        return 50;
    }
}

abstract class AdditionsDecorator : Beverage
{
    public Beverage beverage;
    public AdditionsDecorator(Beverage beverage)
    {
        this.beverage = beverage;
    }

    public abstract string getDestription();
}

class Milk : AdditionsDecorator
{
```

```
51      public Milk(Beverage beverage) : base(beverage)
52      {
53      }
54
55      public override double cost()
56      {
57          return beverage.cost() + 10;
58      }
59
60      public override string getDestription()
61      {
62          return beverage.getDescription() + ", Milk";
63      }
64  }
65
66  class Nuts: AdditionsDecorator
67  {
68      public Nuts(Beverage beverage) : base(beverage)
69      {
70      }
71
72      public override double cost()
73      {
74          return beverage.cost() + 10;
75      }
76
77      public override string getDestription()
78      {
79          return string.Format("{0}, Nuts", beverage.getDescription());
80      }
81  }
82
83  class Program
84  {
85      static void Main(string[] args)
86      {
87          Beverage beverage1 = new Espresso();
88          Console.WriteLine("Napoj {0}\ncena {1}", beverage1.getDescription(),
                beverage1.cost());
89
90          Beverage beverage2 = new Espresso();
91          beverage2 = new Milk(beverage2);
92          beverage2 = new Nuts(beverage2);
93          Console.WriteLine("Napoj {0}\ncena {1}", beverage2.getDescription(),
```

```
            beverage2.cost());
94      }
95  }
```

Zdroj: proband J2

Konceptu Abstrakce jsme tedy přiřadili na základě rozhovorů se studenty a ukázek návrhů jejich aplikace a kódů charakteristiku *problémový*.

Strategie studentů pro překonání problémů s konceptem Abstrakce

Tabulka 5.9 ukazuje způsob, který studenti využili, pokud měli problém s daným konceptem při řešení programových úloh. Studenti s konceptem měli značné obtíže, nicméně, pro úplné pochopení studenti použili i několik strategií uvedených v tab. 5.9.

Tabulka 5.9: Přístup a strategie studentů pro překonání problému s konceptem Abstrakce

Použitá strategie	Přístup	Procento studentů využívajících danou strategii
Učí se od ostatních lidí	Učí se od spolužáků	31,6
	Získá pomoc od učitele	10,5
Učí se z psaných materiálů	Čte instrukce z knihy	10,5
	Čte instrukce z internetu	21,1
Učí se z příkladů	Používá vlastní příklady	10,5
	Hledá příklady na internetu	79,9
Užití jiného přístupu	Pokus a omyl	5,3
	Vyhnutí se problému	0
	Udělá si přestávku a zkusí řešit po přestávce	5,3

Zdroj: autor

Ukázka strategie, kterou jsme nazvali *užití jiného přístupu*, přístup *udělá si přestávku a zkusí řešit po přestávce*, proband K1.

> *„ Tady mám chybu (pozn: zde měla studentka chybu, protože při vytváření v class diagramu nastavila datový typ string. Po chvíli si toho všimla a správně opravila na void). Tady má být void ano, proto to nefungovalo.*

Když jdu níž, tak tady mám třídu translator, předpokládám, že tady musím vytvořit asociaci, protože chceme přistoupit k datům v jiné třídě. Takže si nastavím, že IGermanMam bude german a pak to určíme přes this. A musíme to dát i do parametru.

Chceme tedy angličtinu přehodit na němčinu, takže tam napíšeme do metody speakinginenglish, takže napíšeme german.speakinggerman. Zkusíme, co to udělá v mainu, jdeme si to vyzkoušet.

Takže budeme mít iGermanMan Paul, který bude new Paul a takhle si vytvoříme i George.

A teda musíme si vytvořit translator a ten musí překládat pro George, takže tam by mělo být Ibrittishman translator = new translator a v závorkách by měl být parametr german.

A tady to podtrhává, zkusíme, jestli mi pomůže žárovička, ale asi moc ne. Takže, nevytvořili jsme si tady pole? Ne asi ne. To ne.

(pozn.: zde studentka přestala psát a prohlížela si již hotový zdrojový kód)

Musela jsem si udělat rekapitulaci, dám tam slovíčko german…, hmm, tak to tam nejspíš nepůjde, protože mi to píše, že tady neexistuje. Možná zkusit tam dát Paul, kterej taky bude odkazovat na Němčinu.

Takže máme vytvořený všechny instance, zkusíme jim říct, ať něco udělají. Takže si zavoláme metody, Paul.speakingingerman a pak řekne něco i George (pozn.: george.speakinginenglish).

Takže, tohle tedy funguje, ale nerozumí si.

Takže, zkusím zavolat teď translator.

Ale on mluví paul stejně německy, což by znamenalo, že bych musela to prohodit (pozn.: studentka má na mysli v instanci translator), ale metodu mi to tam nenabízí. A já přemýšlím proč. Takže znovu si tedy ujas-

níme, co tedy chceme. A zkusíme to zavolat. Vypíše to tedy německy, my jsme ale chtěli, aby to přeložilo teda tomu britovi tomu, co říká paul, takže to musíme prohodit, tedy přehodíme translator, takže místo toho to bude dědit z Igermanman a místo asociace z německého rozhraní to bude asociace z anglického rozhraní.

(pozn.: studentka přehazuje a přepisuje metody.)

Takže chceme, aby to co řekl Paul se přeložilo Georgovi do anglličtiny, teď mi to podtrhává parametr v mainu, takže tam napíšeme George, ale to podtrhává, takže taky ne, možná napsat tam tu asociaci s britským rozraním (pozn: typ brittish), ale ne, to nepomohlo.

(pozn.: studentka si dává pauzu)

Takže jsem zpět, musela jsem si dát a pauza a pak se podívat do kódu a zjistila jsem, že musíme vytvořit třídu, v kódu jí máme jako TestDog a u sebe si jí pojmenuju jako Test. A bude to tedy dědit z Ibrittishman a v těle bude tedy, že ta osoba bude mluvit anglicky.

(pozn.: chybná syntaxe byla z důvodu překlepu.)

Vytvoříme si tedy něco v mainu. Takže si vytvoříme Test a do závorek musíme napsat George.

To funguje. A teď si zkusíme napsat překlad pro Paula, takže napíšeme Test a do závorek musíme dát Translator, protože ten překládá z němčiny.

A ano, už to funguje. “

Výpis 5.9: Příklad zdrojového kódu řešení úkolu pro pochopený koncept Abstrakce

```
1 class Program
2 {
3     static void Main(string[] args)
4     {
5         IGermanMan Paul = new Paul();
6         IBrittishMan George = new George();
```

```
        IBrittishMan Translator = new Translator(Paul);

        Paul.speakingInGerman();
        George.speakingInEnglish();

        Console.WriteLine("--------------------------------");
        Translator.speakingInEnglish();

        Test(George);

        Console.WriteLine("----------------------------");
        Test(Translator);
    }

    private static void Test(IBrittishMan george)
    {
        george.speakingInEnglish();
    }
}

public interface IBrittishMan
{
    void speakingInEnglish();
}

public interface IGermanMan
{
    void speakingInGerman();
}

public class George : IBrittishMan
{
    public void speakingInEnglish()
    {
        Console.WriteLine("Oh, hello there, how are you?");
    }
}

public class Paul : IGermanMan
{
    public void speakingInGerman()
    {
        Console.WriteLine("Halo, wie gehts?");
    }
```

```
51 }
52
53 public class Translator : IBrittishMan
54 {
55     IGermanMan german;
56     public Translator(IGermanMan german)
57     {
58         this.german = german;
59     }
60
61     public void speakingInEnglish()
62     {
63         german.speakingInGerman();
64     }
65 }
```

Zdroj: proband K1

Při řešení konkrétního příkladu (výpis 5.9) proband poměrně velmi rychle vyřešil úkol pro koncept Abstrakce. Zajímavé je využití přístupu *udělá si přestávku a zkusí řešit po přestávce*. Proband získal určitý nadhled, byl schopen daný koncept použit, což znamená, že mu i porozuměl.

5.2.9 Koncept Rozhraní

V objektově orientovaném programovacím jazyce, který nemá modulový systém, je využíváno rozhraní pro implementaci modulárního programování na úrovni komponent.

Programování založené na rozhraní umožňuje definovat aplikaci jako kolekci společných komponent, ve kterých lze jejich vzájemné volání mezi sebou provádět prostřednictvím abstraktních rozhraní, a nikoliv prostřednictvím specifických tříd. Příslušné instance tříd se mohou získat prostřednictvím jiných technik, např. prostřednictvím třídy známého vzoru Factory.

Jaké problémy koncept Rozhraní řeší

Použití rozhraní zvyšuje modularitu aplikace a tím i její udržovatelnost. Pouhé rozdělení aplikace na libovolné součásti komunikující prostřednictvím rozhraní

samo o sobě nezaručuje nízkou vazbu nebo vysokou soudržnost, což jsou další dva atributy, které se běžně považují za klíčové pro udržitelnost aplikace.

Rozhraní definuje kontrakt (dohodu). Jakékoliv třídy nebo struktury, které daný konktrakt implementují, musí rovněž nabídnout implementaci všech členů definovaných v použitém rozhraní.

Rozhraní může být definováno ve jmenném prostoru nebo v kontextu třídy. Jeho deklarace může obsahovat následující členy:

- Metody
- Vlastnosti
- Indexery
- Události

Platí, že všechny výše uvedené členy jsou pouze deklarovány, tedy jsou uvedeny pouze jejich signatury bez vlastní implementace. Rozhraní může dědit z jednoho nebo více bázových rozhraní.

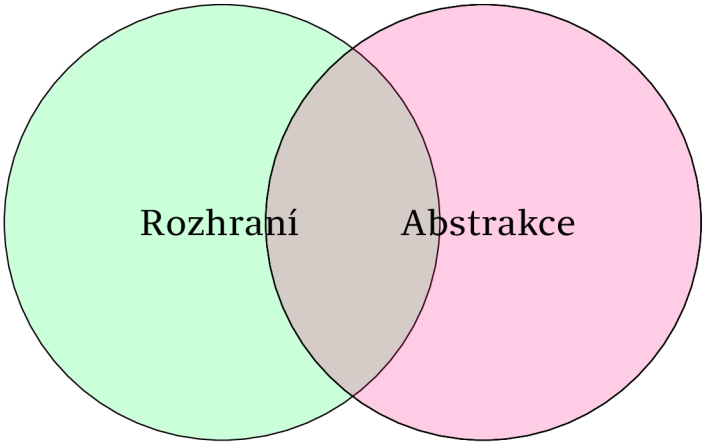
Jedno z prvních principů, co se dozvíme z knihy Design Patterns (Holub, 2004), je tento následující princip:

Programujte proti rozhraní, nikoliv proti implementaci.

Rozhraní je jen jazykový konstrukt. V podstatě je to jen tvar. Je to jako zástrčka a zásuvka. V Evropě používáme jeden druh a USA používají jiný, ale pouze podle konvence přenášíme 230 V přes evropské zásuvky a 110 V přes americké zásuvky. Přestože se zástrčky vejdou pouze do příslušných zásuvek, nic nám nebrání v odesílání 230 V prostřednictvím kombinace US zástrčka/zásuvka.

Rozhraní lze použít jako abstrakce, ale použití rozhraní samo o sobě nezaručuje, že se jedná o abstrakci. Obr. 5.8 zobrazuje, jak bychom takovýto vztah mezi rozhraními a abstrakcemi mohli chápat.

Porušení principu substituce Porušení principu substituce Liskovové je docela zřejmým znamením, že používané rozhraní je špatnou abstrakcí. To může být



Obrázek 5.8: Vztah mezi rozhraním a abstrakcí

Zdroj: autor

nejzřejmější, když uživatel potřebuje přetypovat instanci na rozhraní, aby s ní správně pracoval.

Jak zdůrazňuje Martin (2017), i rozhraní jednoduché jako je zdánlivě neškodná obdélníková „abstrakce“, obsahuje potenciální nebezpečí. Problematický kód, který jsme zkoušeli se studenty, Představuje výpis 5.10.

Výpis 5.10: Deklarace rozhraní IRectangle

```
1 public interface IRectangle
2 {
3     int Width {get; set; }
4     int Height {get; set; }
5 }
```

Zdroj: Martin, 2017

Problém se projeví, když se pokusíme nechat třídu *Square* implementovat *IRectangle*. Chceme-li chránit invarianty čtverce, nemůžeme dovolit, aby se vlastnosti Šířka a Výška lišily. Máme několik možností, z nichž žádná není příliš dobrá:

- Aktualizovat šířku i výšku na stejnou hodnotu, když se zapisuje jedna z nich.
- Ignorovat operaci zápisu, když se volající pokusí přiřadit neplatnou hodnotu.
- Vyvolat výjimku, když se volající pokusí přiřadit šířku, která se liší od výšky (a naopak).

Studenti nejčastěji tuto úlohu řešili prvním způsobem, kdy aktualizovali hodnoty šířky i výšky na stejnou, aktuálně zapisovanou hodnotu.

Student J2 se k dané problematice vyjádřil celkem překvapeně.

„ To mě vůbec todle nenapadlo. Taková prkotina a co to nadělá za problémy. “ (proband J2)

Z pohledu uživatele rozhraní *IRectangle* by jej všechny tyto možnosti ve svém chování přinejmenším překvapily. Řešení pomocí vyvolání výjimek by způsobilo, že by se uživatel choval jinak při konzumaci instancí *Square* na rozdíl od „normálních“ obdélníků.

Problém pramení ze skutečnosti, že operace mají vedlejší účinky. Vyvolání jedné operace změní stav zdánlivě nesouvisejících dat. Čím více členů máme, tím větší je riziko, takže *princip segregace rozhraní* (jak jsme si představili v kapitole 2.2.3) může do jisté míry pomoci.

Hlavičková rozhraní Jelikož vyšší počet členů zvyšuje riziko neočekávaných vedlejších účinků a dočasné vazby, nemělo by být překvapením, že rozhraní mechanicky extrahovaná ze všech členů konkrétní třídy jsou špatné abstrakce.

Microsoft Visual Studio jako vždy usnadňuje dělat špatné věci tím, že nabízí funkci refaktoringu extrahováním Interface.

Taková rozhraní se nazývají Hlavičková rozhraní, protože se podobají hlavičkovým souborům C++. Mají tendenci jednoduše uvádět totéž dvakrát bez zjevného přínosu. To platí zejména v případě, že máme pouze jednu implementaci, což je u rozhraní s mnoha členy velmi pravděpodobné.

Mělká rozhraní Když použijeme funkci refaktoringu *extrahování rozhraní* v sadě Microsoft Visual Studio, i když nerozbalíme všechny členy, výsledné rozhraní je mělké, protože rekurzivně neextrahuje rozhraní z konkrétních typů vystavených extrahovanými členy.

Příkladem může být extrakce rozhraní z LINQ do SQL nebo LINQ do kontextu entit za účelem definování rozhraní úložiště. Jako příklad si můžeme uvést roz-

hraní extrahované z velmi jednoduchého kontextu LINQ to Entities, jak je uvedeno ve výpisu 5.11.

Výpis 5.11: Deklarace rozhraní IPostingContext

```
1 public interface IPostingContext
2 {
3     void AddToPostings(Posting posting);
4     ObjectSet<Posting> Postings { get; }
5 }
```

Zdroj: Martin, 2017

Na první pohled to může vypadat užitečně, ale není tommu tak. I když je to rozhraní, je stále pevně spojeno s konkrétním kontextem objektu. Nejen, že *ObjectSet<T>* odkazuje na Entity Framework, ale třída Posting je definována velmi specifickým, automaticky generovaným kontextem Entity.

Rozhraní nám může dát dojem, že pracujeme proti volně vázanému kódu, ale nemůžeme snadno (pokud vůbec) implementovat jiný IPostingContext s radikálně odlišnou technologií přístupu k datům. Uvízneme u tohoto konkrétního *PostingContext*.

Pokud musíme extrahovat rozhraní, budeme to muset provádět rekurzivně.

Přiřazení charakteristik a přístup studentů konceptu Rozhraní

Na základě rozhovorů, vyjádření studentů, jejich reakcí při řešení úkolů a analýzy získané dokumentace přiřazujeme tomuto konceptu charakteristiky

- transformativní,
- problematický,
- nezvratný,
- integrativní,
- diskurzivní,
- liminální.

Zvládnutí tohoto konceptu vyžaduje od studenta vysokou míru kompetencí. Nestačí si osvojit základní pojmy a být kompetentní v algoritmické, analytické a aplikační rovině, ale student musí též při aplikaci daného konceptu být schopen teoretizování, přemýšlení a zobecnění. Proto jej dle taxonomie SOLO řadíme do nejvyšší úrovně, *úrovně rozšířeného abstraktu*.

Jedná se tedy, dle našeho názoru, o **prahový koncept**.

Následující ukázka diagramu tříd (obr. 5.9) a výpisu kódu (výpis 5.12) studenta K1 nám jasně demonstruje, že si proband sledovaný koncept neosvojil a nepochopil jeho aplikaci. Takovéto řešení se vyskytovalo ve většině případů a lze na něm demonstrovat, že uvedený koncept byl osvojen pouze částěčně, proto přisuzujeme tomuto konceptu charakteristiku *problematický*.

Zadaný úkol nabízí k řešení velmi častý způsob použití a aplikace tohoto konceptu a objevuje se i u ostatních konceptů (např. abstrakce, kompozice či událost). Sledovali jsme, že tento typický případ byl pro studenty poměrně nejasný, byť měli možnost si jej opakovaně trénovat. Konceptu tak porozuměli pouze částečně a s dalšími úkoly tak se dostávali do větších zmatků a nejasností, byť si mnohdy tuto skutečnost v rozhovorech sami nepřiznali. Na získané dokumentaci však tento nedostatek byl patrný. Můžeme tak rovněž potvrdit, že studenti svým chováním projevovali tzv. „mimikry“, jak ve svém výzkumu vysledoval již Land a kol. (2014).

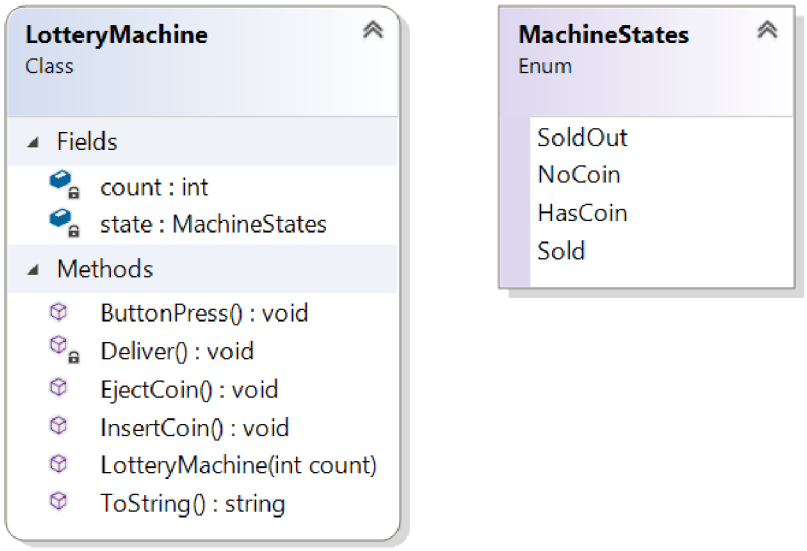
Použití vhodného řešení tohoto konceptu se tak vyhýbali. Z tohoto důvodu přisuzujeme tomuto konceptu charakteristiku *liminální*.

Studenti, kteří si tento koncept osvojili, vykazovali lepší orientaci v ostatních zadaných úkolech a pociťovali lepší pochopení i dalších souvisejících problémů. Z tohoto důvodu přisuzujeme tomuto konceptu charakteristiku *integrativní*.

Z výsledků dokumentace a následných dotazníků a rozhovorů studenta, který tento koncept ovládl (např. v ukázce proband J2) můžeme nalézt značný posun ke kvalitnějšímu využívání programovacích technik. Rovněž bylo u takového studenta patrné, že má jasnou mentální reprezentaci a je schopen danou problema-

tiku programování využívající tento koncept chápat. Z tohoto důvodu přisuzujeme tomuto konceptu charakteristiku *transformativní*.

S vyjasněnou mentální reprezentací a pochopením konceptu se studentovi dále rozšířily možnosti v zápisu zdrojového kódu, který byl takto kvalitnější a přehlednější. Z tohoto důvodu přisuzujeme tomuto konceptu charakteristiku *diskurzivní*.



Obrázek 5.9: Příklad diagramu tříd řešení úkolu pro nepochopený koncept Rozhraní
Zdroj: (proband K1)

Výpis 5.12: Příklad zdrojového kódu řešení úkolu pro nepochopený koncept Rozhraní

```
class Program
{
    static void Main(string[] args)
    {
        LotteryMachine lotteryMachine = new LotteryMachine(5);
        Console.WriteLine(lotteryMachine);
        lotteryMachine.InsertCoin();
        lotteryMachine.ButtonPress();
        Console.WriteLine(lotteryMachine);

        lotteryMachine.InsertCoin();
        lotteryMachine.EjectCoin();
        lotteryMachine.ButtonPress();
        Console.WriteLine(lotteryMachine);
```

```
15      }
16  }
17
18  public class LotteryMachine
19  {
20      private MachineStates state;
21      private int count;
22
23      public LotteryMachine(int count)
24      {
25          this.count = count;
26          if (count > 0)
27          {
28              state = MachineStates.NoCoin;
29          }
30          else
31          {
32              state = MachineStates.SoldOut;
33          }
34      }
35
36      public void InsertCoin()
37      {
38          if (state == MachineStates.HasCoin)
39          {
40              Console.WriteLine("You can't insert another coin");
41          }
42          else if (state == MachineStates.NoCoin)
43          {
44              state = MachineStates.HasCoin;
45              Console.WriteLine("Coin inserted");
46          }
47          else if (state == MachineStates.SoldOut)
48          {
49              Console.WriteLine("Don't insert a coin, machine is sold out.");
50          }
51          else if (state == MachineStates.Sold)
52          {
53              Console.WriteLine("We're already giving you a lottery ticket");
54          }
55      }
56
57      public void ButtonPress()
58      {
```

```
59          if (state == MachineStates.HasCoin)
60          {
61              Console.WriteLine("Preparing a lottery ticket");
62              state = MachineStates.Sold;
63              Deliver();
64          }
65          else if (state == MachineStates.NoCoin)
66          {
67              Console.WriteLine("You pressed a button, but no coin inserted");
68          }
69          else if (state == MachineStates.SoldOut)
70          {
71              Console.WriteLine("You pressed a button, but machine is sold out.");
72          }
73          else if (state == MachineStates.Sold)
74          {
75              Console.WriteLine("We're already giving you a lottery ticket");
76          }
77      }
78
79      private void Deliver()
80      {
81          if (state == MachineStates.Sold)
82          {
83              Console.WriteLine("Giving a lottery ticket");
84              count--;
85              if (count == 0)
86              {
87                  Console.WriteLine("Out of lottery tickets");
88                  state = MachineStates.SoldOut;
89              }
90              else
91              {
92                  state = MachineStates.NoCoin;
93              }
94          }
95          else if (state == MachineStates.NoCoin)
96          {
97              Console.WriteLine("You need to insert a coin");
98          }
99          else if (state == MachineStates.SoldOut)
100         {
101             Console.WriteLine("No lottery ticket delivered");
102         }
```

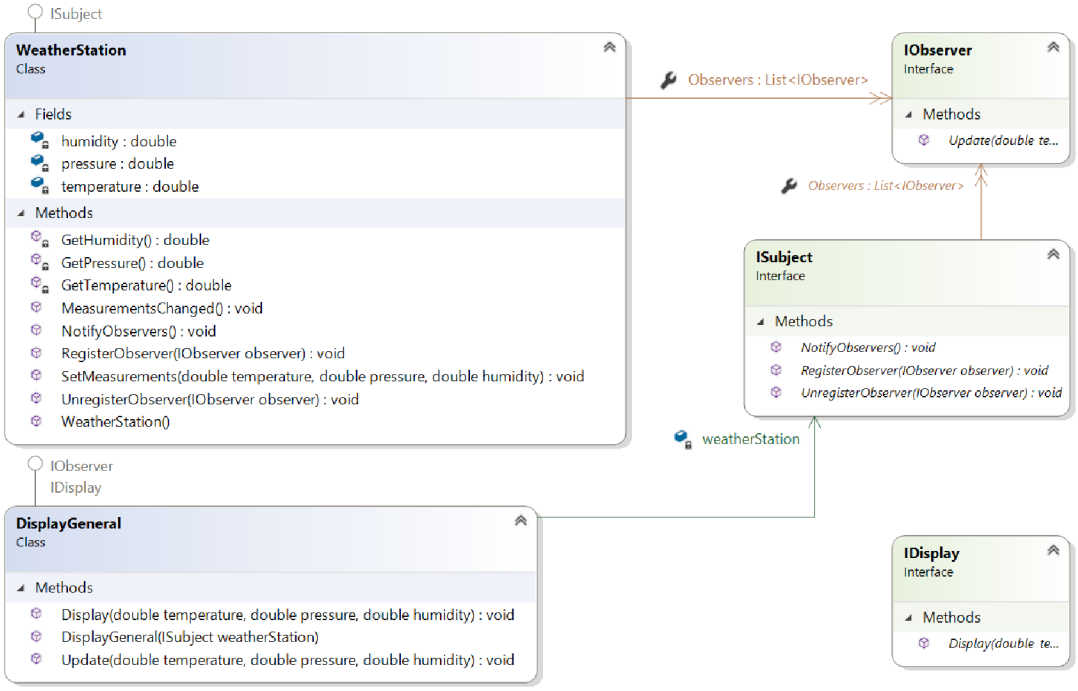
```
103         else if (state == MachineStates.HasCoin)
104         {
105             Console.WriteLine("You need to press button");
106         }
107 
108     }
109     public override string ToString()
110     {
111         string ret = string.Format("Lottery tickets: {0} | current state: {1}",
112             count, state.ToString());
113         return ret;
114     }
115 
116     public void EjectCoin()
117     {
118         if (state == MachineStates.HasCoin)
119         {
120             Console.WriteLine("Coin returned");
121         }
122         else if (state == MachineStates.NoCoin)
123         {
124             Console.WriteLine("You haven't inserted a coin");
125         }
126         else if (state == MachineStates.Sold)
127         {
128             Console.WriteLine("Sorry, you already pressed a button");
129         }
130         else if (state == MachineStates.SoldOut)
131         {
132             Console.WriteLine("You haven't inserted a coin, sold out");
133         }
134     }
135 }
136 
137 public enum MachineStates
138 {
139     SoldOut,
140     NoCoin,
141     HasCoin,
142     Sold
143 }
```

Zdroj: (proband K1)

Na základě konzultace s vybranými vyučujícími dalších odborných předmětů,

jsme dospěli k poznání, že většině studentů chybí kompetence související s abstraktním uvažováním a zobecňováním naučených postupů a pojmů, tj. dle taxonomie SOLO úroveň *rozšířeného abstraktu*.

Ze sledovaných probandů se ve třech případech podařilo studentům si koncept osvojit (viz příklad probanda J2 na obr. 5.10 a výpis 5.13):



Obrázek 5.10: Příklad diagramu tříd řešení úkolu pro pochopený koncept Rozhraní
Zdroj: (proband J2)

Výpis 5.13: Příklad zdrojového kódu řešení úkolu pro pochopený koncept Rozhraní

```
1  class Program
2  {
3      static void Main(string[] args)
4      {
5          WeatherStation weatherStation = new WeatherStation();
6          DisplayGeneral displayGeneral = new DisplayGeneral(weatherStation);
7          weatherStation.SetMeasurements(20, 1000, 60);
8          weatherStation.SetMeasurements(5, 1050, 40);
9      }
10 }
11
12 public class WeatherStation : ISubject
13 {
```

```
14      private double temperature;
15      private double pressure;
16      private double humidity;
17
18      public List<IObserver> Observers { get; set;  }
19
20      public WeatherStation()
21      {
22          Observers = new List<IObserver>();
23      }
24
25      public void NotifyObservers()
26      {
27          foreach (var observer in Observers)
28          {
29              observer.Update(temperature, pressure, humidity);
30          }
31      }
32
33      public void RegisterObserver(IObserver observer)
34      {
35          Observers.Add(observer);
36      }
37
38      public void UnregisterObserver(IObserver observer)
39      {
40          Observers.Remove(observer);
41      }
42      public void MeasurementsChanged()
43      {
44          NotifyObservers();
45      }
46
47      private double GetHumidity()
48      {
49          return humidity;
50      }
51
52      private double GetPressure()
53      {
54          return pressure;
55      }
56
57      private double GetTemperature()
```

```
58      {
59          return temperature;
60      }
61
62      public void SetMeasurements(double temperature, double pressure, double humidity)
63      {
64          this.temperature = temperature;
65          this.pressure = pressure;
66          this.humidity = humidity;
67          MeasurementsChanged();
68      }
69  }
70
71  public interface IObserver
72  {
73      void Update(double temperature, double pressure, double humidity);
74  }
75
76  public class DisplayGeneral : IObserver, IDisplay
77  {
78      private ISubject weatherStation;
79
80      public DisplayGeneral(ISubject weatherStation)
81      {
82          this.weatherStation = weatherStation;
83          weatherStation.RegisterObserver(this);
84      }
85
86      public void Display(double temperature, double pressure, double humidity)
87      {
88          Console.WriteLine($"t = {temperature}, " +
89              $"p = {pressure}, h = {humidity}");
90      }
91
92      public void Update(double temperature, double pressure, double humidity)
93      {
94          Display(temperature, pressure, humidity);
95      }
96  }
97
98  public interface ISubject
99  {
100     List<IObserver> Observers { get; set; }
101
```

```
102     void RegisterObserver(IObserver observer);
103     void UnregisterObserver(IObserver observer);
104     void NotifyObservers();
105 }
106 
107 public interface IDisplay
108 {
109     void Display(double temperature, double pressure, double humidity);
110 }
```

Zdroj: (proband J2)

Strategie studentů pro překonání problémů s konceptem Rozhraní

Tab. 5.10 ukazuje strategie, kterými se studenti snažili překonat problémy s uvedeným konceptem, a které uvedli v rozhovorech. Z rozhovorů se zdálo, že konceptům rozumí, avšak z detailní následné analýzy výsledných kódů, bylo opatrné, že příslušný koncept nebyl detailně pochopen a studenti nebyli schopni správně aplikovat koncepty ve svých programových úlohách.

Tabulka 5.10: Přístup a strategie studentů pro překonání problému s konceptem Rozhraní

Použitá strategie	Přístup	Procento studentů využívajících danou strategii
Učí se od ostatních lidí	Učí se od spolužáků	78,9
	Získá pomoc od učitele	21,1
Učí se z psaných materiálů	Čte instrukce z knihy	10,5
	Čte instrukce z internetu	21,1
Učí se z příkladů	Používá vlastní příklady	10,5
	Hledá příklady na internetu	84,2
Užití jiného přístupu	Pokus a omyl	31,6
	Vyhnutí se problému	52,6
	Udělá si přestávku a zkusí řešit po přestávce	0

Zdroj: autor

Jeden ze studentů, který úspěšně ovládl koncept, využil znalosti z předchozího studia a dva studenti úspěšně uplatnili strategii *učení se od ostatních lidí*,

kdy docházeli na pravidelné konzultace k vyučujícímu. Tito měli silnou vnitřní motivaci se naučit dobře naučit paradigma OOP.

Na rozdíl od strategií, kterými překonávaly obtíže s předchozími koncepty, a které vedly k úspěšnému porozumění, v případě tohoto konceptu uvedené strategie většinou nevedly k úspěšnému porozumění a pochopení, což se projevilo zejména v nesprávném řešení programových úloh. Tyto úlohy sledovaly pochopení a aplikaci konceptu Rozhraní.

Ukázka přepisu probanda M1 k situaci, kdy zkouší, hledá správné řešení přístupem *pokus-omyl*. Následně se snaží najít *radu na internetu*. Následuje přístup *nastolené přestávky*. Po nastalé pauze pokračuje ve snaze úkol vyřešit. S pomocí *zdrojů z internetu* je schopna program zfunkčnit.

> *„Tady mám chybu (pozn: zde měla studentka chybu, protože při vytváření v class diagramu nastavila datový typ string. Po chvíli si toho všimla a správně opravila na void). Tady má být void ano, proto to nefungovalo.*
>
> *Když jdu níž, tak tady mám třídu translator, předpokládám, že tady musím vytvořit asociaci, protože chceme přistoupit k datům v jiné třídě. Takže si nastavím, že IGermanMam bude german a pak to určíme přes this. A musíme to dát i do parametru.*
>
> *Chceme tedy angličtinu přehodit na němčinu, takže tam napíšeme do metody speakinginenglish, takže napíšeme german.speakinggerman. Zkusíme, co to udělá v mainu, jdeme si to vyzkoušet.*
>
> *Takže budeme mít iGermanMan Paul, který bude new Paul a takhle si vytvoříme i George.*
>
> *A teda musíme si vytvořit translator a ten musí překládat pro George, takže tam by mělo být Ibrittishman translator = new translator a v závorkách by měl být parametr german.*
>
> *A tady to podtrhává, zkusíme, jestli mi pomůže žárovička, ale asi moc ne. Takže, nevytvořili jsme si tady pole? Ne asi ne. To ne.*

(pozn.: zde studentka přestala psát a prohlížela si již hotový zdrojový kód)

Musela jsem si udělat rekapitulaci, dám tam slovíčko german..., hmm, tak to tam nejspíš nepůjde, protože mi to píše, že tady neexistuje. Možná zkusit tam dát Paul, kterej taky bude odkazovat na Němčinu.

Takže máme vytvořený všechny instance, zkusíme jim říct, ať něco udělají. Takže si zavoláme metody, Paul.speakingingerman a pak řekne něco i George (pozn.: george.speakinginenglish).

Takže tohle tedy funguje a ale nerozumí si.

Takže zkusím zavolat teď translator.

Ale on mluví paul stejně německy, což by znamenalo, že bych musela to prohodit (pozn.: studentka má na mysli v instanci translator), ale metodu mi to tam nenabízí. A já přemýšlím proč. Takže znovu si tedy ujasníme, co tedy chceme. A zkusíme to zavolat. Vypíše to tedy německy, my jsme ale chtěli, aby to přeložilo teda tomu britovi tomu, co říká paul, takže to musíme prohodit, tedy přehodíme translator, takže místo toho to bude dědit z Igermanman a místo asociace z německého rozhraní to bude asociace z anglického rozhraní.

(pozn.: přehazuje a přepisuje metody.)

Takže chceme, aby to co řekl Paul se přeložilo Georgovi do anglličtiny, teď mi to podtrhává parametr v mainu, takže tam napíšeme George, ale to podtrhává, takže taky ne, možná napsat tam tu asociaci s britským rozraním (slovíčko brittish) ale ne, to nepomohlo.

(pozn.: studentka si dává pauzu.)

Takže jsem zpět, musela jsem se podívat do kódu a zjistila jsem, že musíme vytvořit třídu, v kódu jí máme jako TestDog a u sebe si jí pojmenuju jako Test. A bude to tedy dědit z Ibrittishman a v těle bude tedy, že ta oso-

ba bude mluvit anglicky. (pozn: chyba syntaxe se zobrazovala z důvodu překlepu.)

Vytvoříme si tedy něco v mainu. Takže si vytvoříme Test a do závorek musíme napsat George.

To funguje. A teď si zkusíme napsat překlad pro Paula, takže napíšeme Test a do závorek musíme dát Translator, protože ten překládá z němčiny.

A ano, už to funguje.“ (M1)

Častou strategií studentů bylo *vyhnutí se problému* (použití konceptu), kdy nahrazovali koncept jinými jim známými konstrukty z programovacího jazyka, např. výčet, jak je patrno z ukázky (obr. 5.9 a výpis 5.12).

5.2.10 Koncept Událost

Události umožňují třídě nebo objektu upozornit jiné třídy nebo objekty, když dojde k nějakému zájmu. Třída, která odesílá (nebo vyvolává) událost, se nazývá *Vydavatel* a třídy, které přijmou (nebo zpracovávají) událost se nazývají *předplatitelé*.

V jazyku C♯ se pro systém událostí používají *delegáti*. *Vydavatel* určí, kdy se událost vyvolá. *Předplatitelé* určují, jakou akci provádí v reakci na událost. Událost může mít více *odběratelů*. *Předplatitel* může zpracovávat více událostí od více *vydavatelů*. Události se obvykle používají k signalizaci uživatelských akcí, jako například kliknutí myší na tlačítko nebo výběry nabídky v grafickém uživatelském rozhraní.

Jaké problémy koncept Událost řeší

Události ve frameworku .NET jsou založené na modelu delegáta. Model delegáta se řídí návrhovým vzorem pozorovatele, který umožňuje odběrateli registrovat

se a přijímat oznámení od poskytovatele. Odesílatel události předá oznámení, že došlo k události, a příjemce události toto oznámení přijme a definuje na něj odpověď.

Událost je zpráva odeslaná objektem, která signalizována výskyt akce. Akce může být způsobená interakcí uživatele, jako je kliknutí na tlačítko, nebo z jiné logiky programu, jako je například změna hodnoty vlastnosti. Objekt, který vyvolá událost, se nazývá *odesílatel* (nebo *vydavatel*) události. *Odesílatel* události neví, který objekt nebo metoda bude přijímat (zpracovávat) události, které vyvolá. Událost je obvykle členem odesílatele události. Například událost je členem třídy a událost je členem Click Button *PropertyChanged* třídy, která implementuje *INotifyPropertyChanged* rozhraní.

Přiřazení charakteristik a přístup studentů konceptu Událost

Na základě rozhovorů, vyjádření studentů, jejich reakcí při řešení úkolů a analýzy získané dokumentace přiřazujeme tomuto konceptu všechny charakteristiky:

- tranformativní,
- problematický,
- nezvratný,
- integrativní,
- diskurzivní,
- liminální.

Jedná se tedy, dle našeho názoru, o **prahový koncept**.

Podobně jako u konceptu Rozhraní, i tento koncept vyžaduje od studentů vysokou míru kompetencí. Kromě osvojení základních pojmů a kompetencí v algoritmické, analytické a aplikační rovině, student musí též při aplikaci daného

konceptu být schopen teoretizování, přemýšlení a zobecnění. Proto jej dle taxonomie SOLO řadíme do nejvyšší úrovně, úrovně *rozšířeného abstraktu*.

Na základě konzultace s vybranými vyučujícími dalších odborných předmětů, jsme dospěli k poznání, že většině studentů chybí kompetence související s abstraktním uvažováním a zobezňováním naučených postupů a pojmů, tj. dle taxonomie SOLO úroveň *rozšířeného abstraktu*.

Studenti se použití vhodného řešení pro tento koncept vyhýbali, podobně jako u rozhraní. Z tohoto důvodu přisuzujeme tomuto konceptu charakteristiku *liminální*.

Výsledky rozhovorů a analýz dokumentace u sledovaných studentů vykazovalo podobnosti jako u konceptu rozhraní.

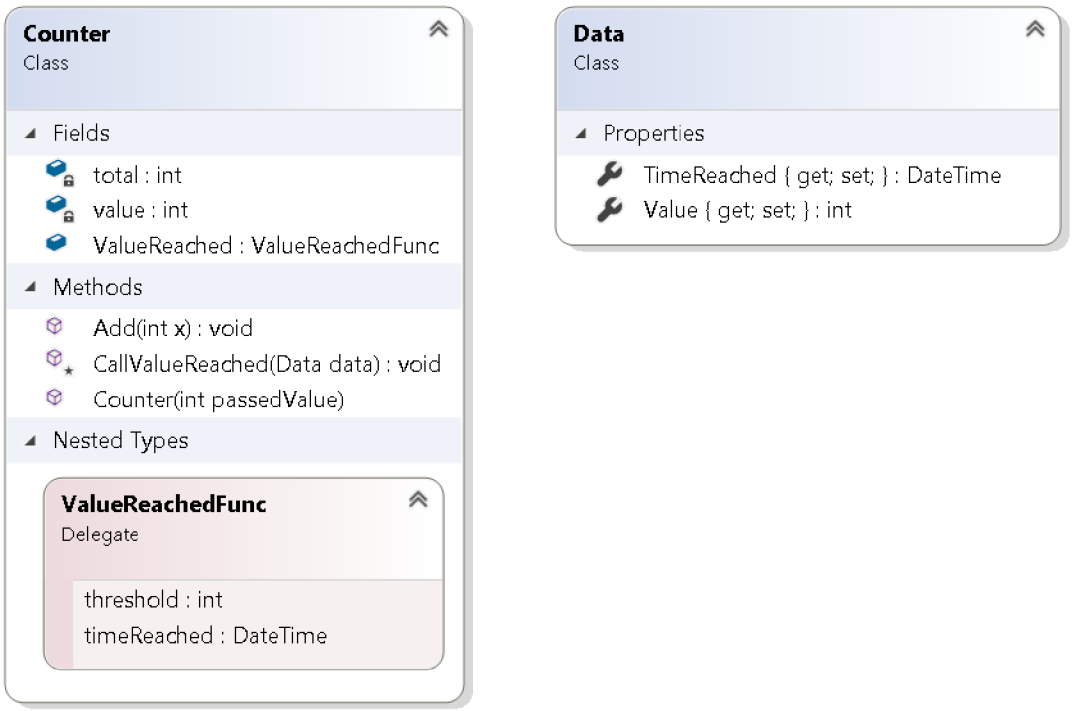
Studenti, kteří si tento koncept osvojili, vykazovali taktéž lepší orientaci v ostatních zadaných úkolech a pociťovali rovněž lepší pochopení i u dalších souvisejících problémů. To se též projevilo na jejich výrazně lepších výsledcích řešení zadaných úkolů. Z tohoto důvodu přisuzujeme tomuto konceptu charakteristiku *integrativní* a *transformativní*.

Z výsledků dokumentace a následných dotazníků a rozhovorů studenta, který tento koncept ovládl (např. v ukázce proband J2), můžeme nalézt značný posun ke kvalitnějšímu využívání programovacích technik. Rovněž bylo u takového studenta patrné, že má jasnou mentální reprezentaci a je schopen danou problematiku programování využívající tento koncept chápat. Toto nám potvrdilo naši volbu přiřadit tomuto konceptu charakteristiku *transformativní*.

S vyjasněnou mentální reprezentací a pochopením konceptu se studentovi dále rozšířily možnosti v zápisu zdrojového kódu, který byl takto kvalitnější a přehlednější. Z tohoto důvodu přisuzujeme tomuto konceptu charakteristiku *diskurzivní*.

Strategie studentů pro překonání problémů s konceptem Událost

Následující diagram tříd (obr. 5.11) a ukázka kódu probanda J2 (výpis 5.14) demonstrují příklad (takovéto případy byly celkem dva), kde studenti měli použít událost při svém řešení. Student ve svém kódu definice události nedokončil, vytvořil pouze delegát, který plánoval použít pro své následné řešení. Z následného rozhovoru vyplynulo, že mentální reprezentaci konceptu událost má student ovládnutu, pouze nestihnul dokončit svůj kód. Pro následné posouzení charakteristik jsme u tohoto studenta považovali problematiku konceptu za zvládnutou. Student pro své řešení použil přístup *hledání na internetu*, kde po prostudování konceptu úkol vyřešil s využitím konstruktu delegát, na kterém je tento koncept založen.



Obrázek 5.11: Příklad diagramu tříd řešení s delegátem úkolu pro koncept Událost
Zdroj: (proband J2)

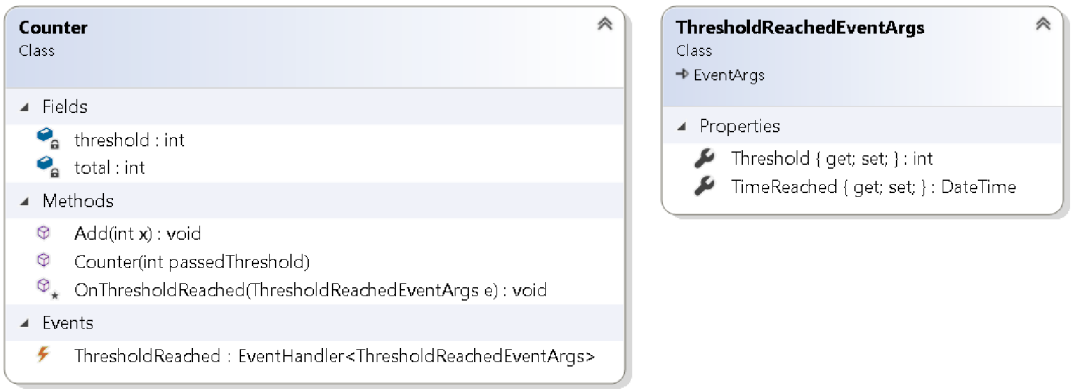
Výpis 5.14: Příklad zdrojového kódu řešení s delegátem úkolu pro koncept Událost

```
1 class Program
2 {
```

```
    static void Main(string[] args)
    {
        Counter c = new Counter(new Random().Next(10));
        c.ValueReached += new Counter.ValueReachedFunc(ValueReachedPrint);

        Console.WriteLine("klavesa 'a' zvysi celkovy pocet");
        while (Console.ReadKey(true).KeyChar == 'a')
        {
            Console.WriteLine("navysuji o 1");
            c.Add(1);
        }
    }

    static void ValueReachedPrint(int Value, DateTime TimeReached)
    {
        Console.WriteLine("Hodnota {0} byla dosazena v {1}.", Value, TimeReached);
        Environment.Exit(0);
    }
}

class Counter
{
    private int value;
    private int total;
    public delegate void ValueReachedFunc(int threshold, DateTime timeReached);
    public ValueReachedFunc ValueReached;

    public Counter(int passedValue)
    {
        value = passedValue;
    }

    public void Add(int x)
    {
        total += x;
        if (total >= value)
        {
            Data args = new Data();
            args.Value = value;
            args.TimeReached = DateTime.Now;
            CallValueReached(args);
        }
    }

```

```
47      protected virtual void CallValueReached(Data data)
48      {
49          ValueReached(data.Value, data.TimeReached);
50      }
51  }
52
53  public class Data
54  {
55      public int Value { get; set; }
56      public DateTime TimeReached { get; set; }
57  }
```

Zdroj: (proband J2)



Obrázek 5.12: Příklad diagramu tříd správného řešení úkolu pro koncept Událost
Zdroj: (proband D1)

Pouze jeden student (proband D1) dokázal vyřešit úlohu komplexně s použitím konceptu Událost. Tento případ demonstruje diagram tříd (obr. 5.12 a ukázka výpisu kódu 5.15). Z následného rozhovoru vyplynulo, že tento student měl již určité předchozí znalosti ohledně tohoto konceptu. K hlubšímu pochopení tohoto konceptu mu pohohl přístup *hledání podobných příkladů na internetu* a následná *konzultace s vyučujícím*.

Výpis 5.15: Příklad zdrojového kódu správného řešení úkolu pro koncept Událost

```
1  class Program
2  {
3      static void Main(string[] args)
4      {
```

```
5         Counter c = new Counter(new Random().Next(10));
6         c.ThresholdReached += c_ThresholdReached;
7
8         Console.WriteLine("ěStisknte klávesu 'a' pro šnavýení hodnoty");
9         while (Console.ReadKey(true).KeyChar == 'a')
10        {
11            Console.WriteLine("řpidávám 1...");
12            c.Add(1);
13        }
14    }
15
16    static void c_ThresholdReached(object sender, ThresholdReachedEventArgs e)
17    {
18        Console.WriteLine("Celková hodnota {0} byla ždosaena v {1}.", e.Threshold,
              e.TimeReached);
19        Environment.Exit(0);
20    }
21}
22
23class Counter
24{
25    private int threshold;
26    private int total;
27
28    public Counter(int passedThreshold)
29    {
30        threshold = passedThreshold;
31    }
32
33    public void Add(int x)
34    {
35        total += x;
36        if (total >= threshold)
37        {
38            ThresholdReachedEventArgs args = new ThresholdReachedEventArgs();
39            args.Threshold = threshold;
40            args.TimeReached = DateTime.Now;
41            OnThresholdReached(args);
42        }
43    }
44
45    protected virtual void OnThresholdReached(ThresholdReachedEventArgs e)
46    {
47        EventHandler<ThresholdReachedEventArgs> handler = ThresholdReached;
```

```
48          if (handler != null)
49          {
50              handler(this, e);
51          }
52      }
53
54      public event EventHandler<ThresholdReachedEventArgs> ThresholdReached;
55  }
56
57  public class ThresholdReachedEventArgs : EventArgs
58  {
59      public int Threshold { get; set; }
60      public DateTime TimeReached { get; set; }
61  }
```

Zdroj: (proband D1)

Tab. 5.11 ukazuje strategie, kterými se studenti snažili překonat problémy s uvedeným konceptem, a které uvedli v rozhovorech. Z rozhovorů se zdálo, že konceptu rozumí, avšak z detailní následné analýzy výsledných kódů, bylo patrné, že příslušný koncept nebyl detailně pochopen a studenti nebyli schopni správně aplikovat koncept ve svých programových úlohách.

Tabulka 5.11: Přístup a strategie studentů pro překonání problému s konceptem Událost

Použitá strategie	Přístup	Procento studentů využívajících danou strategii
Učí se od ostatních lidí	Učí se od spolužáků	78,9
	Získá pomoc od učitele	10,5
Učí se z psaných materiálů	Čte instrukce z knihy	10,5
	Čte instrukce z internetu	84,2
Učí se z příkladů	Používá vlastní příklady	10,5
	Hledá příklady na internetu	84,2
Užití jiného přístupu	Pokus a omyl	5,3
	Vyhnutí se problému	57,9
	Udělá si přestávku a zkusí řešit po přestávce	0

Zdroj: autor

Na rozdíl od strategií, kterými překonávaly obtíže s předchozími koncepty,

a které vedly k úspěšnému porozumění, v případě tohoto konceptu uvedené strategie většinou nevedly k úspěšnému porozumění a pochopení, což se projevilo zejména v nesprávném řešení programových úloh. Tyto úlohy sledovaly pochopení a aplikaci konceptu Událost.

Ze sledovaných probandů se ve třech případech podařilo studentům si koncept osvojit (viz příklad probanda D1:

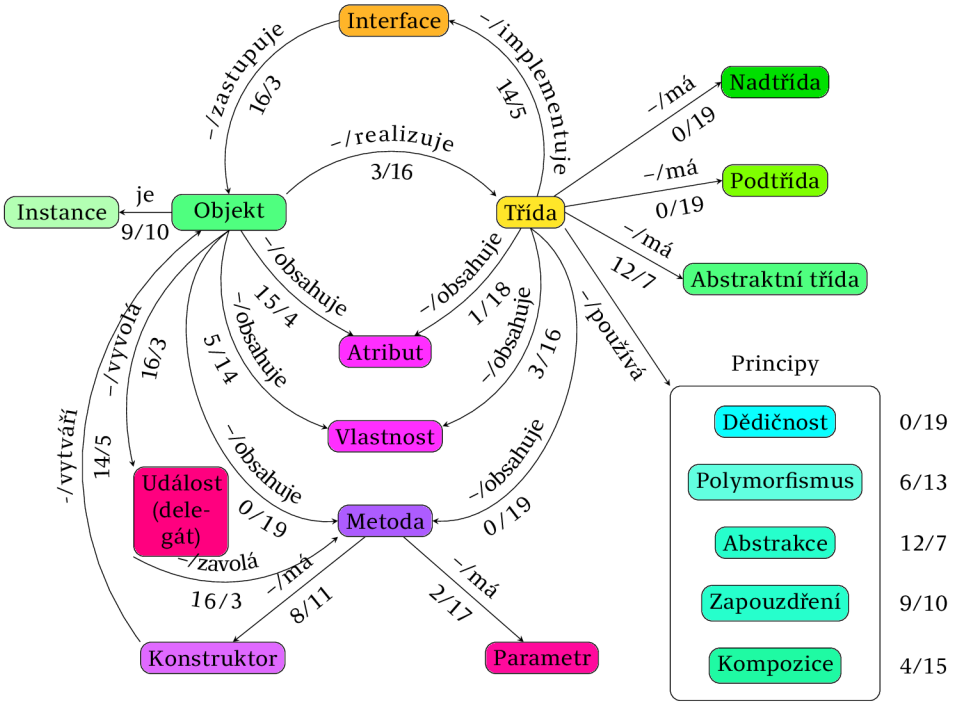
Jeden ze studentů, který úspěšně ovládl koncept, využil znalosti z předchozího studia a dva studenti úspěšně uplatnili strategii *učení se od ostatních lidí*, kdy docházeli na pravidelné *konzultace k vyučujícímu*. Tito měli silnou vnitřní motivaci se naučit dobře naučit paradigma OOP.

5.3 Porozumění objektově orientovaným konceptům

Jako další krok pro porozumění, jak studenti chápou základní koncepty objektově orientovaného programování, jsme zkoumali za pomocí konceptuálních map. Jedná se o výzkum týkající se výzkumné otázky RQ3, která se týká chápání studentů základním konceptům objektově orientovaného programování a vztahům mezi těmito koncepty. Výzkum za použití konceptuálních map navazuje na výzkumné otázky RQ1 a RQ2, kde odpovídající zjištění je popsáno v předchozích kapitolách. Je vhodnou triangulací dat získaných z pozorovní a polostrukturovaných rozhovorů. Výzkum pomocí konceptuání mapy jsme provedli na konci výuky, kdy studenti byli s těmito koncepty a vztahy mezi nimi seznámeni a ve svých úlohách je používali. Cílem bylo získání odpovědí na otázky:

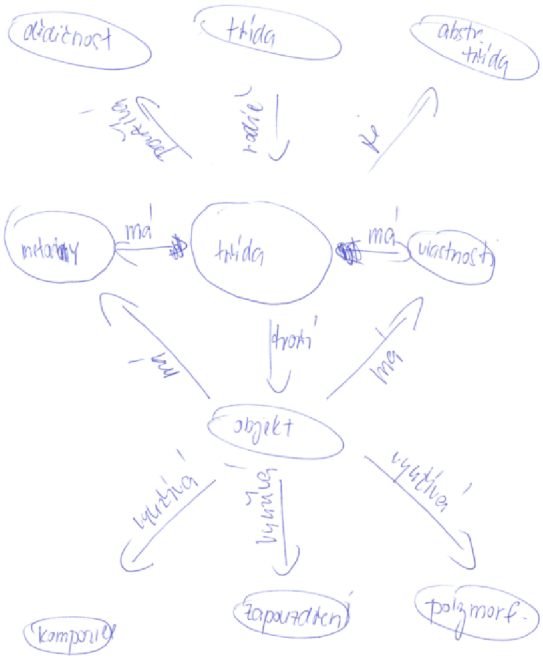
1. Co považují studenti za nejdůležitější koncepty objektově orientovaného programování?
2. Jak vyjadřují vztahy mezi těmito koncepty?

Obě otázky nám poskytnou další obraz, jak chápou a používají studenti základní koncepty objektově orientovaného přístupu v programování, Tento námi



Obrázek 5.13: Agregovaná konceptuální mapa vytvořená z konceptuálních map studentů

Zdroj: autor



Obrázek 5.14: Příklad konceptuální mapy vytvořený studentem

Zdroj: (proband F1)

použitý přístup vychází z několika předchozích projektů použití konceptuáních map pro porozumění pojmům z různých oblastí, například Sanders a kol. (2008), kteří použili tento přístup na zjištění porozumění pojmů z programování.

Studenti byli dopředu seznámeni s tvorbou konceptuáních map. Účastníci tohoto výzkumu tvořili stejnou skupinu devatenácti studentů jako u předchozích podkapitol. V rámci našeho výzkumu dostali úlohu:

Vytvořte konceptuální mapu, která začíná dvěma koncepty „třída“ a „objekt“. Doplňte další koncepty a hranami se šipkou označte vztahy, popište každý vztah jedním nebo maximálně několika slovy mezi nimi vždy ke každé hraně. Nakreslete konceptuální mapu a zahrňte všechny koncepty objektově orientovaného přístupu k programování tak, jak jste se to naučili včetně vztahů mezi těmito koncepty.

Při hledání konceptů a případně mylných představ o konceptech objektově orientovaného programování jsme použili kvalitativní techniky. Zkoumali jsme kvalitativní podrobnosti - zda byly koncepty implicitně přítomny na mapě, jak jsou propojeny skupiny různých souvisejících konceptů atd. Někde bylo potřeba normalizovat názvy vztahů, protože studenti je někdy popisovali různými synonymy, např. „vytváří“, „formuje“ a „může vytvářet“ bylo nahrazeno jednotným „vytváří“ (obr. 5.13 a obr. 5.14).

Obr. 5.13 shrnuje některé strukturální informace o mapách, které se zobrazují jednoduše jako grafy. Tabulka 5.12 uvádí přehledně seznam vybraných konceptů objektově orientovaného přístupu, a procento map, ve kterých byly nalezeny. Koncept nebo vztah mezi koncepty jsme považovali za správně zakreslený, pokud danému uzlu nebo hraně byl přiřazen určitý popisek, který bylo možno normalizovat v rámci našeho zpracování.

Tabulka 5.12: Počty výskytů OOP konceptů v konceptuálních mapách studentů

Koncept	Počet výskytů v mapách studentů	Procento výskytu
Metoda	19	100
Parametr	17	89,4
Instance	13	68,4
Atribut – Třída	18	94,7
Atribut – Objekt	4	21,1
Dědičnost	19	100
Polymorfismus	13	68,4
Abstrakce	4	21,1
Zapouzdření	10	52,6
Kompozice	15	78,9
Konstruktor - Objekt	5	26,3
Konstruktor - Metoda	11	57,9
Rozhraní (Interface) - Objekt	3	15,9
Rozhraní - Třída	5	26,3
Abstraktní třída	7	36,8
Událost	3	15,9

Zdroj: autor

Konceptuální mapy naznačují, že ne všichni studenti plně pochopili koncepci objektu a instance. Pouze třináct studentů uvedlo pojem „Instance“ a správný vztah mezi pojmem „Objekt“ a „Instance“. Nicméně, jeden ze studentů, který v konceptuálních mapách správně uvedl instanci a vztah k objektu, uvedl v rozhovoru „vlastně si nejsem jistý, co přesně instance znamená ve vztahu k objektu“. Dva studenti pojem „Instance“ ve svých konceptuálních mapách úplně vynechali, čtyři studenti neměli vyznačen správný vztah. Tito studenti uvedli, že „Třída má Instanci“, nebo že „Instance je součástí třídy“, nebo „Instance může být Třída“, nebo „Instance je kopie Objektu“. To znamená, že studenti neporozuměli správně konceptům „Objekt“ a „Instance“ a zejména vztahům mezi těmito pojmy. Ve skutečnosti Objekt a Instance často ukazuje na stejný koncept. Oba názvy se ale mohou vyskytnout v různém kontextu, což může být pro studenty, kteří nemají dobře osvojeny principy OOP, matoucí.

Na druhou stranu se zdá, že studenti chápou, že „Třída“ zahrnuje jak data, tak chování. Nezdá se, že by měli mylnou představu o datech jako vlastnostech „Třídy“. Jejich mapy se více zaměřují na behaviorální aspekt tříd než na jejich data. To je vidět z toho, že všichni studenti ve svých konceptuálních mapách měli koncept „Metoda“ a správný vztah mezi „Objektem“ – „Metodou“ a „Třídou“ – „Metodou“.

„Zapouzdření“, „Dědičnost“ a „Polymorfismus“ jsou další důležité rysy objektově orientovaného programování. „Abstrakce“ je základem pro všechny výpočty, předávání zpráv odráží klasický pohled na objektově orientovaný program jako sadu komunikujících objektů.

„Zapouzdření“ jsme považovali za přijatelné na jakékoli konceptuální mapě studentů, která obsahovala něco o viditelnosti objektu, zejména mapy, které zahrnovaly popis jako „soukromé“ a „veřejné“ spojené s „Atributem“, „Metodou“ nebo s „Třídou“. Zhruba polovina studentů zmínila správně tento koncept a popis. Pokud například mapa, která obsahovala pouze výraz „veřejný“, pak nebyla klasifikována jako mapa se správným porozuměním pojmu „Zapouzdření“.

Pojem „Dědičnost“ se objevil na všech mapách. Studenti tento pojem dobře

zvládli, byli schopni ho popsat i v rozhovorech a dobře ho v základu používat při návrhu a implementaci svých úloh.

„Polymorfismus“ se objevil výslovně na 70 procenty mapách. Zde studenti prokázali dobré porozumění tomuto pojmu, např. jeden student spojil hranu „Polymorfismus“ s „Rozhraním“, další popsal „Polymorfismus“: „to může dělat stejné věci různými způsoby (polymorfismus)“.

„Abstrakce“ se objevila jen na čtyřech mapách, jako uzel na konci cesty, ale bez označení např. „používá“ pro hranu „Třída – Abstrakce“. Pokud by se toto označení na konceptuálních mapách vyskytovalo, poukazovalo by to na přeci jen ještě hlubší pochopení konceptu „Abstrakce“ studenty.

Téměř žádná z konceptuálních map studentů neuváděla koncept „Událost“, nebo také „zprávy, které si předávají objekty“. Toto úplné opomenutí odráží neznalost důležitého přístupu objektově orientovaného přístupu. Pouze tři studenti uvedli zmínku „Delegát“, respektive „reference na metodu“, což jsme ve výsledku po upřesňujícím rozhovoru vyhodnotili jako pochopení konceptu.

6 Diskuze a dopady

Náš výzkum se týká porozumění a způsobů, jak se studenti učí a zvládají základní koncepty v oblasti objektově orientovaného programování. Sledovali jsme také způsoby, jakými studenti překonávají problémy, případně jak prožívají proces učení a používání daných konceptů a jevů. Veškerý výzkum a výsledky jsou založeny na empirickém výzkumu, na kvalitativní analýze. Náš výzkum se týkal otázky, zda existují takové koncepty v oblasti objektově orientovaného programování, jejichž zvládnutí významně posunuje dovednosti studentů v oblasti programování, jaké jsou tyto koncepty a jakým způsobem se je studenti učí. Zvolili jsme rámec založený na teorii prahových konceptů (Meyer a Land, 2003), protože nás zaujala myšlenka možnosti identifikace složitých, transformativních, konceptů v objektově orientovaném programování (Rountree, 2009; Sorva, 2013). Tato metoda se nám zdála být velmi vhodná pro náš cíl zlepšit výuku programování.

Tato kapitola tedy poskytuje určitý přehled a diskuzi výsledků našeho výzkumu popsaného v předchozí kapitole. Uvádíme zde teoretické dopady tohoto výzkumu a také praktické přínosy pro pedagogickou praxi. Součástí kapitoly je i popsání limitů naší práce a nastínění možnosti budoucího výzkumu.

6.1 Prahové koncepty v objektově orientovaném programování

Tato kapitola navazuje na zjištění, které se týkalo výzkumné otázky RQ1 – Jaké koncepty objektově orientovaného přístupu k programování jsou pro studenty obtížné a zejména transformativní dle teorie prahových konceptů (Meyer and Land, 2005). Cílem, jak jsme již uvedli, bylo porozumění toho, které koncepty OOP jsou pro studenty nejen obtížné, ale i transformativní. Někteří autoři, kteří se zabývali prahovými koncepty v počítačových vědách, konstatovali například,

že:

„ *Na základě těchto šetření předkládáme důkazy, že prahové koncepty v počítačové vědě existují. Identifikovali jsme dva prahové koncepty, nebo možná široké oblasti, v nichž prahové hodnoty existují: ukazatele a objektově orientované programování.* “ (Boustedt a kol., 2007)

Nicméně takový závěr pro návrh výuky není praktický. Proto jsme si dali za cíl zkoumat, zda můžeme identifikovat prahové koncepty v jednotlivých konceptech objektově orientovaného přístupu k programování. Využití charakteristik dle teorie prahových konceptů, zejména problematických a transformativních, by mělo učinit počáteční identifikaci prahových konceptů relativně jednoduchou. Identifikace konceptů s těmito vlastnostmi v datech cílové skupiny však byla náročnější, než jsme původně očekávali:

(1) Identifikace problematických konceptů. V rozhovorech v mnoha případech účastníci přímo popsali koncepty, se kterými měli problémy. V rámci kódování jsme takovým konceptům přiřadili charakteristiku *problematické*. Nicméně v některých případech studenti zpočátku nepopisovali určitý koncept jako problematický, ale následné diskuse a analýzy výsledků jejich návrhů programů a kódů odhalily, že konceptu porozuměli jen částečně. Zdá se, že důležitým důvodem je to, že problematické koncepty často vedou k částečnému porozumění, které brání účastníkům aktivně identifikovat jejich problematickou povahu. Z analýzy rozhovorů a dokumentů vyvozujeme, že pokud student porozumí určitému konceptu jen částečně, může to být silným indikátorem toho, že se tento student nachází v liminálním prostoru. To znamená, že tento koncept je skutečně dostatečně problematický, aby mohl být kandidátem na **prahový koncept**. A naopak, jakmile účastníci projdou liminálním prostorem, bude pro ně obtížné si pamatovat, že koncept byl někdy obtížný.

(2) Identifikace transformativních konceptů. Yeomans a kol. (2019) popisují určité problémy s identifikací transformativních konceptů, uvádějí:

„ „*Existují určité důkazy o transformativním účinku určitých konceptů*

> *v našich datech - například emoční reakce, kterou někteří účastníci projevili při hlášení zvládnutí úkolu spojeného s konkrétním konceptem. Lze však zpochybnit, zda jsou tyto transformace skutečně tak podstatné, jak naznačuje definice prahových pojmů: Koncept prahu lze považovat za podobný portálu, který otevírá nový a dříve nepřístupný způsob přemýšlení o něčem. Představuje transformovaný způsob porozumění nebo interpretace nebo prohlížení něčeho, bez čeho nemůže žák pokročit“.“* (Yeomans a kol., 2019)

Je zřejmé i na základě literárních zdrojů, že v různých oborech existuje jen málo konceptů, které by měly podstatný transformativní účinek. Takové koncepty mají pak podstatný dopad na schopnost studenta zvládnout dobře daný obor, v našem případě objektově orientované programování. Přestože Yeomans a kol. (2019) uvádějí problémy při identifikaci transformativních konceptů na základě rozhovorů, v rámci našeho výzkumu se domníváme, že analýza dokumentů a analýza výsledků mnoha programátorských úloh u zkoumané skupiny studentů nám umožnila tyto prahové koncepty správně identifikovat. Je třeba poznamenat, že jsme identifikovali další problematické koncepty, které měly ale menší transformativní účinek, proto jsme je nezařadili do prahových konceptů. Pro návrh výuky však i tyto koncepty mohou být stejně důležité. Je třeba, aby jim byla ve výuce věnována patřičná pozornost.

Jak již bylo zmíněno na začátku této kapitoly, v literatuře se diskutovalo o vhodnosti celého objektového přístupu jako kandidáta prahového konceptu pro oblast programování. Nicméně, někteří autoři uváděli, že tento prahový koncept by pokrýval příliš širokou oblast a že by jeho praktická hodnota pro výuku byla nízká. Někteří autoři navrhují v rámci výzkumů objektově orientovaného přístupu jiné prahové koncepty. Například Yeomans a kol. (2019) navrhují jako prahové koncepty „Třída a dědičnost“ a „Abstraktní třída“. Další například Sanders a McCartney (2016) navrhují jako prahové koncepty „Dědičnost“, „Polymorfismus“, „Interakci objektu“ a „Objekty“. Eckerdal a kol. (2006) uvádí, že „Objekty“ a „Třídy“ jsou

pouze problematickými dílčími koncepty objektového přístupu k programování než samy o sobě prahovými koncepty.

Podle našeho názoru je celý objektově orientovaný přístup k programování příliš velký na to, aby byl užitečným kandidátem prahového konceptu. V našem výzkumu jsme se soustředili na jednotlivé koncepty OOP a na základě rozhovorů a analýzy dokumentace jsme jim přiřazovali jednotlivé charakteristiky podle teorie prahových konceptů (Meyer a Land, 2003, 2005). Prahové koncepty identifikované naším výzkumem mají menší granularitu než celý OOP přístup, a i když identifikované prahové koncepty přinášejí stále značné potíže ve výuce, poskytují dostatečnou transformaci v chápání studentů, aby byly užitečnější jako základ pedagogických intervencí.

V rámci výzkumu jsme zkoumali zkušenosti studentů s jednotlivými koncepty objektově orientovaného programování, zejména jsme se v rámci polostrukturovaných rozhovorů snažili zjistit, které koncepty jsou problematické nebo mají jiné charakteristiky, zejména ty, které vedly k určité formě transformace v programování. Shromažďovali jsme údaje od studentů, rozhovorů, pozorování a analýz dokumentů, návrhů a programových kódů. Dále jsme analyzovali shromážděná data z rozhovorů a dokumentů, abychom zjistili, jakými způsoby se studenti transformovali, pokud začali myslet, jednat a identifikovat se více jako programátor, pokud reagují na své vlastní znalosti a které z konceptů jsou ty, které je transformovaly. Jednotlivé charakteristiky OOP konceptů jsme tedy identifikovali následovně:

Syntaktické elementy

Jedná se o základní struktury a pravidla, která jsou určena konkrétním programovacím jazykem. Jedná se například o způsoby deklarace proměnných, používání předdefinovaných datových typů, práce s operátory, způsob práce s řídícími strukturami (např. cykly, větvení), ale též způsob práce s různými abstraktními programovými strukturami.

Konceptu Syntaktické elementy jsme přiřadili charakteristiky: *nezvratný*, *integrativní*, *ohraničený* a *diskurzivní*.

Celkově jsme nezaznamenali závažnější problémy u studentů s porozuměním a aplikací tohoto konceptu při řešení zadaných úloh. Tento koncept nepovažujeme za problematický, považujeme ho za diskurzivní a ohraničený, protože studenti zvládli základní terminologie specifické oblasti programování. Sudenti se vyjadřovali, např.

„Se syntaxí jsme ve třídě neměli velké problémy. Někdy se musím podívat na internet, abych si připoměl některé prvky v programovacím jazyce, které jsem dlouho nepoužil, ale jinak dobrý. "

Tento koncept jsme podle SOLO taxonomie (Biggs a Tang, 2011) zařadili do úrovně *unistrukturální*, protože zahrnuje vcelku jednoduché pojmy a operace. Na této úrovni studenti nemají problémy s porozuměním a používáním odpovídajících pojmů a postupů.

Třída

Schopnost reprezentovat návrhová řešení pomocí grafických notací je důležitou dovedností v profesionálních kontextech. Vyžaduje porozumění objektově orientovaných konceptů, a také tomu, jak se projevují v modelovacím jazyce. Zkoumali jsme také v rámci výzkumu porozumění konceptu Třída, jak studenti zažívají třídní diagramy v notaci používané v UML (Uni ed Modeling Language). To umožnil modul pro kreslení Diagramu tříd, který je součástí vývojového prostředí MS Visual Studio.

Studenti dostali sadu třídních diagramů a v rozhovoru byly tyto třídní diagramy diskutovány různými způsoby. Analýza poskytla výsledky shrnuté níže.

„Diagramy tříd se používají jako dokumentace stávajícího programového kódu, což je dobré pro někoho, kdo se chce dozvědět více o softwaru a možná provede změny v programu. "

„Diagramy tříd se používají jako způsob vývoje návrhů softwaru. Je to nástroj,

který lze použít k modelování řešení problému, a proto také dokumentuje existující nebo neexistující programový kód. "

> *„Diagramy tříd se používají jako způsob navrhování softwaru a slouží jako prostředek pro dialog se členy týmu v procesu dynamického návrhu, který skončí v programovém kódu. Jsou perfektní k použití jako nástroj k vývoji, diskusi a testování modelů na tabuli společně s týmem. "*

Také způsoby vidění diagramů tříd samy o sobě byly rozděleny do tří kategorií. Prostým způsobem lze popsat diagramy tříd jako diagram, který zobrazuje třídy a připojení. Sofistikovanějším způsobem je možné vidět diagram tříd jako diagram, který ukazuje, jakým způsobem jsou třídy příbuzné. V nejpokročilejší kategorii je diagram tříd popsán jako diagram, který ukazuje, jak třídy vzájemně souvisejí v hierarchiích a strukturách.

Třída je uživatelský datový typ. Jedná se o předlohu (šablonu), podle které bude následně objekt vytvořen. Základními členy třídy jsou, shodně s objektem, data (atributy) a metody (funkce definované v kontextu třídy).

Na základě rozhovorů, vyjádření studentů, jejich reakcí při řešení úkolů a analýzy získané dokumentace přiřazujeme tomuto konceptu charakteristiky: *integrativní*, *ohraničený* a *diskurzivní*.

> *„J1: Vytvořím si v class diagramu novou třídu. Pojmenuju jí.*
>
> *V: Proč J1 vytváříte novou třídu?*
>
> *J1: Protože chci vytvořit toho bojovníka tak potřebuju si ho nějak. Potřebuju nějakou šablonu podle, který bude vznikat. "* (J1 – proband, V – výzkumník)

Proband J1 ukazuje v ukázce správné porozumění tomuto konceptu, podobně ale uvažovali i další studenti. Koncept Třída se na základě našeho výzkumu neukazuje jako problematický, i když někteří autoři jej označují v některých situacích spíše jako problematický, např. Kölling (1999) nebo Lister a kol. (2004).

Objekt a Instance objektu

Tento koncept jsme charakterizovali následovně: Objekt je základní stavební entitou v objektově orientovaného programování. V objektu jsou obsažena data a jejich související funkčnost. Data jsou označována jako atributy nebo vlastnosti, jejich funkčnost označujeme jako operace nebo metody. Instance je objekt, který již má alokovaný prostor v paměti, vytváří se pomocí zavolání konstruktoru třídy.

Tento koncept můžeme charakterizovat podobně jako koncept Třída, tedy jeho charakteristiky jsou: *nezvratný*, *integrativní* a *diskurzivní*.

Koncept Objekt a Instance objektu jsme necharakterizovali na základě rozhovorů se studentu jako problematický. Proband M2:

„Docela mě to všechno zapadlo, třída objekt, a teď to všechno používám, “

Naše analýzy rozhovorů a dokumentů vcelku potvrzují slova Pecinovského (2008), který ve svém příspěvku uvádí:

„„OOP přineslo do programování další výrazné zmenšení sémantické mezery mezi způsobem popisu problému běžným člověkem a způsobem jeho popisu v analýze a následně v programu“. “ (Pecinovský, 2008)

Po pochopení konceptů Třída a Objekt byly studenti schopni formulovat určitý problém pomocí těchto konceptů s následným vytvořením určité programové aplikace.

Kompozice

Popisuje vztah mezi třídami, který lze sestavit z jednoho nebo více objektů jiných tříd v atributech instance. Používá se vazba „obsahuje“ nebo „má“. Tento koncept dle taxonomie SOLO řadíme do úrovně *relační*. Na základě rozhovorů a analýz jsme tomuto konceptu přiřadili charakteristiky: *problematický*, *integrativní*, *diskurzivní* a *liminální*.

Tento koncept je na pochopení a na používání složitější než předchozí koncepty. Koncept vyžaduje integraci předchozích konceptů, vytvoření vztahů mezi

různými konstrukty. Zvláště to, že koncept Kompozice umožňuje vytvoření vztahů mezi různými konstrukty, může vést k určitému „zmatku“ ve správném používání tohoto konceptu v úlohách, což je nakonec asi poměrně objektivní míra, že studenti dobře porozuměli určitému konceptu.

„P1: Ja by som teda začal klasicky s class diagramom a, vlastne my sme sa o tom předtím rozprávali a ja by som teda vytvoril dve triedy, takže tam dávam tie dve triedy. “ (Pecinovský, 2008)

V této ukázce přemýšlení nahlas to bylo zřejmé, student měl použít kompozici, ale místo toho stále vytvářel další nové třídy. Koncepty Třída a Objekt dobře chápal, pochopení konceptu Kompozice mu dělala problémy. Stejný problém s tímto konceptem vykazovali i další studenti. Tento poznatek je ve shodě s autory Donchev a Todorova (2008), kteří uvádějí, že studenti mohou mít: „Potíže při budování a pochopení vztahů mezi třídami a jejich vzájemným ovlivňováním“, (strana 165).

Dědičnost

Znamená, že každá třída může mít své “dítě”, tedy třídu, která je od ní odvozená, přebírá všechny její data a metody (Asagba a Ogheneovo, 2008). Tomuto konceptu jsme přiřadili tyto charakteristiky, integrativní a diskurzivní. Charakteristický způsob, jak konceptu Dědičnost porozuměla většina studentů, vyjádřil probant M1:

„najednou se mi to dalo dohromady, pochopil jsem, proč jsme učili třídu a objekt, dědičnost mi moc problémů nedělal s pochopením, je to něco co bych řekl, že je přirozené. “ (Pecinovský, 2008)

Zde jsme na rozdíl od autorů Liberman a kol. (2011) v našem výzkumu nezjistili, že by studenti s konceptem Dědičnost měli nějaké významné problémy.

Polymorfismus

Polymorfismem označujeme vlastnost, kdy je možno jedno jediné jméno použít pro dva nebo více souvisejících, ale technicky různých účelů. Polymorfismus tak umožňuje určit jedním jménem celou obecnou třídu procesů. Uvnitř obecné třídy procesů je pak volba konkrétního procesu dána typem dat. Tomuto konceptu jsme přiřadili charakteristiky *problematický*, *integrativní* a *diskurzivní*. V kapitole 5 uvádíme příklady, kdy student neporozuměl správně tomuto konceptu, viz 5.4. I když studenti tento prpoblém nakonec překonali, můžeme ve shodě s autory Liberman a kol. (2011) konstatovat, že tento koncept je problematický a vyžaduje, aby vyučující tomuto konceptu věnoval značnou pozornost, jak doporučují i zmínění autoři (Liberman a kol., 2011).

Zapouzdření

Tento koncept jsme popsali jako proces skrývání všech vnitřních detailů objektu před vnějším světem. Skrýváme data, které činíme soukromými, a vystavujeme veřejné vlastnosti pro přístup k těmto datům z vnějšku. Konceptu Zapouzdření jsme přiřadili charakteristiky *nezvratný*, *integrativní* a *diskurzivní*. Jak uvádíme v kapitole 5, studenti koncept rychle zvládli. Chyby, které dělali na začátku (např. časté použití veřejných deklarací datových polí /atributů/), již po vyřešení jedné nebo dvou úloh, které tento koncept zahrnovaly, neopakovali.

Abstrakce (abstraktní třída)

Koncept Abstrakce (abstraktní třída) jsme definovali jako proces skrývání stylu práce s objektem a zobrazování užitečných informací, které jsou nutné k pochopení objektu. Při navrhování aplikací je důležité vědět, kdy použít abstraktní třídu a kdy použít rozhraní.

Prvotní analýza dotazníků a neformální rozhovory s učitelem například naznačovaly, že abstrakce by mohla být pravděpodobným kandidátem na prahovou

koncepci. V polostrukturovaných rozhovorech jsme zkoumali podrobněji, jakým konkrétním způsobem studenti chápou pojem „Abstrakce“. Analýzami jsme potom zjistili, že studenti sice diskutovali o různých formách abstrakce, ale pouze v pěti případech byla abstrakce uvedena ve správném kontextu. Nicméně, následnou analýzou jsme museli konstatovat, že nemáme žádné důkazy, které by prokazovaly, že abstrakce sama o sobě je prahovým konceptem, i když je to klíčový koncept a nezbytná znalost.

Tomuto konceptu jsme přiřadili charakteristiky *problematický*, *integrativní* a *diskurzivní*. Koncept se tedy ukázal jako problematický, studentům dalo poměrně hodně práce, když chtěli porozumět konceptu Abstrakce, když mu chtěli porozumět do hloubky a používat ho ve svých úlohách. Např. při řešení úlohy, která při správném návrhu vyžaduje použití abstraktní třídy z 19 studentů na začátku, to znamená po odpřednášení látky, 14 studentů ve svém návrhu představilo pouze konkrétní třídy, které byly zmíněny v zadání. Tito studenti poté pracovali a po několika (neúspěšných) pokusech o nalezení vhodných dědičných vztahů vyřešili problém v dalších příkladech. Ačkoli ne všechna řešení byla úplná a přesná, naznačovala, že studenti se nakonec dobře vypořádali s konceptem abstraktní třídy a byli si vědomi základních situací, ve kterých by to mohlo být užitečné.

Konceptu Abstrakce (abstraktní třída) jsme tedy přiřadili charakteristiku problematický. To, že studenti mají s tímto konceptem problém, popisuje také např. Hadar (2013).

Rozhraní

Rozhraní je koncept, který jsme identifikovali jako *prahový* koncept. Přiřadili jsme mu charakteristiky: *transformativní*, *problematický*, *nezvratný*, *integrativní*, *diskurzivní* a *liminální*.

Pokud studenti zvládnou tento koncept, jejich schopnost programování se povýší na novou úroveň. U tohoto konceptu jsme identifikovali všechny charakteristiky teorie prahových konceptů. Jedná se o vysoce abstraktní pojem, který stu-

dentům dělá velké problémy.

Studenti k rozhraní uváděli:

„Rozhraní je seznam úkolů“.

„Rozhraní je text ve formě seznamu úkolů, který říká programátorovi, co má dělat; jaké operace by měl napsat. Jedná se o nedokončený program, kód kostry nebo šablonu, která má začít od okamžiku, kdy by měla být napsána nová třída“.

„Rozhraní je určitě definováno textem; text však představuje abstraktní „věc“, kterou lze vázat na třídu odkazem na název rozhraní. Třída je tak povinna mít implementace pro všechny operace specifikované rozhraním. Tímto způsobem se rozhraní stává vynucenou smlouvou. Programátor musí implementovat rozhraní a rozhraní má název“.

„Rozhraní je datovým typem pro referenční proměnné, a tedy nepřímo pro objekty. Rozhraní, definované textem, má název pro datový typ, který představuje. Datový typ lze použít k vytvoření proměnných, které zvládnou ty objekty, které odpovídají deklaraci obsahu. Toto je výraz smysluplného vztahu mezi rozhraním, třídou a objektem“.

„Rozhraní je otevřené připojení k novým a neznámým objektům. Účelem popisovačů typu rozhraní je, že představují otevřené připojení k libovolným objektům, které implementují stejné rozhraní. Stejný popisovač se tedy může připojit k několika objektům, definovaným různými třídami. Podle popisů je možné objekty vyměnit a používat různé typy objektů, aniž byste museli měnit zbytek softwaru. Používání rozhraní umožňuje objektům vzájemně komunikovat, i když jsou „cizinci“.

Definici „Rozhraní“ studenti do určité míry zvládali. Avšak analýzou jejich návrhů, kódů bylo zřejmé, že se použití tohoto konceptu vyhýbají a používají např. konstrukt seznam místo rozhraní.

Problémy s popisem použití rozhraní popisuje např. Lavy a kol.(2009):

„]“ (Lavy a kol., 2009)[]*Zaměřili jsme se na zkoumání demonstrace úrovní abstrakce studenty při návrhu hierarchie tříd obecně a na to, zda konkrétně používají*

třídy rozhraní. Výsledky studie ukazují, že většina studentů dokázala vybudovat rozumnou třídní hierarchii; mnoho z nich však nepoužívalo třídy rozhraní jako nástroj pro vyjádření běžného chování.

Většina studentů koncept Rozhraní plně nepochopila. Pouze 3 studenti vzkazovali při řešení svých úloh výsledky, které naznačovali dobré pochopení tohoto tématu. Je zajímavé, že v rozhovorech studenti poskytli rozumné definice pojmu „Rozhraní". Ačkoli definice nebyly přesné, přeci jen se zdálo, že tito studenti byli obeznámeni s konceptem „Rozhraní" a byli si vědomi základních situací, ve kterých by to mohlo být užitečné. Avšak úsilí, které investovali do řešení problému bez použití abstraktní třídy (což bylo zřejmé po analýze jejich programovacích úloh), naznačuje, že ačkoli mají základní znalosti o konceptu, mají potíže s jeho použitím. To znamená, že chybí hlubší porozumění, můžeme toto chování označit jako „mimikry", studenti se pohybují v liminálním prostoru.

U několika studentů, kteří zvládli OOP koncept rozhraní, bylo vidět, že se dostali na jinou úroveň v programování. Byli schopni využít všech předchozích popisovaných OOP konceptů, modelovat a vytvářet opravdu komplexní aplikace. Bylo také vidět, že si uvědomují, že zvládli předmět Programování a bylo vidět i, že měli z činností spojených s řešením programovacích úloh a návrhem i kódováním radost.

Událost

Události jsme definovali, jako procesy, které umožňují třídě nebo objektu upozornit jiné třídy nebo objekty, když dojde k nějakému akci. V jazyku C♯ se pro systém událostí používají delegáti. Tomuto kocenptu jsme přiřadili charkteristiky: *tranformativní*, *problematický*, *nezvratný*, *integrativní*, *diskurzivní* a *liminální*.

Většina studentů však koncept Událost plně nepochopila. Je opět zajímavé, že v rozhovorech studenti poskytli rozumné definice pojmu „Událost". Ačkoli definice zase nebyly přesné, přeci jen se zdálo, že tito studenti byli obeznámeni s konceptem „Událost" a byli si vědomi základních situací, kde by to mohlo být užitečˇ

né. Avšak úsilí, které investovali do řešení problému bez použití tohoto konceptů (což bylo zřejmé po analýze jejich programovacích úloh), naznačuje, že ačkoli mají základní znalosti o konceptu Událost, mají potíže s jeho použitím. To znamená, že chybí hlubší porozumění, můžeme toto chování označit jako „mimikry", studenti se pohybují v liminálním prostoru. Koncept je tedy problematický, jak to popisují v literatuře různí autoři, např. Kim a kol. (2001) a další.

Koncept Událost zvládli dobře i podle analýz dokumentace 3 studenti. Jeden ze studentů, který úspěšně ovládl koncept, využil znalosti z předchozího studia a dva studenti úspěšně uplatnili strategii učení se od ostatních lidí, kdy docházeli na pravidelné konzultace k vyučujícímu. Tito měli silnou vnitřní motivaci se naučit dobře naučit paradigma OOP. Podobně jako u OOP konceptu Rozhraní můžeme konstatovat, že u studentů, kteří zvládli koncept Událost, bylo vidět, že se posunuli na další úroveň v programování. Byli schopni využít všech předchozích popisovaných OOP konceptů, modelovat a vytvářet opravdu komplexní aplikace. Bylo také vidět, že si uvědomují, že zvládli předmět Programování a bylo vidět i, že měli z činností spojených s řešením programovacích úloh a návrhem i kódováním radost. Považujeme tento koncept proto za **prahový**.

6.1.1 Shrnutí charakteristik konceptů v objektově orientovaném programování

Výsledky charakteristik OOP konceptů podle teorie prahových konceptů (Meyer a Land, 2003 a 2005) můžeme shrnout následovně:

- Výsledky analýz z rozhovorů a dokumentů ukazují, že studenti často spojují své učení s konkrétními příklady problémů, které řešili, a že z těchto zkušeností učinili zobecnění, které potom popisovali v rozhovorech. Nebo se studenti se naučili nějaký abstraktní pojem, v rozhovorech jej potom celkem přijatelně definovali, ale z výsledků programovacích úloh bylo zřejmé, že studenti koncept plně nepochopili a neimplementovali jej v celém roz-

sahu. Zejména u abstraktnějších konceptů měli problém s návrhem řešení a implementací.

- V našem výzkumu jsme identifikovali dva prahové koncepty, jejichž překonání je pro studenty obtížné, ale po překonání vzrostla jejich schopnost naprogramovat i komplexní úlohy správným způsobem a i uspokojení z takového řešení. Transformace v těchto případech vedly ke změně chování a jeví se, že jsou spojeny spíše s pochopením toho, *proč*, než *co*.
- Identifikované prahové koncepty jsou **Rozhraní** a **Události**.
- Ostatní koncepty, nemůžeme prohlásit za prahové, neboť jsme všechny charakteristiky dle teorie prahových konceptů Meyer a Land (2003, 2005) neidentifikovali.
- Dále jsme identifikovali koncepty, které jsou *problematické*. Jedná se o koncepty: **Abstrakce**, **Polyformismus** a **Kompozice**. Z rozhovorů a analýz dokumentů usuzujeme, že na tyto koncepty je potřeba též zaměřit větší pozornost, neboť množstvím identifikovaných charakteristik se do jisté míry přibližuje k definici, že bychom je mohli označit jako prahové koncepty.
- Koncepty, se kterými studenti neměli velké problémy a poměrně rychle jim porozuměli a byli je schopni aplikovat ve svých úlohách, byly: **Třída**, **Objekt**, **Syntaxe**, **Zapouzdření**, **Dědičnost**.

Je třeba poznamenat, že koncepty objektově orientovaného programování jou do jisté míry kontraintuitivní a studenti se s nimi v jiných předmětech nesetkají. Zvládnutí těchto OOP konceptů vyžaduje od studentů vysokou míru kompetencí. Kromě osvojení základních pojmů a kompetencí v algoritmické, analytické a aplikační rovině, student musí též při aplikaci těchto konceptů být schopen analýzy, přemýšlení o modelované situace, klasifikovat, klást si otázky a umět zobecňovat. Jak již jsme také uvedli, většina studentů je shopna uvažovat o konkrétních problémech. Většině studentům však chybí kompetence související s abstraktním uvažováním a zobezňováním naučených postupů a pojmů.

6.2 Jaké strategie používají studenti ve výuce programování při překonávání těžkostí u problematických konceptů

Tato kapitola souvisí s výzkumnou otázkou RQ2: Jaké strategie používají studenti ve výuce programování při překonávání těžkostí u problematických konceptů? Je známé, že učení, zvyšování znalostí a dovedností, neprobíhá vždy konstantní rychlostí. Při učení se novým pojmům a dovednostem se studenti setkávají s epistemologickými překážkami (Meyer a Land, 2005). To má ale ten důsledek, že se na určitém problému zadrhnou a nejsou schopni dosáhnout pokroku v učení a porozumění.

V rámci našeho výzkumu jsme zkoumali i způsoby, strategie, které používali studenti při výuce programování při řešení programovacích úloh, aby jednak vyřešili zadanou úlohu, a jednak aby dosáhli pokroku v učení se těmto pojmům a dovednostem. Brali jsme v úvahu takové strategie, které studenti používali, většinou kombinaci různých přístupů a strategií, a které je ve většině případů vedli k úspěšnému vyřešení programovacích úloh.

Jak jsme uvedli již v předchozích kapitolách, data byla shromážděna pomocí polostrukturovaných rozhovorů. Některé otázky v rozhovorech se výslovně zabývaly myšlenkou, kde studenti měli problémy a jak se s těmito problémy vyrovnávali. Jednotlivé strategie jsou uvedeny u jednotlivých konceptů v předchozí kapitole.

Existuje poměrně rozsáhlá literatura o strategiích řešení problémů, například je možno uvést práci Pólya a Conway (2015). Wankat a Oreovicz (2015) shrnují velké množství studií porovnávajících začínající studenty a odborné řešitele problémů. Přestože řešení problémů a konceptuální učení se velmi liší, autoři (Wankat a Oreovicz, 2015) vidí podobnosti. Návody na překonání problémů uvádějí jako „použijte heuristiku“, „vytrvejte“ a „použijte brainstorming“, ale také „buďte

vytrvalí/nepřestávejte“ a „diskutujte“. Poslední dva přístupy mají analogii v našem seznamu strategií používaných studenty při řešení programových úloh.

Kromě sledování strategií při učení, byly zkoumány též strategie studentů při programování, porozumění programu a řešení problémů. Např. Robins a kol. (2003) uvádějí nedostatek vhodných programovacích strategií jako příčinu problémů u začínajících programátorů. Davies (1993) ve své práci hodnotil mimo jiné studie, které se zabývaly strategickými aspekty programovacích dovedností. Píše, že „strategické prvky programovacích dovedností mohou mít v některých případech větší význam než znalost určitých komponent“.

Nejčastější strategie, které jsme identifikovali v rámci našeho výzkumu, byly:

Učení se od ostatních studentů

Tuto strategii používala většina studentů (70 %) při překonávání problematických konceptů. Student P2 uvedl v rozhovorech o abstrakci:

„Furt jsem se na to jen díval a neustále se to pokoušel uchopit, nakonec jsem něco pochopil. Na další jsem se zeptal kolegy, který je docela dobrý v programování. Ten mi to vyjasnil. A pak to šlo, už jsem to chápal. “

Jiný student uvedl ke konceptu Rozhraní vcelku typicky:

„Pokoušel jsem se to pochopit, ptal jsem se kolegy, který to docela používal. Potom dalších, ale ti nevěděli co s tím, tak jsem to nechal. No, ani pak jsem to nepochopil. Nějak jsem ale ty úlohy splácal. “

Učení se z internetu

To byla nejčastější strategie při řešení některých úloh. Po zadání se studenti dívali na internet, zda nenajdou podobné řešení. Pokud nenašli, hledali, co jiného mohou použít, jaké části kódu. Tuto strategii používalo přes 75 % studentů. Lepší bylo, když hledali syntaxi určité funkce nebo knihovny. Tento případ uvádělo 15 % studentů.

Student M2 například uvedl:

„Dycky se nejdřív podívám na internet, jestli tam není něco podobného. Co by se dalo upravit a použít. Ono i to často pomůže, na internetu je všechno. Ale někdy to není k použití. Už jsem se několikrát spálil. “

Strategie pokus – omyl

Velmi častá strategie, kterou použilo přes 20 % studentů. Studenti ji hodně používají v případě, kdy úplně nerozumí určitému OOP konceptu.

Student J1 ji například popisuje v případě konceptu Polymorfismus takto:

„Nevěděl jsem, co s tím. Něco mi trochu utkvělo v paměti, tak jsem to zkusil udělat. No nebylo to ono, musel jsem se pak zeptat souseda. A pak to šlo, dokonce jsem věděl už jak na to. “

Je zřejmé, že studenti nepoužívali při překonávání těžkostí s porozuměním a aplikací OOP konceptů jedinou strategii. Ve většině případů začali například hledat na internetu (Učení se z internetu). Zkusili vytvořit nějaký návrh a podle něj i kód (Strategie pokus – omyl), pokud neuspěli, ptali se sousedů, jiných studentů (Učení se od ostatních studentů), nebo učitele. Zajímavé bylo, že v případě konceptů, které jsme identifikovali jako prahové („Rozhraní“ a Událost), studenti přes použití kombinace všech identifikovaných strategií neuspěli. Student D2:

„Zkoušel jsem, jak to udělat s rozhraním, moc jsem to nepochopil. Něco jsem tam vždy napsal, abych se použití rozhraní vyhnul. Různě jsem se zkoušel ptát sousedů, na internetu a tak, a nic, nevím jak na to. No, ale dá se to obejít. “

Jak jsme uvedli, tyto prahové koncepty („Rozhraní“ a „Událost“) zvládli tři studenti z celkového počtu devatenácti, u kterých jsme strategie zkoumali. Jeden z těchto tří přímo na otázku, jak zvládl tento koncept, řekl:

> *„Dalo mi to dost práce. Programoval jsem už dříve v Javě. Ale až teď jsem to rozhraní celý pochopil. Musel jsem pročíst hodně příkladů. Docela mi pomohlo, že jsem měl příklady na návrhové vzory. Asi to chce hodně příkladů a hodně času nad tím strávit. “*

Žádná ze strategií, které studenti uváděli, nebyla pro nás překvapivá. Hodně z toho, o čem se zmiňovali, jako strategii pro překonávání problémů, bylo sociální. Většina studentů uváděla, že se hodně naučili s pomocí svých kolegů studentů nebo se zeptali svého učitele. Zajímavé bylo, že zdůrazňovali, jak je důležité spojit nové znalosti s něčím, co již dobře znali. Zmínili se také, že mají rádi podrobné instrukce, když se učí něco nového. Jeden student také poznamenal, že je důležitá vytrvalost a dobrá praxe u nějaké softwarové firmy i v pozici začátečníka, tak získají možnost učit se od zkušených programátorů.

Studenti tedy používali velké množství strategií a postupů při překonávání problémů, se kterými se setkávají při učení se programovat. V této souvislosti je třeba zdůraznit pro výuku důležitost sociálních interakcí i aktivní odpovědnosti studentů.

6.3 Jak studenti chápou základní koncepty objektově orientovaného přístupu v programování

Tato kapitola souvisí o výzkumnou otázkou RQ3, kterou jsme zformulovali v kapitole 1.4 následovně: RQ3 – Jak studenti začátečníci při výuce programování chápou základní koncepty objektově orientovaného přístupu v programování? Pomocí konceptuální mapy jsme studovali, jak studenti chápou objektovou orientaci a objektově orientované programování. Konceptuální mapy pro účely porozumění v učení a získání statického obrazu o tom, co studenti vědí, byly použity např. v pracích Nash a kol. (2006), dos Santos et al. (2017) nebo Steyvers a Tenenbaum (2005). Podrobný popis zjištění pomocí analýzy konceptuálních map v rámci našeho výzkumu je uveden v kapitole 5.

Zkoumali jsme, které koncepty OOP studenti zakreslí, a co je důležité, jak tyto koncepty navzájem propojí. Data byla shromážděna tím, že studenti byli požádáni, aby nakreslili konceptuální mapy. Byli instruováni, aby začali vložením termínů „Třída“ a „Objekt“ do mapy, a poté by měli přidat všechny související pojmy, na

které by si mohli vzpomenout, a popsat vztah mezi všemi pojmy pomocí hran, šipek a popisku hrany. Výsledky takové analýzy pomocí konceptuálních map nám ukázaly určitý statický pohled studentů na koncepty objektově orientovaného přístupu. Konceptuální mapy, zejména analýza zakreslených vztahů mezi koncepty nám ukazala, jak studenti se studenti orientují v jednotlivých konceptech OOP a zejména zakreslení vztahů a popisky těchto vztahů nám ukázaly, jak tyto koncepty chápou v určitém celku. Jak píše Novak a Gowin (1984): „Konceptuální mapy jsou založeny na teorii, že si lidé myslí koncepty a že tyto konceptuální mapy slouží k externalizaci těchto konceptů a zlepšení jejich myšlení“.

Konceptuální mapy, které studenti vytvořili, jsme analyzovali a vytvořili jednu agregovanou konceptuální mapu dle postupů popsaných v kapitole 4.3 a kapitole 5.3, viz obr. 5.13.

Ačkoli objektově orientovaný koncept „Dědičnost“ souvisel v řadě případů se „Třídou“, ne ve všech případech studenti zahrnuli do svých konceptuálních map i další ústřední objektově orientované koncepty, jako jsou „Zapouzdření“, „Abstrakce“ a „Polymorfismus“ nebo „Kompozice“. Ukázalo se však, že studenti dobře propojují data i chování s konceptem třídy. Nevidí třídu pouze jako datový kontejner, což naznačovaly některé předchozí výsledky (Holland a kol., 1997). To potvrzuje dobré porozumění konceptu „Třída“ jako základnímu konceptu OOP včetně související dědičnosti.

V rámci hodnocení konceptuálních map studentů se ukázalo, že studenti měli problémy s OOP koncepty „Rozhraní (Interface)“ a „Událost (Delegát)“. Jen tři studenti uvedli tento koncept ve svých konceptuálních mapách. Popis vztahu těchto konceptů k „Objektu“ a „Třídě“, který studenti uváděli, nebyl úplně přesný, ale blížil se podle našeho názoru správnému pochopení. To dokazuje naše správné zařazení těchto dvou konceptů mezi prahové koncepty objektově orientovaného přístupu v programování. Většina programových úkolů těchto studentů byla také bez větších problémů. Tito studenti také popisovali, že zvládnutí těchto konceptů jim hodně pomohlo v pokročení schopnost programovat:

„Dalo mi to docela zabrat a trvalo mi to dlouhou dobu, než jsem přišel na to, jak to funguje. Potom ale jsem rozhraní hodně využíval, hodně mi to pomohlo v řešení úloh z programování. "

Další student uvedl:

„Zvládnutí rozhraní, a potom programování, to je docela jiný level. Programoval jsem už předtím, ale teď teprve jsem to pochopil a jsme tomu rád. "

Obecný dojem byl ale takový, že většina studentů (kromě těchto tří) v rámci svého studia programování dosud nepřekročila tyto prahové koncepty. Neintegrovali plně potřebné složky do porozumění orientace objektu jako celku. Jsou sice schopni napsat jednoduché programy, ale je pro ně problematické komplexnější úlohy správně navrhnout a naprogramovat.

6.4 Omezení výzkumu

Domníváme se, že náš výzkum poskytuje užitečné poznatky o výuce objektově orientovaného programování. Ukázky přemýšlení studentů v rámci řešení programových úloh může být velký přínos naší práce. Využili jsme rámce teorie prahových konceptů (Mayer a Land, 2003, 2005. Domníváme se, že se nám podařilo dosáhnout dobrého porozumění obtížností vybraných OOP konceptů na řadě úloh. Přitom jsme použili sběr dat založených na aktivitách a metody jako polostrukturované rozhovory, metodu myšlení nahlas a konceptuální mapy. Na základě toho se nám podle našeho názoru podařilo vybrat skutečné prahové koncepty v objektovém přístupu v programování.

Zobecnitelnost našich zjištění má však určitá omezení. Některé z nich byly z hlediska našich možností nevyhnutelné, například studenti byli zařazeni do vzorku z jedné instituce a jednalo se o studenty se smíšenými schopnostmi. Jednalo se o studenty prvního ročníku a neověřovali jsme dopředu jejich schopnosti. Dalším omezením bylo, že ověřování našich výzkumných otázek a vlastní výzkum proběhl na základě výuky konceptů obsažených v osnovách kurzu. Aby se

zmírnila předpojatost, kterou to mohlo způsobit, byly provedeny další otevřené rozhovory s účastníky, kde mohli volně zvážit jakýkoli koncept, který by shledali problematickým nebo transformativním. Využití výzkumu v rámci výuky také umožnilo účastníkům diskutovat o tom, proč si mysleli, že je tento koncept vyučován, a umožnilo nám analyzovat, zda konceptu zcela nebo částečně porozuměli.

6.5 Důsledky pro pedagogiku a výzkum koncepce prahových hodnot

Zjištění popsaná v této práci vycházejí z našeho výzkumu s cílem identifikovat vhodné prahové koncepty v rámci výuky objektově orientovaného programování. Prahové koncepty mají pro tuto výuku řadu důsledků. Jsou to klíčové koncepty, kterým musí studenti porozumět, aby se naučili programovat tak, aby byli schopni správně navrhovat a implementovat komplexnější systémy. Nezískání tohoto porozumění a s tím spojený nedostatek pokroku v naučení se programovat může vést k frustraci, špatnému porozumění tomu, jak do sebe různé pojmy zapadají a jak je možno vytvářet různé aplikace, může to vést k tomu, že studenti informatických oborů se po nástupu do praxe programování zcela vyhýbají. Je proto důležité rozumět, jak studneti chápou tyto koncepty, jak se je učí. Je užitečné vědět, že studenti mají tendenci uvíznout na konkrétním konceptu, ale i znát strategie, kterí studenti používají při překonávání obtíží v učení. Vědět například, jaké další OOP koncepty jiný konkrétní prahový koncept integruje, může učiteli poskytnout kontext, ve kterém by se tento prahový koncept mohl efektivně učit (Timmermans a Meyer, 2017).

Davies (2006) uvádí, že pokud si učitelé budou jen myslet, že se studenti naučili určitý prahový koncept, může to způsobit, že studenti se dále budou učit jen povrchně nebo používat „mimikry“:

„Při absenci tohoto porozumění se mohou studenti uchýlit pouze k používání

povrchního učení a k povrchnímu jazyku v naději, že tímto mohou překlenout své skutečné porozumění.“ (Davies, 2006)

Identifikace prahových konceptů mohou poskytovat učitelům určité příležitosti. Mohou pomoci zvládat stále rostoucí osnovy. Například učební osnovy výuky v rámci počítačové vědy vytořené společně organizacemi IEEE a ACM (JTFCC (2013) obsahují 63 základních jednotek, z nichž každá se skládá z několika témat. Pokud dokážeme v rámci osnov identifikovat relativně malý počet prahových konceptů, mohou se učitelé zaměřit na pomoc studentům s těmito koncepty. Prahové koncepty jsou integrativní, proto je může učitel použít k tomu, aby studentům pomohl vidět souvislosti v rámci disciplíny, které překračují hranice jednotlivých kurzů.

V objektově orientovaném programování se porozumění studenta vyvíjí od chápání objektu jako jednoduchého mechanismu, který má některé vlastnosti, po např. rozhraní. Studenti uvedli, že překročení „prahů“ pro ně byl postupný proces, ne nutně „aha“ okamžik.

V průběhu času osnovy výuky programování narůstají. Další pojmy jsou zaváděny s cílem poskytnout studentům další nástroje a dovednosti, které, i když samy o sobě jsou nepopiratelně užitečné (např. naučit se pracovat jako tým nebo používat anonymní vnitřní třídy při psaní obsluhy událostí), jim ve skutečnosti mohou způsobit potíže při chápání prahového konceptu. Například anonymní vnitřní třídy považovali studenti za problematické, protože je spojovali pouze s obsluhami událostí, což je námi identifikovaný prahový koncept. Ve výuce je třeba výuku takových pojmů oddálit, dokud nebude zvládnut hlavní prahový koncept. Další identifikaci prahových konceptů lze zjednodušit učivo pomocí přístupu přístupu „méně je více“, který je doporučován např. Barradellem (2013) nebo Sorvou (2013). To umožní studentům strávit více času výukou konceptů, které mají prioritu, a zajistí, že úspěšně prošli svým „mezním prostorem“. Jak zdůrazňuje Sorva (2013), student, který překročil práh, má lepší předpoklady k tomu, aby se snadněji učil novým souvisejícím pojmům. Tento přístup také ve výuce

používá Pecinovský (2010, 2013a, 2013b, 2015), který na začátku výuky programování probírá OOP koncept „Rozhraní“ (námi identifikovaný prahový koncept) a postupně na tento koncept nabaluje další OOP pojmy. Studenti podle Pecinovského mají dost času konceptu „Rozhraní“ porozumět. Prahové koncepty, které jsou přímo problematické a transformativní, mají potenciálně pozorovatelnější období liminality, takže množství času věnovaného výuce takového konceptu by mělo být větší. Studenti by měli mít zkušenosti s aplikací prahového konceptu v různých kontextech a příležitost vrátit se k nim v pozdější fázi. Při zodpovězení mnoha z těchto otázek by velmi pomohla dlouhodobá studie, která by sledovala rozvíjející se porozumění prahových konceptů prahů studentů a korelovala je s výukou.

6.6 Budoucí výzkum

Na základě našich zjištění a širší literatury o prahových konceptech v programování (Sanders a McCartney, 2016) je možno formulovat další zajímavé otázky pro výzkum. Zejména v současné době je výuka programování na všech úrovních vzdělávání neoddělitelně spjata s vývojovými nástroji (např. MS Visual Studio). Bylo by vhodné prozkoumat, jak naše zjištění mohou ovlivnit výukové strategie používané souběžně s těmito nástroji a jejich návrhové rozhraní. V posledních letech došlo k nárůstu programovacích prostředí, blokových i textových; hodnocení, jak tyto nástroje pomáhají studentům vybudovat konkrétní konceptuální a procedurální porozumění a jejich vzájemný vztah v programování, nejsou však stále dostatečně zkoumána. Je proto důležité zvážit a prozkoumat koncepci a pedagogické principy používané při budování těchto nástrojů, jakož i způsob, jakým jsou ve výuce využívány, a jejich vliv na porozumění studentů. Zkoumání za pomoci teorie prahových konceptů (Meyer a Land, 2003) by možná osvětlilo způsob, jakým lze různé obtížné koncepty řešit pomocí nástrojů podporovaných počítačem. Literatura například naznačuje, že vizuální nástroje mohou transfor-

movat abstrakce do reálných reprezentací (Crews a Butterfield, 2002) a že vývojové diagramy a konceptuální mapy mohou pomoci učitelům sledovat pokroky studentů v učení a studentům budovat konkrétnější konceptuální porozumění v programování (např. Hubwieser a Mühling, 2011; dos Santos et al., 2017). To je zvláště důležité pro integrační část prahových konceptů, protože konceptuální mapy a vývojové diagramy mohou znázorňovat vztahy mezi koncepty a daty. Bude vhodné prozkoumat, jak lze tyto vizualizační nástroje použít, aby pomohly řešit problémy s překonáváním prahových konceptů. Další oblastí, která se zdá slibná pro řešení porozumění studentů, jsou blokové nástroje, protože výzkum naznačuje, že podporují konceptuální porozumění studentů (Weintrop a Wilensky, 2016). Otázkou, kterou je však třeba se zabývat, je, zda bloková prostředí v rámci specifických výukových strategií mohou řešit nedorozumění studentů, mylné představy a poté pokroky studentů při překonávání prahů. Budoucí výzkum by tedy měl dále zkoumat, jak mohou instruktážní strategie a rozhraní a nástroje počítačového programování spolupracovat při vývoji rámce, který může studentům pomoci vyřešit prahové hodnoty a úspěšně projít liminalitou. Další otázky se nabízejí, jak může identifikace prahových konceptů konkrétně formovat obsah učiva a přístupy k výuce objektově orinetovaného programování? Má pořadí, ve kterém se pojmy vyučují, vliv na rozdíl v tom, jak jsou náročné na porozumění? Jak můžeme nejlépe rozhodnout, jaké „pomocné“ koncepty a dovednosti jsou nezbytné k tomu, aby se mohly učit vedle prahových konceptů, a které se dají z obsahu učiva vyjmout? Naučit se dobře programování je poměrně náročné. Požadavky a nástroje se velmi rychle mění, to klade na učitele značné nároky a je třeba neustále zkoumat, jak výuku programování zefektivnit a pro studenty zatraktivnit.

7 Závěr

„Master programmers think of systems
as stories to be told rather than programs to be written.“
(Robert C. Martin, 2009)

Výuka programování je základní disciplína vyučovaná v různých informatických oborech na vysokých školách. V současné době je dominantním paradigmatem ve výuce objektově orientovaný přístup k programování. Naučit se dobře programovat není ale jednoduché. Velké množství studentů vzdá učení se programovat již po úvodním kurzu programování. Cílem naší práce byla odpověď na otázku: Proč je disciplíně programování, se zaměřením na objektově orientovaný přístup, tak obtížné se naučit, jaké jsou zásadní OOP koncepty, které studentům brání nebo naopak umožňují pochopit celkový obraz?

Naše práce se opírá o teorii prahových konceptů (Meyer a Land, 2003, 2005), která umožňuje svými charakteristikami (transformativní, problematický, ireverzibilní, integrativní, ohraničený, diskurzivní, rekonstituční, liminální) určit takové koncepty objektově orientovaného programování, jejichž porozumění vede ke zvládnutí programování a umožňuje pochopení, jak navrhovat a implementovat správně komplexnější systémy. Náš výzkum staví na kvalitativním přístupu. Data z našeho empirického vyšetřování byla analyzována z několika perspektiv, využili jsme fenomenografie a obsahovou analýzu k získání odpovědí na stanovené výzkumné otázky: (1) jaké koncepty objektově orientovaného přístupu k programování jsou pro studenty obtížné a zejména transformativní dle teorie prahových konceptů, (2) jaké strategie používají studenti ve výuce programování při překonávání těžkostí u problematických konceptů, (3) jak studenti začátečníci při výuce programování chápou základní koncepty objektově orientovaného přístupu v programování.

Pro náš výzkum jsme oslovili studenty oboru Ekonomické informatiky, kteří

mají ve svém učebním plánu dva semestry výuky programování. Požádali jsme je o udělení poučeného souhlasu. Provedli jsme s nimi polostrukturované rozhovory, studenti popisovali pomocí metody mluvení nahlas svůj postup při řešení různých zadaných úloh týkajících se návrhu a kódu. Provedli jsme pozorování obrazovek studentů při jejich práci na úkolech, analyzovali jsme návrhy a kódy vztahující se k jednotlivým úlohám a požádali studenty o zakreslení konceptuální mapy, abychom zjistili, které koncepty si pamatují a jak rozumí vztahům mezi nimi. Zkoumali jsme strategie, které studenti používají, aby překonali problémy s porozuměním určitých OOP konceptům. Přitom jsme využili induktivní analýzu. Pro zjišťování prahových konceptů byla využita jak induktivní, tak i deduktivní analýza, při které jsme použili specifické charakteristiky teorie prahových konceptů.

Ve skutečnosti mnoho konceptů v počítačové vědě je skutečně komplikovaných. Naše práce o studijních zkušenostech studentů v rámci výuky programování nám poskytly řadu příkladů konceptů, které byly na základě analýz rozhovorů se studenty a analýz výsledků jejich úloh problematické, těžko integrovatelné a transformativní. Některé koncepty byly velmi specifické a jiné měly zastřešující charakter.

Studenti hovořili hlavně o konceptech souvisejících s programováním, objektovou orientací a návrhem software. Při pozorování a při analýze popisu postupu práce na úloze formou mluvení nahlas jsme viděli, že studenti často při práci s programovacími projekty narazili na problémy. Programové úkoly a řešení problémů jsou přirozenou součástí tradice výuky informatiky a naše výsledky zdůrazňují jejich význam pro konceptuální porozumění.

Naše analýzy studentského učení jednotlivých OOP konceptů ukazují, že mnoho studentů má problémy s osvojením ústředních pojmů. Analýza schopnosti studentů zvládnout tyto problémy však také ukázala, že studenti disponují širokou škálou strategií, které mohou použít, když se „zaseknou“ ve svém učení.

Jak jsme již uvedli, z literárních zdrojů, které jsme prostudovali, vyplynulo,

že v různých oborech a specializacích existuje jen málo konceptů, které by měly podstatný transformační účinek. To znamená, posunuly schopnosti studenta na jinou kvalitativní úroveň v tomto daném oboru. Takové koncepty mají tedy podstatný dopad na schopnost studenta zvládnout dobře daný obor, v našem případě objektově orientované programování. Z našeho výzkumu vyplynulo, že v rámci objektově orientovaného programování existují dva prahové koncepty. Jedná se o OOP koncept „Rozhraní“ a OOP koncept „Událost“. Po překonání těchto dvou prahových konceptů je schopen student v programování řešit komplexnější úlohy, využívat efektivně návrhové vzory. Otevře se mu cesta k tomu, stát se softwarovým profesionálem. Náš výzkum také naznačuje, že problémy studentů s učením se programovat částečně závisí na složitém vztahu konceptů a vzájemné závislosti mezi nimi. Proto také OOP koncept „Rozhraní“ je tím prahovým konceptem. Integruje totiž více dalších konceptů a vztahů. Je třeba poznamenat, že jsme identifikovali další problematické koncepty, které měly ale menší transformační účinek, proto jsme je nezařadili do prahových konceptů. Pro výuku však i tyto koncepty mohou být velmi důležité, je třeba, aby jim byla ve výuce věnována patřičná pozornost.

V práci jsme také diskutovali přínosy našeho výzkumu pro pedagogickou praxi. Je třeba neustále hledat způsoby, jak zefektivnit výuku programování a jak ji udělat pro studenty atraktivnější. Osnovy předmětů obecně vždy odrážejí odborný pohled na to, které koncepty by měly být zahrnuty a zdůrazněny, to se týká výuky programování. Náš výzkum může pomoci učitelům hledat a zařadit nejdůležitější koncepty včetně jejich pořadí. Je také důležité, aby si byl učitel vědom, jak studenti chápou základní pojmy a také, jak studenti své učení prožívají. Takové znalosti umožní učitelům pomoci studentům objevit pokročilé způsoby porozumění. To však klade velké nároky na učitele, jak na jeho znalost metod programování, v našem případě objektově orientovaného programování, tak i znalost a aplikaci vhodných výukových metod.

Oblast vývoje software se velmi rychle rozvíjí. Množina úloh, které již ne-

ní třeba programovat, protože je umí naprogramovat určitý generátor kódu, se stále rozšiřuje. Důraz se začíná klást na návrh architektury programů (Pecinovský, 2013). Budoucí výzkum by se proto měl, podle našeho názoru, soustředit na výzkum prahových hodnot v oblasti návrhu programu a architektury. Jak uvádí Thomas a kol. (2017):

„Pokročilí studenti stále mají problémy, když jsou požádáni o provedení návrhových úkolů. Návrh programu včetně architektury je základní dovedností ve vzdělávání v informatice. “

Publikační aktivity a projekty

Publikace

BERÁNEK, L. a R. REMEŠ, 2021. The Use of a Game Theory Model to Explore the Emergence of Core/Periphery Structure in Networks and Its Symmetry. In *Symmetry*. 13(7):1214. https://doi.org/10.3390/sym13071214

BERÁNEK, L. a R. REMEŠ. Network Analysis of Intermediaries in Ecommerce. In *38th International Conference on Mathematical Methods in Economics*. Brno: Mendelova univerzita v Brně, 2020, s. 46–52. ISBN 978-80-7509-734-7.

BERÁNEK, L. a R. REMEŠ. Distribution of Node Characteristics in Evolving Tripartite Network. In *Entropy*, 2020, roč. 22, č. 3.

BERKOVÁ, I., MRKVIČKA, T., KLUFOVÁ, R. a R. REMEŠ. Detection Of Firms' Clustering By Local Scaling. In *Regional Science Inquiry*, 2020, roč. 12, č. 1, s. 115–127.

REMEŠ, R. a L. BERÁNEK. Understanding of basic concepts in novice programming courses. In *Proceedings of The 17th International Conference Efficiency and Responsibility in Education 2020 (ERIE)*. Praha: Czech University Life Sciences Prague, 2020, s. 242–248. ISBN 978-80-213-3022-1. ISSN 2336-744X.

DOBIÁŠ Václav, REMEŠ, Radim a Patrik KLOFÁČ (Eds.) 2020. *Sborník konference PRIT 2020* [DVD]. České Budějovice: Jihočeská univerzita, Pedagogická fakulta. ISBN 978-80-7394-805-4.

REMEŠ, R. a L. BERÁNEK. Objective Design in the Novice Programming Course in the Tertiary Education. In *Proceedings of The 16th International Conference Efficiency and Responsibility in Education 2019 (ERIE)*. Praha: Czech University Life Sciences Prague, 2019, s. 226–233. ISBN 978-80-213-2878-5. ISSN 2336-744X.

BERÁNEK, L. a R. REMEŠ. The use of belief function theory in recommendation based on a similarity diffusion. In *Conference Proceedings of 37th Internatio-*

nal Conference on Mathematical Methods in Economics 2019. České Budějovice: University of South Bohemia in České Budějovice, Faculty of Economics, 2019, s. 356–361. ISBN 978-80-7394-760-6.

DOBIÁŠ Václav, REMEŠ, Radim a Václav ŠIMANDL (Eds.) 2019. *Sborník konference PRIT 2019* [DVD]. České Budějovice: Jihočeská univerzita, Pedagogická fakulta. ISBN 978-80-7394-758-3.

BERÁNEK, L a R. REMEŠ. E-commerce Network with Price Comparator Sites. In *2019 9th International Conference on Advanced Computer Information Technologies ACIT'2019*. České Budějovice: Jihočeská univerzita v Č. Budějovicích, 2019. s. 401–404. ISBN 978-1-7281-0449-2.

REMEŠ, R. a L. BERÁNEK. The mental models of novice programmers for the assignment statement. In *Mezinárodní vědecká konference INPROFORUM 2018*. České Budějovice: Jihočeská univerzita v Č. Budějovicích, Ekonomická fakulta, 2018. s. 341–346. ISBN 978-80-7394-726-2.

BERÁNEK, L a R. REMEŠ. Model of e-commerce network with price comparison sites. In *Mezinárodní vědecká konference INPROFORUM 2018*. České Budějovice: Jihočeská univerzita v Č. Budějovicích, Ekonomická fakulta, 2018. s. 347–352. ISBN 978-80-7394-726-2.

REMEŠ, R. Výuka programování u začátečníků v terciárním vzdělávání. In: *ICTE 2018*. PhD section, PF, Ostravská univerzita v Ostravě, 2018. s. 89–104. ISBN 978--80-7599-029-7.

DOBIÁŠ, V. a R. REMEŠ (Eds.). *Sborník konference PRIT 2018* [DVD]. České Budějovice: Jihočeská univerzita, Pedagogická fakulta. ISBN 978-80-7394-707-1.

REMEŠ, R. Výuka programování pro začínající programátory. In *DITECH 2018*. Hradec Králové: Univerzita Hradec Králové. 2018.

REMEŠ, R. Teaching Programming for Novices. In *STEAM Education Conference*. Linz: Johannes Kepler Universität. 2018.

BERÁNEK, L. a R. REMEŠ. Modeling of e-shop selection processes the presence of price comparison sites. In *36th International Conference on Mathematical Methods in Economics*. Praha: Matfyzpress, 2018, s. 43–48. ISBN 978-80-7378-371-6.

BERÁNEK, L. a R. REMEŠ. Creating and Experience with Knowledge Assessment Module in Teaching Programming. In *Proceedings of The 15th International Conference Efficiency and Responsibility in Education 2018 (ERIE)*. Praha: Czech University Life Sciences Prague, 2018, s. 14–20. ISBN 978-80-213-2858-7. ISSN 2336--744X.

BERÁNEK, L. a R. REMEŠ. Model of e-commerce network with price comparison site. In *Mezinárodní vědecká konference INPROFORUM 2018*. České Budějovice: Jihočeská univerzita v Č. Budějovicích, Ekonomická fakulta, 2018.

BERÁNEK, L. a R. REMEŠ. Reasoning with streamed information from unreliable sources. In *Proceeding of the 14th International Conference Advanced Computer Information Technologies, ACIT*. Ternopil National Economic University, 2018, s. 149–152. ISBN 987-966-654-489-9.

BERÁNEK, L. a R. REMEŠ. Experience with Automatic Testing System in Teaching C♯ Programming. In *Proceedings of The 14th International Conference on Efficiency and Responsibility in Education (ERIE)*. Praha: Czech University Life Sciences Prague, 2017, s. 17–24. ISBN: 978-80-213-2762-7.

BERÁNEK, L., REMEŠ, R. a V. NÝDL. Application of structural equation modeling to explain online shoppers' response to price comparison sites. In *ACM International Conference Proceeding Series*. New York: ACM New York, NY, USA, 2017, s. 72–75. ISBN 978-1-4503-5248-2.

BERKOVÁ, I., MRKVIČKA, T., KLUFOVÁ, R. a R. REMEŠ. Detection of firms' clustering by local scaling. *Spatial statistics 2017: One world, one health*. Lancaster. United Kigdom, 2017 (poster).

REMEŠ, R. Solving Resource Allocation by Using Spreadsheet. In *The 11th International Scientific Conference INPROFORUM*. November 9-10, 2017, České Budě-

jovice, s. 373–378, ISBN 978-80-7394-667-8.

BERÁNEK, L. a R. REMEŠ. Modelling E-Commerce Processes in the Presence of a Price Comparison Site. In *The 11th International Scientific Conference INPROFORUM*. November 9–10, 2017, České Budějovice, s. 327–332, ISBN 978-80-7394--667-8.

REMEŠ, R., BERÁNEK, L. a J. MILOTA. Zkušenosti s virtualizačním nástrojem Proxmox ve výuce. In *Informatika XXX/2017*, Sborník abstraktů z mezinárodní odborné pedagogicky zaměřené konference. Brno: Mendelova univerzita, 2017. ISBN: 978-80-7509-512-1.

REMEŠ, R. (Ed.). *Sborník konference PRIT 2017* [DVD]. České Budějovice: Jihočeská univerzita, Pedagogická fakulta, 2017. ISBN 978-80-7394-657-9.

BERÁNEK, L., NÝDL, V. a R. REMEŠ. Click Stream Data Analysis for Online Fraud Detection in E-Commerce. In *Proceedings of the 10th International Scientific Conference INPROFORUM*. Threatened Europe - Socio-Economic and Environmental Changes 10, 2016, p. 175–180. ISBN 978-80-7394-607-4.

BERÁNEK, L. a R. REMEŠ. Prediction of unsuccessful students based on their activities in the LMS Moodle. In *Efficiency and Responcibilty in Education: 13th International Conference (ERIE 2016)*. Prague: Czech Univeristy of Life Sciences, 2016, s. 35–42. ISBN 978-80-213-2646-0.

REMEŠ, R. Solving Economic Problems by Using OML Modeling Language. In *Proceedings of the 10th International Scientific Conference INPROFORUM 2016*. Threatened Europe – Socio-Economic and Environmental Changes. 10, s. 218–224. 2016. ISBN 978-80-7394-607-4.

REMEŠ, R., L. BERÁNEK, L. HANZAL a J. MILOTA. Plánování projektů simplexovou metodou. In *Sborník abstraktů z konference Informatika XXIX/2016*. Brno: MZLU, 2016. ISBN 978-80-7509-420-9.

BERÁNEK, L. a R. REMEŠ. The Analysis of Factors Influencing an Information Security Awareness od Students. In *12th International Conference on Efficiency*

and Responsibility in Education – ERIE 2015. Praha: ČZU Praha, 2015, s. 34–41. ISBN 978-80-213-2560-9.

BERÁNEK, L., V. NÝDL a R. REMEŠ. Factors Influencing Customer Repeated Purchase Behavior in the E-commerce Context. In *Proceedings of the 9th International Scientific Conference INPROFORUM: Common challenges – Different solutions – Mutual dialogue*. Č. Budějovice: Jihočeská univerzita v Českých Budějovicích, 2015, s. 123–128. ISSN 2336-6788.

REMEŠ, R., L. BERÁNEK a J. MILOTA. Použití virtualizačního nástroje Proxmox ve výuce. In *Informatika XXVIII/2015*. Brno: Mendelova univerzita v Brně, s. 67–68. ISBN: 978-80-7509-344-8.

BERÁNEK, L. a R. REMEŠ. Support of students' engagement using wiki technology in the subject e-commerce. In *Efficiency and Responcibilty in Education 2014*. Praha: ČZU Praha, 2014, roč. 11, s. 32–38. ISBN 978-80-213-2468-8.

BERÁNEK L., NÝDL, V. a R. REMEŠ. Identification of Successful Sellers in Online Auction. In *Proceedings of the 8th International Scientific Conference Inproforum*. České Budějovice: Jihočeská univerzita v Č. Budějovicích, Ekonomická fakulta, 2014, s. 155–160. ISBN 978-80-7394-484-1.

REMEŠ, R., BERÁNEK, L. a J. MILOTA. Řešení úloh lineárního programování pomocí algebraického modelovacího jazyka OML. In *Informatika 2014*. Brno: Mendelova univerzita v Brně, 2014, s. 67–68. ISBN 978-80-7509-126.

REMEŠ, R., BERÁNEK, L. a J. MILOTA. Projektování informačních systémů – přechod na software Enterprise Architect. In *Konference Informatika XXVI/2013*. Brno: Mendelova univerzita v Brně, 2013, 11–12 (P03-6 stran). ISBN 978-80-7375--834-9.

BERÁNEK, L. a R. REMEŠ. Evaluation of the Support of Entrepreneurial Competences of Students in the Subject E-Commerce. In *Efficiency and Responsibility in Education, 10th International Conference*. Prague:Czech University of Life Sciences Prague, 2013, s. 32–38. ISBN 978-80-213-2378-0.

BERÁNEK, L., NÝDL, V. a R. REMEŠ. Odhad přesného hodnocení získávaného z více informačních zdrojů. In *The International Scientific Conference INPROFORUM 2013*. České Budějovice: Ekonomická fakulta, Jihočeská univerzita, 2013, 37–42. ISBN 978-80-7394-440-7.

REMEŠ, R. (ed.). *Kvantitativní metody v ekonomii 2013*. Sborník prací z mezinárodního vědeckého semináře. České Budějovice: Jihočeská univerzita, Ekonomický fakulta, 2013. ISBN 978-80-7394-419-3.

BERÁNEK, L. a R. REMEŠ. The course of e-commerce based on active leasing. In *9th International Conference on Efficiency and Responsibility in Education – ERIE 2012*. Praha: ČZU Praha, 2012, s. 36–44. ISBN 978-80-213-2289-9.

REMEŠ, R., L. BERÁNEK a A. CARBOVÁ. Modelování problémů s omezujícími podmínkami v programovacím jazyce C♯. In *Informatika 2012*. Brno: Mendelova univerzita v Brně, 2012. s. 93–94. ISBN 978-80-7375-628-4.

GYEPES, M., BERÁNEK, L. a R. REMEŠ. *HomeBK*. University of South Bohemia, Faculty of Economics, 2011. [2021-06-06]. Dostupné z: http://ec.ef.jcu.cz/projects/homebk/

BERÁNEK, L., KNÍŽEK, J. a R. REMEŠ. Model zpracování bezpečnostních zpráv ze systémů pro fázi monitorování stavu bezpečnosti informací organizace. In *Sborník z mezinárodní vědecké konference Inproforum 2011 „Globální ekonomická krize – regionální dopady"*. Č. Budějovice: Jihočeská univerzita v Č. Budějovicích, Ekonomická fakulta, 2011, s. 29–35. ISBN 978-80-7394-316-5.

BERÁNEK, L. a R. REMEŠ. Výuka předmětu Podnikání a obchodování na internetu. In *Informatika XXIV/2011*. Brno: Mendelova univerzita v Brně, 2011, s. 1–6. ISBN 978-80-7375-520-1.

REMEŠ, R., BERÁNEK, L. a A. CARBOVÁ. Řešení logických úloh pomocí počítače. In *5. konference „Užití počítačů ve výuce matematiky"*. Č. Budějovice: Jihočeská univerzita v Českých Budějovicích, Pedagogická fakulta, 2011, s. 311–319. ISBN 978-80-7394-324-0.

BERÁNEK, L., NÝDL, V. a R. REMEŠ. The security of electronic auction systems. In *Acta Universitatis Bohemiae Meridionales*. Č. Budějovice: JU v Č. Budějovicích, 2010, vol. 13, no. 1, p. 127–131. ISSN 1212-3285.

BERÁNEK, L. a R. REMEŠ. Education of Information Security for Future Teachers. In *Proceedings of 7th International Conference Efficiency and Responsibility in Education*. Prague: Czech University of Life Science, 2010, p. 42–49. ISBN 978--80-213-2084-0.

BERÁNEK, L., TLUSTÝ, P. a R. REMEŠ. The usage of Belief Functions for an Online Auction Reputation Model. In *Mathematical Methods in Economics 2010*. České Budějovice: University of South Bohemia – Faculty of Economics, 2010, p. 49–54. ISBN 978-80-7394-218-2.

BERÁNEK, L., TLUSTÝ, P. a R. REMEŠ. An Online Auction Trust Model for Based on the Contextual Information. In *Proceedings of the Workshop on the Theory of Belief Functions (Belief 2010)*. France, Brest, 2010, Paper 110, p. 1–6. ISBN 3-642--14048-3.

REMEŠ, R. a L. BERÁNEK. *Ebix*. University of South Bohemia, Faculty of Economics, 2010. [2021-06-06]. Dostupné z: http://ec.ef.jcu.cz/projects/ebix/

REMEŠ, R., BERÁNEK, L. a M. ŠULISTA. *INFA in English*. University of South Bohemia, Faculty of Economics, 2010. [2021-06-06]. Dostupné z: http://ec.ef.jcu.cz/projects/infaen/

REMEŠ, R., BERÁNEK, L., a L. FRIEBEL. Multimediální opora výuky Manažerské informatiky v angličtině. In *Informatika XXIII 2010*. Brno: MU v Brně, 2010. s. 101–102. ISBN 978-80-7375-394-8.

BERÁNEK, L. a R. REMEŠ. Risk Analysis Methodology Designed for Small and Medium Enterprises. In *Acta Universitatis Bohemiae*. České Budějovice: University of South Bohemia, 2009, XII, (2), pp. 65–70. ISSN 1212-3285.

BERÁNEK, L. a R. REMEŠ. The Proposal of Software Development and Acquisition Metrics Based on ISO/IEC 27001 Standard. In *Acta Universitatis Bohemiae*.

České Budějovice: University of South Bohemia, 2009, XII, (3), pp. 93–98. ISSN 1212-3285.

BERÁNEK, L., REMEŠ, R. a L. FRIEBEL. Obor Ekonomická informatika na Jihočeské univerzitě v Českých Budějovicích. In *Informatika XXII/2009*. Luhačovice: KONVOJ, 2009, s. 18–19. ISBN 978-80-7302-152-8.

REMEŠ, R., BERÁNEK, L. a M. ŠULISTA. Tvorba multimediálních pomůcek pro výuku matematiky. In *Sborník 4. konference „Užití počítačů ve výuce matematiky"*. České Budějovice: Jihočeská univerzita v Č. Budějovicích – Pedagogická fakulta, 2009, s. 186–191. ISBN 978-80-7394-186-4.

ŠULISTA, M. a R. REMEŠ. Aplety jako první krok k implementaci metody CLIL ve výuce matematiky. In *Sborník 4. konference „Užití počítačů ve výuce matematiky"*. České Budějovice: Jihočeská univerzita v Č. Budějovicích – Pedagogická fakulta, 2009, s. 218–221. ISBN 978-80-7394-186-4.

REMEŠ, R. *Programujeme v jazyku Python*. České Budějovice: Johanus, 2008. iv, 78 s. ISBN 978-80-7394-128-4.

REMEŠ, R. a P. TLUSTÝ. Odsouzenci a matematika. In *XXVI International Colloquium on the Management of Educational Process: proceedings of abstracts and electronic versions of reviewed contributions on CD-ROM* [CD-ROM]. Brno: UO, 2008. Adresář: 6clanky/1remesr.pdf. ISBN 978-80-7231-511-6.

REMEŠ, R. Zpřístupnění matematického zápisu na stránkách WWW. In *Sborník příspěvků 3. konference „Užití počítačů ve výuce matematiky"*. Č. Budějovice: Jihočeská univerzita v Č. Budějovicích – PF, 2007, s. 209–213. ISBN 978-80-7394-048--5.

REMEŠ, R. a P. TLUSTÝ. Publikování matematických vzorců na internetu. In *International Colloquium on the Management of Educational Process*. Brno: Univerzita obrany – Fakulta ekonomiky a managementu, 2007. ISBN 978-80-7231--228-3.

ŠULISTA M., BISKUP R. a R. REMEŠ. Implementace natrénované neuronové sítě.

In *Tvorba sotfwaru 2007*. Ostrava: VŠB-TU – Ekonomická fakulta, 2007. ISBN 80--248-1427-8.

REMEŠ, R. Porovnání OpenOffice.org a Microsoft Office. In *XXIV. mezinárodní kolokvium*, 18. 5. 2006. Brno: Univerzita obrany – Fakulta ekonomiky a managementu, 2006. ISBN 80-7231-139-5.

REMEŠ, R. Possibilities and Use of a University E-learning System, m-8. In *WDS ´06*, Proceedings of contributed papers, Praha: Charles University, part I – Mathematics and Computer Sciences, 2006, s. 22–26. ISBN 80-86732-84-3.

REMEŠ, R. a P. TLUSTÝ. Aplikace dělitelnosti v běžném životě. In *XXIV. mezinárodní kolokvium*. Brno: Univerzita obrany – Fakulta ekonomiky amanagementu, 2006. ISBN 80-7231-139-5.

ROST, M., KLUFOVÁ, R. a R. REMEŠ. R jako alternativní prostředí pro výuku statistiky na ZF JU. In *Efficiency and Responsibility in Education*, Proceedings of Papers 2005. Czech university of Agriculture in Prague, Faculty of Economics and Management, Praha, 2005, s. 187–192. ISBN 80-213-1349-8.

PROKÝŠEK M. a R. REMEŠ. Tvorba databází v Microsoft Visual Foxpro a databázové nástroje v jazyce Borland Delphi. In *Tvorba softwaru 2005*. Tanger s.r.o., Ostrava, 2005, s. 211–215. ISBN 80-86840-14-X.

PROKÝŠEK M., REMEŠ, R., VANĚČEK, P. Interactive Stereoscopic Projection in Geometry. In *Univ. S. Boh. Dept. Math. Rep. Series*. Č. Budějovice: JU v Č. Budějovicích, 2005, vol. 13, p. 171–174. ISSN 1214-4681.

REMEŠ, R. Porovnání opensource CMS pro využití v e-learningu. In *DIVAI 2005 – Dištančné vzdelávanie v Aplikovanej informatike*. Nitra: UKF – FPV, 2005, s. 69. ISBN 80-8050-828-3.

REMEŠ R. Learning Management System. In *WDS'05 Proceedings of Contributed Papers: Part I*. Prague: Matfyzpress, 2005, p. 207–212. ISBN 80-86732-59-2.

REMEŠ, R. a M. ROST. Živé Linuxové distribuce. In *Efficiency and Responsibility*

in Education, Proceedings of Papers 2005. Praha: ČZU – Provozně-ekonomická fakulta, 2005, s. 182–185. ISBN80-213-1349-8.

REMEŠ, R., ROST, M. a R. BISKUP. Quantian jako vědecké počítačové prostředí. In *Forum Statisticum Slovacum*. Slovenská štatistická a demografická spoločnosť, 2005. s. 211–214. ISSN 1336-7420.

REMEŠ, R., HOCOVÁ, P. a P. TLUSTÝ. Simulace pravděpodobnostních jevů pomocí Galtonovy desky. In *Univ. S. Boh. Dept. Math. Rep. Series*. Č. Budějovice: JU v Č. Budějovicích, 2005, roč. 13, s. 175–178. ISSN 1214-4681.

REMEŠ, R. a P. TLUSTÝ. Galtonova deska. In *Zborník vedeckých prác z mezinárodnej vedeckej konferencie Matematika vo výučbe, výskume a praxi*. Nitra: Polnohospodárska univerzita v Nitre, Katedra matematiky, 2003, s. 77–79. ISBN 80-8069--203-3.

Projekty

Projekt TAČR TL03000222 2020–2023. Rozvoj informatického myšlení pomocí situačních algoritmických problémů (spoluřešitel).

Projekt Interreg V-A ATCZ16d 2017–2019 SIP-SME (Service Innovations Prozess für Klein-und Mittelunternehmen) (spoluřešitel).

Projekt institucionálního plánu 2016–2018 Inovace studijních programů na Ekonomické fakultě JU (spoluřešitel).

Projekt FRVŠ č. 1754/2010: Inovace předmětu Podnikání a obchodování na internetu (spoluřešitel).

Projekt FRVŠ č. 2120/2009: Tvorba multimediálních pomůcek v anglickém jazyce pro předmět Informatika I (hlavní řešitel).

Projekt FRVŠ č. 2492/2007: Tvorba nového předmětu „HTML a kaskádové styly“ (spoluřešitel).

Interní projekt ZF JU v Č. Budějovicích, IG 18/2004: Výpočetní technika – e-learning a internetová výuka (hlavní řešitel).

Seznam použité literatury

ADAMS, J. a J. FRENS, 2003. Object centered design for Java: Teaching OOD in CS-1. In *ACM SIGCSE Bulletin*. 35(1), 273–277. Doi.org/10.1145/792548.611986.

ADLER, P. A. a P. ADLER, 1994. Observational techniques. In DENZIN, N. K. a Y. S. LINCOLN. *Handbook of qualitative research*. Thousand Oaks: Sage, 377–392. ISBN 978-08-0394-679-8.

ALPHONCE, C. a P. VENTURA, 2002. Object orientation in CS1-CS2 by design. In CASPERSEN, M. E. A D. JOYCE. *Proceedings of the 7th annual conference on Innovation a technology in computer science education 2002*. Aarhus, Juni 24–26 2002. New York: ACM Press, 70–74. ISBN: 978-1-58113-499-5.

ALPHONCE, C., M. CASPERSEN, A. DECKER a B. TRASK, 2006. Killer examples for design patterns. In P. TARR. OOPSLA '06: *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. Portland October 22 – 26, New York: ACM, 625-626. ISBN: 978-1-59593--491-8. DOI: 10.1145/1176617.1176640.

AMSTRONG, D. J., 2006. The Quarks of Object-Oriented Development. In *Communications of the ACM*, 49(2), 123-128. DOI: 10.1145/1113034.1113040.

ASAGBA, P. O. a E. E. OGHENEOVO, 2008. A Comparative Analysis of Structured and Object-Oriented Programming Methods. In *Journal of Applied Sciences and Environmental Management*. 12(4), 41-46.

ASHWIN, A., 2008. What do students' examination answers reveal about threshold concept acquistion in the 14 - 19 age group. In LAND, R., J. H. F. MEYER A J. SMITH. *Threshold Concepts within the Disciplines*. Rotterdam: Sense Publishers, 2008, 173–184. ISBN: 978-90-8790-268-1

ATMAN, C. J., a K. M. BURSIC, 1998. Verbal protocol analysis as a method to document engineering student design processes. In *Journal of Engineering Education*. 87(2), 121–132.

BALEY, K. a D. BELCHAM, 2010. *Brownfield Application Development in .NET*. Greenwich, CT, USA: Manning Publications. ISBN 978-193-398-871-9

BARRADELL, S., 2013. The identification of threshold concepts: A review of theoretical complexities and methodological challenges. In *Higher Education*, 65(2), 265-276. DOI: 10.1007/s10734-012-9542-3

BARNES, D. J., S. FINCHER, a S. THOMPSON, 1997. Introductory Problem Solving in Computer Science. In DAUGHTON, G. A P. MAGEE. *5th Annual Conference on the Teaching of Computing* [online]. Dublin: Dublin City University, 1997, 36–39. [cit. 2020-8-22]. Dostupné z: https://kar.kent.ac.uk/21468/2/Introductory_Problem_Solving_in_Computer_Science.pdf

BOUSTEDT, J., A. ECKERDAL, R. MCCARTNEY, J. MOSTRÖM, M. RATCLIFFE, K. SANDERS a C. ZANDER, 2007. Threshold concepts in computer science: Do they exist and are they useful? *ACM SIGCSE Bulletin*, 39(1). 504-508. DOI: 10.1145/1227504.1227482.

BAYMAN, P. a R. E MAYER, 1983. A Diagnosis of Beginning Programmers' Misconceptions of BASIC Programming Statements. In *Communication of ACM*, 26(9), 677-679. DOI: 10.1145/358172.358408

BELL, J., 2005. *Doing your research project: a guide for first-time researchers in education, health and social science*. Maidenhead: Open University Press. ISBN 978-0335215041

BEN-ARI, M., 2001. Constructivism in computer science education. Journal of Computers in Mathematics and Science Teaching, 20 (1), 45-73.

BENNEDSEN, J. a M. CASPERSEN, 2004. Programming in context: a model-first approach to CS1. ACM SIGCSE Bulletin, 36(1), 477–481. DOI: 10.1145/1028174.971461.

BENNEDSEN, J. a C. SCHULTE, 2008), What does 'objects-first' mean? An international study of teachers' perceptions of objects-first. In LISTER, R. *Koli Calling*

'07: Seventh Baltic Sea Conference on Computing Education Research, Koli National Park, Finland, November 15-18 2007. Darlinghurst: Australian Computer Society, Inc. 21–29. ISBN: 978-1-920682-69-9

BENNEDSEN, J. a C. SCHULTE, 2013. Object Interaction Competence Model v. 2.0. In *Learning and Teaching in Computing and Engineering (LaTiCE)*, Macau, March 22-24 2013. Los Alamitos: IEEE Press. 9–16. ISBN 978-0-7695-4960-6. DOI: 10.1109/LaTiCE.2013.43.

BERGES, M., 2018. Object orientation in the literature and in education. In *Information Technology*. 60(2), 69 - 77.

BERTRAND, Y., (1998). *Soudobé teorie vzdělávání*. Praha: Portál. ISBN: 80-7178--216-5.

BIELIKOVÁ, M. a P. NÁVRAT, 2009. *Funkcionálne a logické programovanie* [online]. Slovenská technická univerzita v Bratislave: STU, s. 1-7 [cit. 2020-9-01]. ISBN 978-80-227-3225-3. Dostupné z: https://www.researchgate.net/publication/ 268432411_ Funkcionalne_a_logicke_programovanie.

BIGGERSTAFF, D. a A. R. THOMPSON, 2008. Interpretive Phenomenological Analysis (IPA): a qualitative methodology of choice in healthcare research. Qualitative Research in Psychology, 5(3), 214 - 224.

BIGGS, J. B. a C. TANG. (2011). Teaching for Quality Learning at University. 4th edition. New York: Open University Press. ISBN: 978-0-3352-4275-7.

BIGGS, J. B. a K. F. COLLIS. (1982). Evaluating the quality of learning: The SOLO taxonomy (Structure of the Observed Learning Outcome). New York. Academic Press. ISBN 978-1-4832-7331-0.

BLOCH, J., 2002. Java Efektivně. Praha: Grada. ISBN 80-247-0416-1.

BLOOM, B. S. (1956). Taxonomy of educational objectives: The classification of educational goals. Handbook 1: Cognitive Domain. New York: Longmans, Green. ISBN 978-0-582-28010-6.

BOOTH, M. L., T. CHEY a A. BAUMAN, 2001. The reliability and validity of the physical activity questions in the WHO health behaviour in schoolchildren (HBSC) survey: A population study. British journal of sports medicine. 35(4), 263–7. DOI: 10.1136/bjsm.35.4.263.

BORGE, R. E., 2004. Teaching OOP using graphical programming environments: An expermental study, Disertační práce, University of Oslo, Oslo.

BOUSTEDT, J., 2008. A methodology for exploring students' experiences and interaction with large-scale software through role-play and phenomenography. In ICER'08 – Proceedings of the ACM Workshop on International Computing Education Research. 27–38. Sydney, Australia: ACM. DOI: 10.1145/1404520.1404524.

BOWDEN, J., 2000. The nature of phenomenographic research. In: BOWDEN J. a E. WALSH. Warburton symposium; Phenomenography. Melbourne: RMIT University Press, 1–18. ISBN: 0864590199. [cit-2020-10-05]. Dostupné z: https://search.informit.org/doi/10.3316/INFORMIT.733212511947808

BROCKI, J. M. a A. J. WEARDEN, 2006. A critical evaluation of the use of interpretive phenomenological analysis (IPA) in health psychology. *Psychology and Health*, 21(1), 87–108.

BRUCE, K. B., A. DANYLUK, a T. MURTAGH, 2001. A library to support a graphics based object-first approach to CS 1. *ACM SIGCSE Bulletin*, 33(1), 6–10. DOI: 10.1145/366413.364527.

BRYMAN, A., 2006. Integrating quantitative and qualitative research: how is it done? *Qualitative Research*, 6(1), 97–113. DOI: 10.1177/1468794106058877.

BUCKS, G., a W. C. OAKES, 2011. Phenomenography as a Tool for Investigating Understanding of Computing Concepts. Paper presented at *2011 ASEE Annual Conference & Exposition*, Vancouver, BC. [online]. DOI: 10.18260/1-2-18485. [cit. 2020-11-01]. Dostupné z: http://www.asee.org/public/conferences/1/papers/660/

ČADA, O., 2009. *Objektové programování: naučte se pravidla objektového myšlení*. 1. vydání. Praha: Grada. ISBN 978-80-247-2745-5.

CLARKE, C., 2009. An introduction to interpretive phenomenological analysis: a useful approach for occupational therapy research. *British Journal of Occupational Therapy*, 72(1), 37 - 39. DOI: 10.1177/030802260907200107.

CORNEY, M., D. TEAGUE, A. AHADI a R. LISTER, 2012. Some empirical results for Neo-Piagetian reasoning in novice programmers and the relationship to code explanation questions. In *Proceedings of the 14th Australasian Computing Education Conference (ACE'12)*. Vol. 123. Melbourne, Australia: Australian Computer Society Inc., 77–86. ISBN: 978-1-921770-04-3

COHEN, L., L. MANION a K. MORRISON, 2000. *Research methods in education*. 5.vyd., New York: RoutledgeFalmer. ISBN 0-415-19541-1.

COOPER, S., W. DANN, a R. PAUSCH, 2003. Teaching objects-first in introductory computer science. In *Proceedings of the 34th SIGCSE technical symposium on Computer science education*, Reno, February 19-23 2003, ACM Press, New York, pp. 191–195. ISBN: 978-1-58113-648-7

CREWS, T. a J. BUTTERFIELD, 2002. Using technology to bring abstract concepts into focus: A programming case study. In *Journal of Computing in Higher Education*, 13(2), 25–50. DOI: 10.1007/BF02940964.

COUSIN, G., 2008. Threshold concepts: old wine in new bottles or a new form of transactional curriculum inquiry? In LAND, R., J. H. F. MEYER A J. SMITH. *Threshold Concepts within the Disciplines*. Rotterdam: Sense Publishers, 2008, 259-272. ISBN: 978-90-8790-268-1. DOI: 10.1163/9789460911477_020.

CRESWELL, J. W., 1998. *Qualitative Inquiry and Research Design: Choosing Among Five Traditions*. Thousand Oaks: Sage Publications. ISBN 978-1-4129-1606-6.

DAVIES, P., 1993. Models and theories of programming strategy. In *International Journal of Man-Machine Studies*. 39(2), 237–267. DOI: 10.1006/imms.1993.1061.

DAVIES, P., (2006). Threshold concepts: How can we recognise them? In MEYER, J. a R. LAND. *Overcoming Barriers to Student Understanding*. London: Routledge, Chapter 5, 15s. ISBN 97-802-039-66-273.

DAVIES, P., a J. MANGAN, 2005. Recognising threshold concepts: an exploration of different approaches. In *11th Conference of the European Association for Research on Learning and Instruction (EARLI 2005)*. [online]. August 23-27, 2005, Nicosia, Cyprus, EARLI. [cit. 2020-04-25]. Dostupné z: https://www.researchgate.net/publication/228377119_Recognising_Threshold_Concepts_an_exploration_of_different_approaches.

DECKER, R. a S. HIRSHFIELD, 1994. The top 10 reasons why object-oriented programming can't be taught in CS 1. *ACM SIGCSE Bulletin*, 26(1), 51–55. DOI: 10.1145/191033.191054.

DECKER, A., 2003. A tale of two paradigms. *Journal of Computing Sciences in Colleges*. 19(2), 238–246.

DENNY, P., A. LUXTON-REILLY a B. SIMON, 2008. Evaluating a new exam question: Parsons problems. In CASPERSEN, M. *Proceedings of the 4th international workshop on Computing Education Research (ICER'08)*. Sydney, Australia: ACM, 113–124. ISBN: 978-1-60558-216-0

DEWEY, J., 1933. *How we think*. Boston: D. C. Heath & Co., nové vydání: Endymion Press 2018, Free Kindle Edition, 109 s., ASIN B07BXBPJTP.

DIETHELM, I., 2007. Strictly *models and objects first: Unterrichtskonzept und –methodik für objektorientierte Modellierung im Informatikunterricht*, Disertační práce, Universität Kassel, Kassel.

DONCHEV, I a E. TODOROVA, 2008. Object-Oriented Programming in Bulgarian Universities' Informatics and Computer Science Curricula. In *Informatics in Education*, 7(2), 159-172. DOI:10.15388/infedu.2008.10.

DOOLEY, L. M., 2002. Case Study Research and Theory Building. In *Advances in Developing Human Resources*, 4(3), pp. 335-354. DOI: 10.1177/1523422302043007.

DOS SANTOS, V., DE SOUZA, É. F., FELIZARDO K. R. a N. L. VIJAYKUMAR, 2017. Analyzing the use of concept maps in computer science: A systematic mapping study. *Informatics in Education*, 16(2), 257–288. DOI: 10.15388/infedu.2017.13.

DRBAL, Pavel, 1994. Vývoj programování a jeho další cesta aneb Od selského rozumu k objektům. In *Programování '94*. Ostrava: Dům techniky Ostrava, s. 1-7. [online]. [cit. 2020-08-25]. Dostupné z: http://prog-story.technicalmuseum.cz/images/dokumenty/Programovani-TSW--1975-2014/1994/1994-03.pdf

DRIVER, R., H. ASOKO, J. LEACH, E. MORTIMER, a P. SCOTT, 1994. Constructing Scientific Knowledge in the Classroom. In *Educational Researcher*. 23(7), 5-12. DOI: 10.3102/0013189X023007005

DU BOULAY, B., 1986. Some difficulties of learning to program. In *Journal of Educational Computing Research*. 2(1), 57–73. DOI: 10.2190/3LFX-9RRF-67T8-UVK9

ECKERDAL, A., R. MCCARTNEY, J. E. MOSTRÖM, M. RATCLIFFE, K. SANDERS a C. ZANDER, 2006. Putting Threshold Concepts into Context in Computer Science Education. In *Proc 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITICSE'06)*. ACM, 103–107. DOI: 10.1145/1140124.1140154.

EEZY, 2021. Business People Icon Collection 2380208 Vector Art at Vecteezy. *Vecteezy* [online]. FITRIATI, A. [2021-06-06]. Dostupné z: https://www.vecteezy.com/vector-art/2380208-business-people-icon-collection

EHLERT, A. a C. SCHULTE, 2009. Empirical comparison of objects-first and objects later. In CLANCY, M., M. CASPERSEN a R. LISTER. *Proceedings of the 5th international workshop on Computing education research workshop*, Berkeley, Aug 10-11 2009, New York ACM Press, 15–26. ISBN: 978-1-60558-615-1. DOI: 10.1145/1584322.1584326.

EHLERT, A., 2012. *Empirische Studie: Unterschiede im Lernerfolg und Unterschiede im subjektiven Erleben des Unterrichts von Schülerinnen und Schülern im In-*

formatik-Anfangsunterricht (11. Klasse Berufliches Gymnasium) in Abhängigkeit von der zeitlichen Reihenfolge der Themen (OOP-First und OOP-Later). Disertační práce, Freie Universität Berlin, Berlin.

EISENHART, M., 2009. Generalization from Qualitative Inquiry. In: K. ERCIKAN a W. M. ROTH. Generalizing from Educational Research. New York: Routledge. Ch. 4, 16 s. ISBN 9780203885376.

ELO, S., a H. KYNGÃS, 2008. The Qualitative Content Analysis Process. In *Journal of Advanced Nursing*, 62(1), 107-115. DOI: 10.1111/j.1365-2648.2007.04569.x.

ENTWISTLE, N. J., 2008. Threshold concepts and transformative ways of thinking within research into higher education. In LAND, R., J. H. F. MEYER A J. SMITH. *Threshold Concepts within the Disciplines*. Rotterdam: Sense Publishers, 2008, 21–35. ISBN: 978-90-8790-268-1. DOI: 10.1163/9789460911477_003.

FELTEN, P., 2016. On the Threshold with Students. In: R. LAND, J. H. F. MEYER a M. FLANAGAN. *Threshold Concepts in Practice. Educational Futures (Rethinking Theory and Practice)*. SensePublishers, Rotterdam, 3–9. ISBN 978-94-6300-512-8. DOI: 10.1007/978-94-6300-512-8_1, pp. 3-9.

FLEURY, A. E., 2000. Programming in Java. *ACM SIGCSE Bulletin*. 32(1), 197–201. DOI: 10.1145/330908.331854.

FORMAN, J. a L. DAMSCHRODER, 2008. Qualitative Content Analysis: A Primer. In *Empirical Methods for Bioethics*. 11, 39-62. DOI: 10.1016/S1479-3709(07)11003--7.

FRIBERG, F., DAHLBERG, K., PETERSSON, M., a J. ÖHLÉN, 2000. Context and methodological decontextualization in nursing research with examples from phenomenography. In *Scandinavian Journal of Caring Sciences*, 14(1), 37–43. DOI: 10.1111/j.1471-6712.2000.tb00559.x

GAMMA, E., HELM, R., JOHNSON, R. a J. Vlissides, 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Indianapolis, IN: Addison-Wesley Professional. ISBN 0-201-63361-2.

GARDNER, P., 1998. Classroom teachers and educational change 1876 - 1996. In *Journal of Education for Teaching*. 24(1), 33–49. DOI: 10.1080/02607479819908.

GINAT, D., a E. MENASHE, 2015. SOLO Taxonomy for assessing novices' algorithmic design. In DECKER, A., K. EISELT. *Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE'15)*. Kansas City, Missouri, USA: ACM, 452–457. ISBN 978-1-4503-2966-8. DOI: 10.1145/2676723.2677311.

GRAINGER, R. a D. TOLHURST, 2005. Organisational factors affecting teachers' use and perception of information communication technology. In *South East Asian Regional Computer Confederation (SEARCC) 2005*, Sydney, Australia, September 2005 [online]. Sydney: Australian Computer Society, Inc., 2005, s. 13-22 [cit. 2019-12-05]. DOI: 10.5555/1151681.1151683

GIORGI, A., 1986. *A phenomenological analysis of descriptions of concepts of learning obtained from a phenomenographic perspective* (Report No. 6). Gothenburg: Department of Education, University of Götborg. ISSN 0282-2180.

GIORGI, A. (2008). Difficulties encountered in the application of the phenomenological method in the social sciences. In *Indo-Pacific Journal of Phenomenology*, 8(1), 1–9. DOI: 10.1080/20797222.2008.11433956

GIORGI, A., 2009. *The descriptive phenomenological method in psychology: A modified husserlian approach*. Pittsburgh, PA: Duquesne University Press. ISBN 978--0-8207-0418-0.

GROVER, S., R. PEA a S. COOPER, 2015. Designing for deeper learning in a blended computer science course for middle school students. In *Computer Science Education*. 25(2), 199–237. DOI: 10.1080/08993408.2015.1033142.

HADAR, I. a E. HADAR, 2007. An iterative methodology for teaching object oriented concepts. In *Informatics in Education*. 6(1), 67–80. DOI: 10.5555/1322395.1322401.

HADAR, I., 2013. When intuition and logic clash:The case of the object-oriented paradigm. In *Science of Computer Programming*, 78, 1407–1426. DOI: 10.1016/

j.scico.2012.10.006.

HALSTEAD, M. H. (1977). *Elements of Software Science*. New York: Elsevier. ISBN 978-0-444-00205-1.

HAN, F., a R.A. ELLIS, 2019. Using Phenomenography to Tackle Key Challenges in Science Education. In *Frontiers in Psychology*, 10(1414). DOI:10.3389/fpsyg.2019.01414.

HARLOW, A., S. JONATHAN, P. MIRA, C. BRONWEN, 2011. 'Getting stuck' in analogue electronics: threshold concepts as an explanatory model. *European Journal of Engineering Education*. 36. 435-447. DOI: 10.1080/03043797.2011.606500.

HOLLAND, S., R. GRIFFITHS a M. WOODMAN, 1997. Avoiding object misconceptions. *ACM SIGCSE Bulletin*, 29(1), 131–134. DOI: 10.1145/268085.268132.

HOLMSTRÖM, I., C. HALFORD a U. ROSENQVIST, 2003. Swedish health care professionals' diverse understandings of diabetes care. In *Patient Education and Counseling*, 51 (1), 53–58. DOI: 10.1016/s0738-3991(02)00212-4.

HOLUB, A., 2004. *Holub on Patterns: Learning Design Patterns by Looking at Code*. Berkeley: Apress. ISBN 978-1-59059-388-2.

HUBÁLOVSKÝ, Š. a O. KOŘÍNEK, 2015a. Algorithmic Thinking in Paradigms of Programming. In *Recent Advances in Educational Technologies*, The 2015 International Conference on Education and Modern Educational Technologies (EMET 2015), Series: Educational Technologies Series, Zakynthos Island, Greece, July 16-20, pp. 48-53. ISBN 978-1-61804-322-1, ISSN 2227-4618.

HUBÁLOVSKÝ, Š. a O. KOŘÍNEK, 2015b. Object Thinking in Paradigms of Programming. In *Recent Advances in Educational Technologies*, The 2015 International Conference on Education and Modern Educational Technologies (EMET 2015), Series: Educational Technologies Series, Zakynthos Island, Greece, July 16-20, pp. 54-58. ISBN 978-1-61804-322-1, ISSN 2227-4618.

HUBWIESER, P., 2007. A smooth way towards object oriented programming in

secondary schools. In BENZIE, D. a M. IDING. *Informatics, Mathematics and ICT: A golden triangle: Proceedings of the Working Joint IFIP Conference: WG3.1 Secondary Education, WG3.5 Primary Education; College of Computer and Information Science*, Northeastern University Boston, Massachusetts, USA 27th - 29th June 2007 [online]. [cit. 2019-10-29]. Dostupné z: https://www.edu.tum.de/fileadmin/tuedz01/ddi/Publikationen/2005-09/2007-Hubwieser-IMICT-Boston.pdf.

HUBWIESER, P. a A. MÜHLING, 2011. What students (should) know about object oriented programming. In SANDERS, K. a CASPERSEN, M.E. *Proceedings of the seventh international workshop on computing education research (ICER ´11)*. New York: ACM, 77–84. ISBN 978-1-4503-0829-8 DOI: 10.1145/2016911.2016929.

HUGHES, J. a P. DAWN, 2006. Assisting CS1 Students to Learn: Learning Approaches and Object-Oriented Programming. *ACM Sigcse Bulletin*. 38(3). 275-279. DOI: 10.1145/1140123.1140197.

HSIEH, H.F. a S.E. SHANNON, 2005. Three Approaches to Qualitative Content Analysis. In *Qualitative Health Research*, 15(9), 1277-1288. DOI: 10.1177/1049732305276687.

IS2020, 2020. The Joint ACM/AIS IS2020 Task Force. IS2020. A Competency Model for Undergraduate Programs in Information Systems 2020. Chairs: Paul Leidig, Hannu Salmela, *ACM and AIS*. DOI: 10.1145/3460863. [cit. 2021-01-14]. Dostupné z: https://dl.acm.org/citation.cfm?id=3460863

JTFCC, 2013. Joint Task Force on Computing Curricula. Computing Curricula (2013) Computer Science. In *Journal of Educational Resources in Computing (JERIC), 2013*. ISBN 978-1-4503-2309-3. DOI: 10.1145/2534860. [cit. 2020-11-29]. Dostupné z: https://dl.acm.org/doi/book/10.1145/2534860

KABO, J. a C. BAILLIE, C., 2010. Engineering and social justice: Negotiating the spectrum of liminality. In LAND, R., J.H.F. MEYER a C. BAILLIE. *Threshold Concepts and Transformational Learning*. Rotterdam: Sense, 303-315. ISBN 978-94-

-6091-206-1. DOI: 10.1163/9789460912078_019

KALHOUS, Z., O. Obst, 2002. *Školní didaktika*. Praha: Portál. ISBN 978-80-7367--571-4.

KASTO, N. a J. WHALLEY. (2013). Measuring the difficulty of code comprehension tasks using software metrics. In *Proceedings of the 15th Australasian Computer Education Conference (ACE'13)*. (Vol. 136, pp. 59–65). Adelaide, South Australia: Australian Computer Society Inc.

KAZMAN, R., C. CLEMENTS a L. BASS, 2012. *Software Architecture in Practice*, Third Edition. Boston, MA, USA: Addison-Wesley. ISBN 978-0321815736.

KELLY, K., B. CLARK, V. BROWN a J. Sitzia, 2003. Good practice in the conduct and reporting of survey research. In *International Journal for Quality in Health Care*. 15(3), 261-266. DOI: 10.1093/intqhc/mzg031.

KIM, B., DANYLUK, A. a T. MURTAGH, 2001. Event-driven programming is simple enough for CS1. In *ACM SIGCSE Bulletin*, 33(3). 1-4. DOI: 10.1145/377435.377440.

KINCHIN, I., D. HAY a A. ADAMS, 2000. How a qualitative approach to concept map analysis can be used to aid learning by illustrating patterns of conceptual development. In *Educational Research*, 42(1), 43–57. DOI: 10.1080/001318800363908.

KINGSTON, E., 2008. Emotional competence and drop-out rates in higher education. *Education and Training*. 50(2), 128-139. DOI: 10.1108/00400910810862119.

KLIMEŠ, C., J. SKALKA, G. LOVÁSZOVÁ, P. ŠVEC, 2008. *Informatika: pro maturanty a zájemce o studium na vysokých školách*. Nitra: Enigma. České vydání - aktualizováno a upraveno. ISBN 978-80-89132-71-3.

KÖLLING, M., 1999. The problem of teaching object-oriented programming: Part I: Languages. In *Journal of Object-Oriented Programming*. 11(8), 8-15. [online].

[cit. 2020-05-18]. Dostupné z: https://www.bluej.org/papers/1999-08-JOOP1-languages.pdf.

KONEČNÝ, J. a V VYCHODIL, 2008. *Paradigmata programování 1A* [online]. Olomouc [cit. 2019-11-10]. Dostupné z: https://phoenix.inf.upol.cz/esf/ucebni/pp1a.pdf

KOŘÍNEK, O., 2017. *Výzkum koncepcí ve výuce programování*. Hradec Králové, Disertační práce. Univerzita Hradec Králové.

KUMAR, A. N., 2013. A study of the influence of code-tracing problems on code--writing skills. In *Proceedings of the 18th conference on Innovation Technology in Computer Science Education (ITiCSE'13)*. Canterbury, United Kingdom: ACM, 183–188. ISBN 978-1-4503-2078-8. DOI: 10.1145/2462476.2462507.

KUTSAR, D. a A. KÄRNER, 2010. Exploration of societal transitions in Estonia from the threshold concepts perspective of teaching and learning. In LAND, R., J.H.F. MEYER a C. BAILLIE. *Threshold Concepts and Transformational Learning*. Rotterdam: Sense, 383-397. ISBN 978-94-6091-206-1.

LAND, R., 2012. Threshold Concepts and Troublesome Knowledge [Elon TLT] [video]. *Youtube*. [cit. 2020–05–26]. Dostupné z: https://www.youtube.com/watch?v=WR1cXIdWnNU.

LAND, R., J. RATTRAY a P. VIVIAN, 2014. Learning in the liminal space: a semiotic approach to threshold concepts. In *Higher education*. 67(2), 199-217. DOI: 10.1007/s10734-013-9705-x.

LARKIN, M., S. Watts a E. CLIFTON, 2006. Giving voice and making sense in interpretive phenomenological analysis. In *Qualitative Research in Psychology*, 3(2), 102-120. DOI: 10.1191/1478088706qp062oa.

LAURILLARD, D., 1993. *Rethinking university teaching: a conversational framework for the effective use of learning technologies*. London: Routledge. ISBN 97--804-152-5679-7.

LAVY, I., R. RASHKOVITS a R. KOURIS, 2009. Coping with abstraction in object orientation with a special focus on interface classes. In *Computer Science Education*, 19(3), 155-177. DOI: 10.1080/08993400903255218.

LEVY, M., a P. POWELL, 2003. Exploring SME Internet Adoption: towards a Contingent Model. In *Electronic Markets*, 13(2). DOI: 10.1080/1019678032000067163.

LEWIS, J., 2000. Myths about Object-Orientation and its Pedagogy. In *ACM SIGCSE Bulletin*, 32(1), 245–249. DOI: 10.1145/331795.331863.

LIBERMAN, N., Beeri, C. a Y. Kolikant, 2011. Difficulties in Learning Inheritance and Polymorphism. In *ACM Transactions on Computing Education*, 11(4), 1–23. DOI: 10.1145/1921607.1921611.

LISKOV, B. H., 1980. Modular Program Construction Using Abstractions. In BJØORNER. *Abstract Software Specifications. Lecture Notes in Computer Science*, vol 86. Springer, Berlin, Heidelberg, 354-389. DOI: 10.1007/3-540-10007-5_43.

LISKOV, B. H. a M. WING, 1994. A Behavioral Notion of Subtyping. In *ACM Trans. Program. Lang. Syst.* 16(6), 1811-1841. DOI: 10.1145/197320.197383.

LISTER, R., ADAMS E. S. a S. FITZGERALD, 2004. A Multi-National study of reading and tracing skills in novice programmers. In *ACM SIGCSE Bulletin*, 36(4), 119-150. DOI: 10.1145/1041624.1041673.

LISTER, R., CLEAR T. a B. SIMON, 2009. Naturally occurring data as research instrument: Analyzing examination responses to study the novice programmer. *ACM SIGCSE Bulletin*. 41(4), 156–173. DOI: 10.1145/1709424.1709460.

LISTER, R., FIDGE C. a D. TEAGUE, 2009. Further evidence of a relationship between explaining, tracing and writing skills in introductory programming. In *ACM SIGCSE Bulletin*, 41(3), 161–165. DOI: 10.1145/1595496.1562930.

LISTER, R., SIMON, B., THOMPSON, E. WHALLEY, J. L. a C. PRASAD, 2006. Not seeing the forest for the trees: Novice programmers and the SOLO taxonomy. In

Proceedings of the 11th annual SIGCSE conference on Innovation and Technology in Computer Science Education (ITICSE'06). Bologna, Italy: ACM, 118–122. ISBN 978-1-59593-055-2 DOI: 10.1145/1140124.1140157

LISTER, R., SCHULTE, C. a J. L. WHALLEY, 2006. Research perspectives on the objects-early debate. In *ACM SIGCSE Bulletin*. 34(4), 146–165. DOI: 10.1145/1189136.1189183.

LISTER, R., CLEAR, T. a B. SIMON, 2009a. Naturally occurring data as research instrument: Analyzing examination responses to study the novice programmer. In *ACM SIGCSE Bulletin*, 41(4), 156–173. DOI: 10.1145/1709424.1709460.

LISTER, R., FIDGE, C., a D. TEAGUE, 2009b. Further evidence of a relationship between explaining, tracing and writing skills in introductory programming. In *Proceedings of the 9th conference on Innovation and Technology in Computer Science Education (ITiCSE'09)*. (41)3, pp. 161–165). Leeds, United Kingdom: ACM. DOI: 10.1145/1595496.1562930.

LOGO, 2021. Logo Programming Language [online]. *Logo Foundation*. Boston: MIT, 2021 [cit. 2021-02-26]. Dostupné z: https://el.media.mit.edu/logo-foundation/what_is_logo/logo_programming.html.

LOH, J., 2013. Inquiry into Issues of Trustworthiness and Quality in Narrative Studies: A Perspective. In *The Qualitative Report*. 18(33), 1-15. DOI: 10.46743/2160--3715/2013.1477.

LOPEZ, M., WHALLEY, J., ROBBINS, P. a R. LISTER, 2008. Relationships between reading, tracing and writing skills in introductory programming. In *Proceedings of the 14th International workshop on Computing Education Research (ICER'08)*. Sydney, Australia: ACM, 101–112. ISBN 978-1-60558-216-0. DOI: 10.1145/1404520.1404531

LORENZ, M. a J. KIDD, 1994. *Object-Oriented Software Metrics: a practical guide*. Englewood Cliffs, NJ: PTR Prentice Hall. ISBN 978-0-131-79292-0.

LUI, A. K., KWAN, R., POON, M. a Y. H. CHEUNG, 2004. Saving weak programming students: applying constructivism in a first programming course. In *ACM SIGCSE Bulletin*, 36(2), 72–76. DOI: 10.1145/1024338.1024376.

LUKER, P. A., 1989. Never mind the language, what about the paradigm? In *ACM SIGCSE Bulletin*, 21(1), 252–256. DOI: 10.1145/65294.71442.

MAŇÁK, J. a V. ŠVEC, 2003. *Výukové metody*. Brno: Paido. ISBN: 80-7315-039-5.

MAGENHEIM, J., NELLES, W., RHODE, T. a N. SCHAPER, 2010. Towards a methodical approach for an empirically proofed competency model. In HROMKOVIČ, J., R. KRÁLOVIČ a J. VAHRENHOLD. *Teaching Fundamentals Concepts of Informatics. ISSEP 2010*. Lecture Notes in Computer Science, vol 5941. Berlin: Springer, 124–135. DOI: 10.1007/978-3-642-11376-5_12

MAREŠ, J., 1998. *Styly učení žáků a studentů*. Praha: Portál. ISBN: 80-7178-246-7.

MARTIN, R. C., 2000. Design Principles and Design Patterns. [online]. *Object Mentor*. [cit. 2020-02-07] Dostupné z: http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf.

MARTIN, R. C., 2009. *Clean code: a handbook of agile software craftsmanship*. Upper Saddle River, NJ: Prentice Hall. ISBN 978-0-13-235088-4.

MARTIN, R. C., 2017. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Boston: Pearson. ISBN 978-0-13-449416-6.

MARTON, F., 1981. Phenomenography—Describing Conceptions of the World Around Us. In *Instructional Science*, 10, 177–200. DOI: 10.1007/BF00132516.

MARTON, F., 1986. Phenomenography: A research approach investigating different understandings of reality. In *Journal of Thought*, 21(3), 28–49.

MARTON, F. a S. BOOTH, 1997. The Idea of Phenomenography. In MARTON, F. a S. BOOTH. *Learning and Awareness*. New York: Routledge, 110–136. ISBN 97--802-030-5369-0. DOI: 10.4324/9780203053690

MARTON, F. a S. BOOTH, 1998. The Learner's Experience of Learning. In OLSON,

D. R. a N. TORRANCE. *The Handbook of Education and Human Development: New Models of Learning, Teaching and Schooling*. London: Blackwell Publishers Ltd., 513-541. ISBN:9780631211860. DOI: 10.1111/b.9780631211860.1998.00025.x.

MARSHALL, C. a G. B. ROSSMAN, 2006. *Designing Qualitative Research* (Fourth ed.). Thousand Oaks: Sage Publications. ISBN 978-1412924887.

MCCABE, T., 1976. A Complexity Measure. In *IEEE Transactions On Software Engineering*. SE-2(4), 308-320. DOI: 10.1109/TSE.1976.233837.

MCCLURE, J., SONAK, B. a H. SUEN, 1999. Concept map assessment of classroom learning: Reliability, validity, and logistical practicality. In *Journal of Research in Science Teaching*, 36(4):475–492. DOI: 10.1002/(SICI)1098-2736(199904)36:4<475::AID-TEA5>3.0.CO;2-O.

MCCRACKEN, M., ALMSTRUM, V., DIAZ, D., THOMAS, L., GUZDIAL, M., UTTING, I. a D. HAGAN, 2001. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students A framework for first-year learning objectives. In *ITiCSE-WGR '01: Working group reports from ITiCSE on Innovation and technology in computer science education*, 125–180. ISBN 978-1-4503-7359--3. DOI: 10.1145/572133.572137.

MERRIAM, S. B., 2015. *Qualitative Research: A Guide to Design and Implementation*. San Francisco, CA: Jossey Bass. 4. vyd. ISBN: 978-1-119-00361-8.

METZ, S., 2018. *Practical Object-Oriented Design: An Agile Primer Using Ruby*, Second edition. Boston, MA, USA: Addison-Wesley. ISBN 978-0134456478.

MEYER, B., 1997. *Object-oriented software construction*. New York: Prentice Hall. ISBN 978-0-13-6291558-X.

MEYER, J. a R. LAND, 2003. Threshold concepts and troublesome knowledge: Linkages to ways of thinking and practising within the disciplines. In RUST, C. *Improving Student Learning – Theory and Practice Ten Years On*. Oxford: Oxford Centre for Staff and Learning Development (OCSLD), 412-424. ISBN 978-1873576694. ETL Project Occasional Report 4.

MEYER, J., a R. Land, 2005. Threshold Concepts and Troublesome Knowledge (2): Epistemological Considerations and a Conceptual Framework for Teaching and Learning. In *Higher Education*, 49, 373–388 (2005). DOI: 10.1007/s10734--004-6779-5.

MICROSOFT, 2021a. Members - C♯ Programming Guide. In *Developer tools, technical documentation and coding examples*. Wagner, B. [cit. 2021-05-21]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/members

MICROSOFT, 2021b. Polymorphism. In *Developer tools, technical documentation and coding examples*. Wagner, B. [cit. 2021-05-21]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/object-oriented/polymorphism

MITCHELL, W., 2000. A paradigm shift to OOP has occurred...implementation to follow. In BRYANT R. a D. HANSEN. *Proceedings of the second annual CCSC on Computing in Small Colleges Northwestern konference, October 2000. Journal of Computing Sciences in Colleges*, 16, 94-105.

MOSTRÖM, J. E., 2011. *A Study of Student Problems in Learning to Program*. Umea, Švédsko, 2011. Disertační práce. Umea University.

MÜHLING, A. M., 2014. *Investigating Knowledge Structures in Computer Science Education*, 2014. Disertační práce. Technische Universität München, Germany.

MULLER, O., 2005. Pattern oriented instruction and the enhancement of analogical reasoning. In *Proceedings of the 5th International workshop on Computing Education Research (ICER'05)*. Seattle, WA, USA: ACM, 57–67. ISBN 978-1-59593--043-9. DOI: 10.1145/1089786.1089792

MURPHY, L., FITZGERALD, S., LISTER, R. a R. McCAULEY, 2012. Ability to "Explain in Plain English" linked to proficiency in computer-based programming. In *Proceedings of the 9th International workshop on Computing Education Research (ICER'12)*. Auckland, New Zealand: ACM, 111–118. ISBN 978-1-4503-1604-0. DOI:

10.1145/2361276.2361299

MYERS, M. D., 2009. *Qualitative research in business and management*. London, UK: Sage Publications. ISBN 978-0857029744.

NASH, J., R. BRAVACO a S. SIMONSON, 2006. Assessing knowledge change in computer science. In *Computer Science Education*, 16(1), 37–51. DOI: 10.1080/08993400500430362.

NICOLETTE, D., 2015. *Software Development Metrics*. Shelter Island, NY, USA: Manning Publications. ISBN 9781617291357.

NOVAK, J. a D. GOWIN, 1984. *Learning How to Learn*. Cambridge: Cambridge University Press. ISBN 9781139173469. DOI: 10.1017/CBO9781139173469

OATES, B. J., 2006. *Researching information systems and computing*. Sage Publications Limited. ISBN 978-141290224-3.

OLIVER, D., DOBELE, T., GREBER, M. a T. ROBERTS, 2004. This course has a Bloom rating of 3.9. In *Proceedings of the 6th Australasian Computing Education Conference – Volume 30 (ACE'04)*. Dunedin, NZ: Australian Computer Society, Inc., 227–231. ISBN 1920682120.

PABIAN, P., 2012. Jak se učí na vysokých školách: Výzkumný směr k učení přístupů. In *AULA*, 1 (1), 48–77. ISSN 1210-6658.

PATTON, M. Q., 2014. *Qualitative research and evaluation methods*. London: SAGE Publications Ltd. 4. vyd. ISBN 978-1412972123.

PEARS, A., SEIDMAN, S., MALMI, L., MANNILA, L., ADAMS, E., BENNEDSEN, J., DEVLIN, M. a J. PATERSON, 2007. A survey of literature on the teaching of introductory programming. In *ACM SIGCSE Bulletin*, 39 (4), 204 – 223. DOI: 10.1145/1345375.1345441

PECINOVSKÝ, R., 2006. Výuka programování podle metodiky design patterns first, In *Konference Tvorba software 2006*, Ostrava, VŠB-Technická univerzita Ostrava, 5.-6.6.2006. [online]. [cit. 2019-05-08]. Dostupné z: http://prog-story.

technicalmuseum.cz/index.php/m-virtualni-sbirky-tm-v-brne/digitalizovane-kolekce/programovani-tvorba-sw-ostrava/2005-2014/2006-tvorba-softwaru-ostrava/5448-2006-vyuka-programovani-podle-metodiky-design-patterns-first

PECINOVSKÝ, R., 2007. Metodika výuky programování na rozcestí. In *Poškole 2007*. Praha: MOV Poškole, 48-57. ISBN 978-80-239-9126-0. [online]. [cit. 2020--02-21]. Dostupné z: http://vyuka.pecinovsky.cz/prispevky/2007_Po_Metodika_vyuky_na_rozcesti.pdf

PECINOVSKÝ, R., 2008. Mýty ve výuce programování a metodika Design Patterns First. In *Objekty 2008*. Žilina: Žilinská univerzita FRI, 17-27. ISBN 978-80-8070--927-3. [online]. [cit. 2020-03-05]. Dostupné z: http://vyuka.pecinovsky.cz/prispevky/2008-B_MytyVeVyuceOOPaDPF.pdf

PECINOVSKÝ, R., 2009. *Myslíme objektově v jazyku Java: kompletní učebnice pro začátečníky*. 2., aktualiz. a rozš. vyd. Praha: Grada. ISBN 978-80-247-2653-3.

PECINOVSKÝ, R., 2010. Metodika Design Patterns First v roce 2010. In *Objekty 2010*. Ostrava: Ostravská univerzita, 207-217. ISBN 978-80-7368-899-8. [online] [cit. 2019-10-23] Dostupné z: http://vyuka.pecinovsky.cz/prispevky/2010_OB_Metodika%20DPF%20v%20roce%202010.pdf

PECINOVSKÝ, R., 2013a. Metodika Architecture First. In *Journal of Technology and Information Education: Časopis pro technickou a informační výchovu*, 5(1), 107–114. DOI: 10.5507/jtie.2013.016. 2013. ISSN 1803-537X.

PECINOVSKÝ, R., 2013b. *OOP – learn object oriented thinking and programming*. Řepín: Tomáš Bruckner. Academic series. ISBN 978-80-904661-8-0.

PECINOVSKÝ, R., 2015. Metodika Architecture First. In *DidInfo 2015*: 21. ročník národnej konferencie [online]. Banská Bystrica: Univerzita Mateja Bela, Fakulta prírodných vied v Banskej Bystrici, 118-122 [cit. 2019-05-10]. ISBN 978-80-557--0852-2. Dostupné z: https://zssnpbb.edupage.org/files/Didinfo2015i.pdf

PETTY, G., 2013. *Moderní vyučování*. Praha: Portál. ISBN 978-80-262-0367-4.

PERKINS, D., HANCOCK, C., HOBBS, M., FAY, S. a R. SIMMONS, 1989. Conditions of learning in novice programmers. In SOLOWAY, E. a J. C. SPOHRER. *Studying the novice programmer*. Hillsdale, NJ: Lawrence Erlbaum, ch. 13, 20 s. ISBN 9781315808321.

PERKINS, D., 2008. Beyond understanding. In LAND, R., J. H. F. MEYER A J. SMITH. *Threshold Concepts within the Disciplines*. Rotterdam: Sense Publishers, 2008, 1–19. ISBN: 978-90-8790-268-1. DOI: 10.1163/9789460911477_002.

PIAGET, J., 1999. *Psychologie inteligence*. Praha: SPN. ISBN: 80-7178-309-9.

POLYA, G. a J. H. CONWAY, 2015. *How to Solve It: A New Aspect of Mathematical Method*. Princeton: Princeton University Press; Princeton Science Li edition. ISBN 978-0691119663.

PUNCH, K. F., 2013. *Introduction to Social Research: Quantitative and Qualitative Approaches*. London: Sage Publications Inc., 3. vyd., ISBN 978-1446240939.

QUINNELL, R. a R. THOMPSON, 2010. Conceptual intersections: re-viewing academic numeracy in the tertiary education sector as a threshold concept. In LAND, R., J.H.F. MEYER a C. BAILLIE. *Threshold Concepts and Transformational Learning*. Rotterdam: Sense, 147-145. ISBN 978-94-6091-206-1. DOI: 10.1163/9789460912078_010.

REID, K., FLOWERS, P. a M. LARKIN, 2005. Exploring lived experience. In *The Psychologist*, 18(1), 20-23. ISSN 0952-8229.

ROBERTS, G. W., 2010. Advancing new approaches to learning and teaching - introducing appreciative inquiry to a problem-based learning curriculum. In *Journal of Applied Research in Higher Education*, 2(1), 15 -24. DOI: 10.1108/17581184201000002.

ROBINS, A., ROUNTREE, J. a N. ROUNTREE, 2003. Learning and teaching programming: a review and discussion. In *Computer Science Education*. 13(2), 137–172. DOI: 10.1076/csed.13.2.137.14200.

ROBINS, A., 2010. Learning edge momentum: a new account of outcomes in CS1. In *Computer Science Education*, 20(1), 37–71. DOI: 10.1080/08993401003612167.

REGES, S., 2006. Back to basics in CS1 and CS2. In BALDWIN, D., P. TYMANN, S. HALLER a I. RUSSELL. *Proceedings of the 37th SIGCSE technical symposium on Computer Science Education (SIGCSE'06)*. Houston, Texas, USA: ACM, 293–297. ISBN 978-1-59593-259-4. DOI: 10.1145/1121341.1121432

REYNOLDS-HAERTLE, R., 2002. *OOP: objektově orientované programování Visual Basic .NET, Visual C♯ .NET krok za krokem*. Praha: Mobil Media. ISBN 80-86593--25-8.

ROUNTREE, J. a N. ROUNTREE, 2009. Issues Regarding Threshold Concepts in Computer Science. In *Proc. 11th Australasian Conference on Computing Education - Volume 95 (ACE '09)*. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 139–146. [online]. [cit. 2020-06-18]. Dostupné z: http://dl.acm.org/citation.cfm?id=1862712.1862733.

SAELI, M., PERRENET, J., WIM, M., JOCHEMS, G. a B. ZWANEVELD, 2011. Teaching programming in secondary school: A pedagogical content knowledge perspective. In *Informatics in Education*, 10(1), 73–88. DOI:10.15388/INFEDU. 2011.06.

SANDELOWSKI, M., 2000. Whatever Happened to Qualitative Description? In *Research in Nursing & Health*, 23(4), 334-340. DOI: 10.1002/1098-240x(200008)23: 4<334::aid-nur9>3.0.co;2-g.

SANDERS, K., BOUSTEDT, J., ECKERDAL, A., MCCARTNEY, R., MOSTRÖM, J., THOMAS, L. a C. ZANDER, 2008. Student understanding of object-oriented programming as expressed in concept maps. In *ACM SIGCSE Bulletin*, 40(3), 332-336. DOI: 10.1145/1352135.1352251.

SANDERS, K., a R. MCCARTNEY, 2016. Threshold Concepts in Computing: Past, Present, and Future. In SHEARD, J. a C. S. MONTERO. *Proceedings of the 16th Koli Calling International Conference on Computing Education (Koli Calling '16)*.

Koli Finland, November 24-27, 2016. New York: ACM, 91–100. ISBN 978-1-4503--4770-9. https://doi.org/10.1145/2999541.2999546.

SANDBERG, J., 1994. *Human competence at work*. Göteborg. Dizertační práce. Department of Educational Sciences, University of Göteborg.

SANDERS, K. a R. McCARTNEY, 2016. Threshold Concepts in Computing: Past, Present, and Future. In *Proc. 16th Koli Calling Int'l Conference on Computing Education Research (Koli Calling '16)*. New York, NY, USA: ACM, 91–100. DOI: 10.1145/2999541.2999546

SATTAR, A., a T. LORENZEN, 2009. Teach Alice programming to non-majors. In *ACM SIGCSE Bulletin*, 41(2), 118–121. DOI: 10.1145/1595453.1595488.

SHANAHAN, M. P., FOSTER, G. a J. H. F. MEYER, 2010. Threshold concepts and attrition in first-year economics. In LAND, R., J. H. F. MEYER a C. BAILLIE. *Threshold Concepts and Transformational Learning*. Rotterdam: Sense, 207-226. ISBN 978-94-6091-206-1. DOI: 10.1163/9789460912078_014.

SCHEJA, M., 2006. Delayed Understanding and Staying in Phase: Students' Perceptions of their Study Situation. In *Higher Education*, 52(3), 421-445. DOI: 10.1007/s10734-004-7765-7.

SCHILDT, H., 2012. *Java 7: Výukový kurz*. Brno: Computer Press. ISBN 978-80-251--3748-2.

SCRATCH, 2021. Scratch [online]. Boston: MIT Media Lab, 2021 [cit. 2021-06-06]. Dostupné z: https://scratch.mit.edu/

SFARD, A., 1998. On Two Metaphors for Learning and the Dangers of Choosing Just One. In *Educational Researcher*, 27(2), pp. 4-13. DOI: 10.3102/ 0013189X027002004.

SJÖSTRÖM, B., a L. DAHLGREN, 2002. Applying phenomenography in nursing research. In *Journal of Advanced Nursing*, 40 (3), 339–345. DOI: 10.1046/j.1365--2648.2002.02375.x.

SMITH, J. A., P. FLOWERS a M. LARKIN, 2009. *Interpretative phenomenological analysis: theory, method and research*. London: SAGE Publications Ltd. ISBN: 9781412908337

SOLE, A. D., 2012. *Microsoft Visual Studio LightSwitch Unleashed*. Indianapolis, IN, USA: Sams. ISBN 978-0672335532

SOLOWAY, E., a J. C. SPOHRER, 1989. *Studying the Novice Programmer*. Hillsdale, New Jersey: Lawrence Erlbaum Associates. ISBN-13 : 978-0805800036. Kindle version: 2013. ASIN .B00H1S1EEU

SORVA, J., 2013. Notional machines and introductory programming education. In *ACM Transactions on Computing Education*, 13(2), 1–31. DOI: 10.1145/2483710.2483713.

SPENCER, L., RITCHIE, J. a W. O. CONNOR, 2003. Analysis: Practices, Principles and Processes. In RITCHIE, J., J. LEWIS a R. ORMSTON. *Qualitative Research Practice*. London: Sage Publications, 199 218. ISBN 9781446209110

SPOHRER, J. C. a E. SOLOWAY1986. Novice mistakes: are the folk wisdoms correct? In *Communications of the ACM*, 29(7), 624–632. DOI: 10.1145/6138.6145.

STAUDKOVÁ, H.. 2016. *Uplatnění fenomenografického přístupu na příkladu výzkumu využívání digitálních technologií ve vzdělávání*. Pedagogická orientace, 2016, 26(3), 442–456. DOI: 10.5817/PedOr2016-3-442.

STEYVERS, M. a J. TENENBAUM, 2005. Graph theoretic analyses of semantic networks: Small worlds in semantic networks. In *Cognitive Science*, 29, 41–78.

STOLZ, S. A., 2020. Phenomenology and phenomenography in educational research: A critique. In *Educational Philosophy and Theory*. Routledge. 52(10), 1077–1096. DOI: 10.1080/00131857.2020.1724088.

ŠVAŘÍČEK, R. a K. ŠEĎOVÁ, 2014. *Kvalitativní výzkum v pedagogických vědách*. Vyd. 2, Praha: Portál. ISBN 978-80-262-0644-6.

TALANQUER, V., 2015. Threshold Concepts in Chemistry: The Critical Role of Im-

plicit Schemas. In *Journal of Chemical Education*, 92, 3-9. DOI: 10.1021/ed500679k.

TAYLOR, C., 2006. Threshold concepts in Biology. In MEYER, R.J.H.F. a R. LAND. *Overcoming Barriers to Student Understanding*. London: Routledge, 87 - 99. ISBN 9780415514187.

TAYLOR, C. E., a J. H. F. MEYER, 2010. The testable hypothesis as a threshold concept for biology students. In LAND, R., J. H. F. MEYER a C. BAILLIE. *Threshold Concepts and Transformational Learning*. Rotterdam: Sense, 179–192. ISBN 978--94-6091-206-1. DOI: 10.1163/9789460912078_012.

THOMAS, L., ZANDER, C., LOFTUS, Ch. a A. ECKERDAL, 2017. Student Software Designs at the Undergraduate Midpoint. In DAVOLI, R. a M. GOLDWEBER. *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '17)*. Bologna Italy, July 3-5, 2017. New York: ACM, 34–39. ISBN 978-1-4503-4704-4. DOI: 10.1145/3059009.3059016.

TIGHT, M., 2015. Phenomenography: The development and application of an innovative research design in higher education research. In *International Journal of Social Research Metodology*, 19(3), 1–20. DOI: 10.1080/13645579.2015.1010284.

TIMMERMANS, J. A. a J. H. F. MEYER, 2017. A framework for working with university teachers to create and embed 'Integrated Threshold Concept Knowledge' (ITCK) in their practice. In *International Journal for Academic Development*, 24(4), 354-368. DOI: 10.1080/1360144X.2017.1388241

THE BLUE PAGE, 2021. THE BLUE PAGE – TEACHING OBJECT ORIENTED PROGRAMMING (2021). [online] *University of Kent*.[cit. 2021-05-22] Dostupné z: http://www.cs.kent.ac.uk/people/staff/mik/blue/.

TSUI, F. F., KARAM, O. a B. BERNAL, 2017. *Essentials of Software Engineering*, 4. vyd. Burlington, MA, USA: Jones & Bartlett. ISBN 978-1284106008.

TUCKER, V., WEEDMAN, J., BRUCE, C. a S. EDWARDS, 2014. Learning Portals: Analyzing Threshold Concept Theory for LIS Education. In *Journal of Education*

for Library and Information Science, 55(2), 150-165.

ULJENS, M., 1996. On the philosophical foundations of phenomenography. In DALLALBA, G. a B. HASSELGREN. *Reflections on Phenomenography*. Göteborg: Acta universitatis Gothoburgensis, 105–130. ISBN 9173462993.

VAN MANEN, M. (2017). But is it phenomenology? In *Qualitative Health Research*, 27(6), 775–779. doi:10.1177/1049732317699570

VAN ROY, P., ARMSTRONG, J., FLATT, M. a B. MAGNUSSON, 2003. The role of language paradigms in teaching programming. In SCOTT, G. *Proceedings of the 34th SIGCSE technical symposium on Computer science education*, Reno, February 19-23 2003. New York: ACM Press, 269–270. ISBN 978-1-58113-648-7. DOI: 10.1145/611892.611908.

VENABLES, A., TAN, G. a R. LISTER, 2009. A closer look at tracing, explaining and code writing skills in the novice programmer. In CLANCY, M., M. CASPERSEN a R. LISTER. *Proceedings of the 5th International workshop on Computing Education Research (ICER'09)*. Berkeley, CA, USA: ACM, 117–128. ISBN 978-1-60558-615-1. DOI: 10.1145/1584322.1584336

VUJOŠEVIĆ-JANIČIĆ, M. a D. TOŠIĆ, 2008. The role of programming paradigms in the first programming courses [online]. *The Teaching of Mathematics*. 11(2), 63-83 [cit. 2020-02-11]. ISSN 2406-1077. Dostupné z: http://elib.mi.sanu.ac.rs/files/journals/tm/21/tm1122.pdf.

WALKER, D., TOPPING, K. a S. RODRIGUES, 2008. Student reflections on formative e-assessment: expectations and perceptions. In *Learning, Media and Technology*, 33(3), 221-234. DOI: 10.1080/17439880802324178.

WANKAT, P. C. a F. S. OREOVICZ, 2015. *Teaching Engineering*. West Lafayette: Purdue University Press. 2. revidované vyd. ISBN 978-1557537003.

WASSBERG, J., 2020. *Computer programming for absolute beginners: learn essential computer science concepts and coding techniques to kick-start your programming career*. Birmingham, UK: Packt. ISBN 978-1-83921-253-6.

WATKINS, Ch., 2011. Collaborative learning. In *Creative Teaching and Learning*, 2(1), 8-11. Též dostupné z: https://www.chriswatkins.net/download/102/

WEINERT, F. E., 2001. Concept of competence: A conceptual clarification. In RYCHEN D. S. a L. H. SALGANIK. *Defining and selecting key competencies*. Seattle: Hogrefe & Huber, 45–65. ISBN 2-7475-1523-0.

WEINTROP, D. a U. WILENSKY, 2016. Bringing blocks-based program ming into high school computer science classrooms. In *Annual Meeting of the American Educational Research Association (AERA)*. Washington, DC: AERA [online]. [cit. 2021-02-13]. Dostupné z: https://ccl.northwestern.edu/2016/Weintrop_Wilensky_AERA_2016.pdf.

WEBB, A.S., 2016. Threshold Concepts and the Scholarship of Teaching and Learning. In R. LAND, R., J. H. F. MEYER a M. FLANAGAN. *Threshold Concepts in Practice*. Rotterdam: Sense, 299-308. ISBN 978-94-6300-511-1 DOI: 10.1163/9789463005128_023.

WEISFELD, M., 2019. *The Object-Oriented Thought Process*, Fifth Edition. Boston, MA, USA: Addison-Wesley. ISBN 9780135182130.

WHALLEY, J., LISTER, R., THOMPSON, E., CLEAR, T., ROBBINS, P. a C. PRASAD, 2006. An Australasian study of reading and comprehension skills in novice programmers, using the Bloom and SOLO Taxonomies. In: TOLHURST, D. a S. MANN. *Proceedings of the 8th Australasian Computing Education Conference - Volume 52 (ACE '06)*. Vol. 52. Darlinghurst, Australia: Australian Computer Society, Inc., 243–252. ISBN 978-1-920682-34-7.

WHALLEY, J., CLEAR, T., ROBBINS, P. a E. THOMPSON, 2011. Salient elements in novice solutions to code writing problems. In: HAMER, J. a M. de RAADT. *Proceedings of the 13th Australasian Computing Education Conference – Volume 114 (ACE'11)*. Darlinghurst, Australia: Australian Computer Society, Inc., 37–46. ISBN 978-1-920682-94-1.

WILSON, L. O., 2001. Anderson and Krathwohl – Bloom's Taxonomy Revised: Un-

derstanding the New Version of Bloom's Taxonomy. *The Second Principle*. [cit. 2018-08-30]. Dostupné z: https://thesecondprinciple.com/teaching-essentials/beyond-bloom-cognitive-taxonomy-revised/

WINSLOW, L. E., 1996. Programming pedagogy-a psychological overview. In *ACM SIGCSE Bulletin*, 28(3), 17–22. DOI: 10.1145/234867.234872

YAMAMOTO, M., SEKIYA, T., MORI, K. a K. YAMAGUCHI, 2012. Skill hierarchy revised by SEM and additional skills. In *Proceedings of the International Conference on Information Technology Based Higher Education and Training (ITHET'12)*. Istanbul: IEEE, 1–8. DOI: 10.1109/ITHET.2012.6246009.

YATES, C., PARTRIDGE, H. a C. BRUCE, 2012. Exploring information experiences through phenomenography. In *Library and Information Research*, 36, 96-119. DOI: 10.29173/lirg496.

YEOMANS, L., ZSCHALER, S. a K. STEFFEN, 2019. Transformative and Troublesome? Students' and Professional Programmers' Perspectives on Difficult Concepts in Programming. In *ACM Transactions on Computing Education*. 19(3), 1-27. DOI: 10.1145/3283071.

YIN, R. K., 2013. *Case Study Research: Design and methods*. 5. vyd. Thousand Oaks, CA: Sage Publications. ISBN 978-1452242569.

Seznam obrázků

Seznam tabulek

Seznam výpisů

Seznam použitých zkratek

.NET .NET Framework, soubor technologií v softwarových produktech

ACM Association for Computing Machinery, mezinárodní učená společnost působící v oblasti výpočetní techniky

API Application Programming Interface, rozhraní pro programování aplikací

CLR Common Language Runtime, Společný jazykový runtime modul, součást virtuálního stroje Microsoft .NET Framework, řídí spouštění programů .NET

DIT Depth of Inheritance Tree, maximální délka cesty ze třídy do kořenové třídy v dědičné struktuře systému (hloubka stromu dědičnosti)

ID Identification, identifikace

IEEE Institute of Electrical and Electronics Engineers, Institut pro elektrotechnické a elektronické inženýrství

IPA Interpretive Phenomenological Analysis, interpretační fenomenologická analýza

IT Informační technologie

LOC Lines of Code, počet logických řádek zdrojového kódu

OOD Objektově orientovaný design, objektově orientovaný návrh

OOP Objektově orientované programování

PHP PHP: Hypertext Preprocessor, Hypertextový preprocesor, skriptovací programovací jazyk

SOLID mnemonický akronym pro pět principů návrhu, jejichž cílem je učinit návrhy softwaru srozumitelnějšími, flexibilnějšími a udržovatelnějšími.

SOLO Structure of Observed Learning Outcomes, struktura pozorovaných výsledků učení

TC Threshold Concepts, prahové koncepty

UML Unified Modeling Language, Unifikovaný Modelovací Jazyk, grafický jazyk pro vizualizaci, specifikaci, navrhování a dokumentaci programových systémů

Přílohy

A Souhlas s účastí na výzkumu

B Otázky pro rozhovor ke konceptu

C Dotazník k řešení úkolu

D Agregovaná konceptuální mapa

E Myšlenková mapa

F Ukázka zadání úkolu Hodiny a displeje

G Ukázka zadání úkolu Auto konfigurátor

H Ukázka řešení úkolu Hodiny a displeje

I Ukázka řešení úkolu Auto konfigurátor

J Ukázka přepisu mluvení nahlas (úkol Postavy)

K Ukázka přepisu mluvení nahlas (úkol Kavárna)

A Souhlas s účastí na výzkumu

Informovaný souhlas účastníka výzkumu

Vážená paní, vážený pane,

v souladu se zásadami etické realizace výzkumu[1] Vás žádám o souhlas s Vaší účastí ve výzkumném projektu v rámci disertační práce.

Název projektu Výuka objektově orientovaného programování pro začínající programátory

Řešitel projektu Mgr. Radim Remeš (inrem@ef.jcu.cz)

Název pracoviště Katedra informatiky, Pedagogické fakulty Jihočeské univerzity v Českých Budějovicích.

Vedoucí práce doc. Ing. Ladislav Beránek, CSc., MBA (beranek@ef.jcu.cz)

Cíl výzkumu Pochopení hlavních konceptů objektově orientovaného programování studenty začínajících se učit programovat

Popis výzkumu Výzkum bude probíhat během aktuálního akademického roku. Zapojení účastníků do výzkumu bude probíhat formou sledování účastníků při řešení tématicky zaměřených zadaných programovacích úloh a následného rozboru jejich zpracovaných řešení. V průběhu řešení budou dle technických možností využita záznamová zařízení pro možnost následné analýzy průběhu řešení úkolů. Součástí účasti na výzkumu bude též realizace polostrukturovaných rozhovorů a vyplnění odpovědí na zadané dotazníky. Výzkum bude probíhat jak souběžně s rozvrhovanou výukou účastníků výzkumu, tak i v čase mimo plánovanou

[1] Všeobecnou deklarací lidských práv, nařízením Evropského parlamentu a Rady (EU) č. 2016/679 o ochraně fyzických osob v souvislosti se zpracováním osobních údajů a o volném pohybu těchto údajů a o zrušení směrnice 95/46/ES (obecné nařízení o ochraně osobních údajů) a dalšími obecně závaznými právními předpisy (jimiž jsou zejména Helsinská deklarace přijatá 18. Světovým zdravotnickým shromážděním v roce 1964, ve znění pozdějších změn (Fortaleza, Brazílie, 2013), zákon č. 372/2011 Sb., o zdravotních službách a podmínkách jejich poskytování (zákon o zdravotních službách), ve znění pozdějších předpisů, zejména ustanovení jeho § 28 odst. 1, a Úmluva na ochranu lidských práv a důstojnosti lidské bytosti v souvislosti s aplikací biologie a medicíny: Úmluva o lidských právech a biomedicíně publikované pod č. 96/2001 Sb. m. s., jsou-li aplikovatelné).

výuku. Účast na výzkumu je dobrovolná a je možné ji kdykoliv v průběhu výzkumu ukončit.

Získaná data od účastníků výzkumu budou využita pouze pro výzkumné a vědecké účely. Data budou uchovávána a předávána pouze bez identifikačních údajů (tj. jako anonymní data). Jméno ani získaný multimediální materiál zapojených osob do výzkumu se nebudou vyskytovat v publikovaných výstupech z tohoto výzkumu. Získané materiály mohou být zveřejněny v textové nebo grafické podobě v publikovaných výstupech nebo na konferencích z tohoto výzkumu se zachováním anonymity osob zúčastněných na výzkumu.

datum a podpis řešitele projektu

Prohlášení a souhlas účastníka s jeho zapojením do výzkumu

Prohlašuji a svým níže uvedeným vlastnoručním podpisem potvrzuji, že dobrovolně souhlasím s účastí ve výše uvedeném projektu a že jsem měl/a možnost si řádně a v dostatečném čase zvážit všechny relevantní informace o výzkumu, zeptat se na vše podstatné týkající se účasti ve výzkumu a že jsem dostal/a jasné a srozumitelné odpovědi na své dotazy. Byl/a jsem poučen/a o právu odmítnout účast ve výzkumném projektu nebo svůj souhlas kdykoli odvolat bez represí.

Jméno a příjmení účastníka

Podpis účastníka

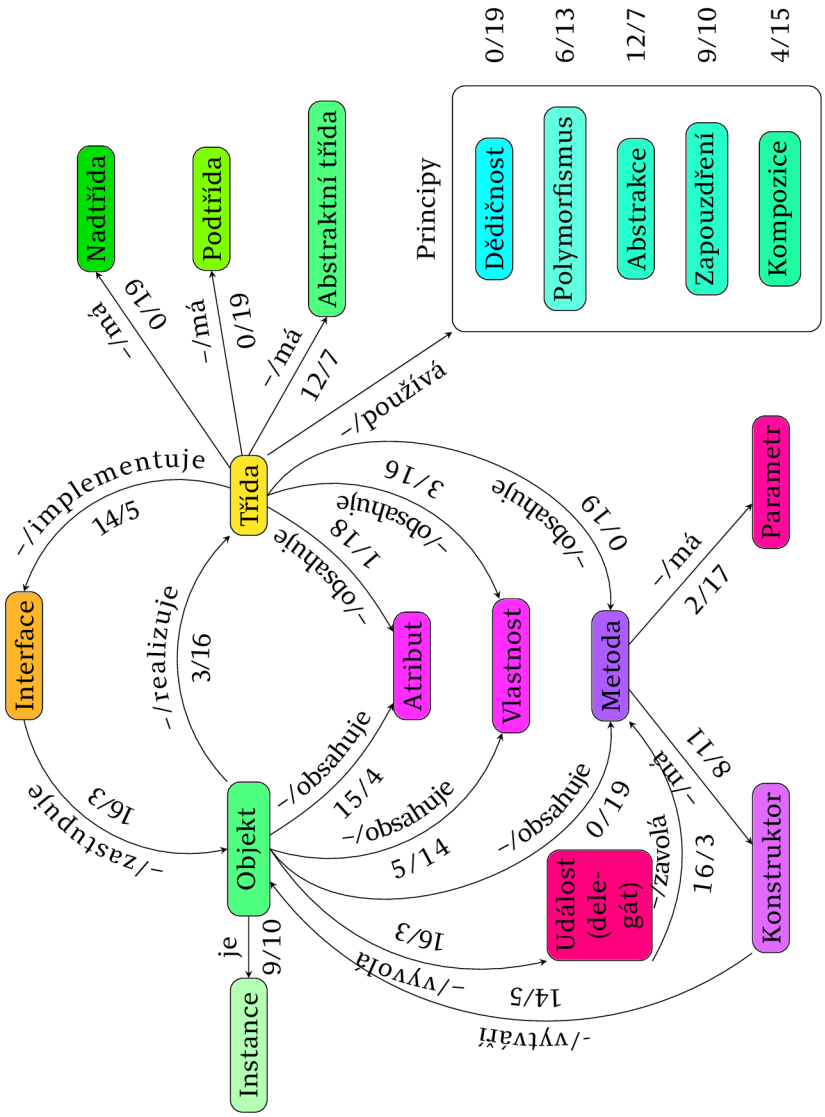
B Otázky pro rozhovor ke konceptu

1. Jaké aplikace (pomůcky) používáte při programování?
2. Mohu začít tím, že Vás požádám, abyste mi řekli, jak rozumíte *konceptu*?
3. Představte si, že byste koncept vysvětloval někomu, kdo se jej právě snaží naučit. Jak byste to provedli?
4. Můžete mi říct, co Vám pomohlo pochopit *koncept*?
5. Můžete popsat, jak jste vnímali (prožívali) *koncept* během doby, co jste se se jej snažili osvojit a jak jste totéž vnímali/prožívali později?
6. Můžete popsat, jaké jste volili postupy během doby, co jste se *koncept* snažili osvojit?
7. Na základě svých zkušeností, jakou radu byste dali ostatním studentům, kteří by se mohli potýkat s *konceptem*?
8. Řekněte mi, prosím, jaké další věci potřebujete k tomu, abyste dobře porozuměli *konceptu*?
9. Můžete mi říci, jak pochopení *konceptu* ovlivnilo vaše chápání ostatních věcí?
10. Bylo vaše pochopení *konceptu* něčím, co jste museli neustále zkoumat nebo jste se to jednou naučili a už to bylo jasné?
11. Popište, jak a v jakém kontextu jste *koncept* použili od doby, co jste se toto téma naučili.
12. Můžete popsat, jakékoliv další situace, kde jste se nejprve zasekli, ale pak se situace vyjasnila a pokračovali jste dále?
13. Je ještě něco dalšího, co byste chtěli sdělit nebo se zeptat ohledně *konceptu*?

C Dotazník k řešení úkolu

1. Vybavíte si situaci, kdy jste se při řešení úkolu nejdříve zasekli, ale následně se situace vyjasnila? (Nadále bude uvedeno jako problém.)
2. Mohu začít tím, že Vás požádám, abyste mi řekli, jak rozumíte problému?
3. Představte si, že byste vysvětloval problém někomu, kdo se právě snaží tento materiál naučit. Jak byste to provedli?
4. Povězte mi o Vašich myšlenkách, Vašich reakcích před, během a poté, co jste se zabývali problémem.
5. Můžete mi říct, co Vám pomohlo pochopit problém?
6. Můžete popsat, jak jste vnímali (prožívali) problém během doby, co jste se zasekli a jak jste totéž vnímali/prožívali později?
7. Na základě svých zkušeností, jakou radu byste dali ostatním studentům, kteří by se mohli potýkat s problémem?
8. Řekněte mi, prosím, jaké další věci potřebujete k tomu, abyste dobře porozuměli problému?
9. Můžete mi říci, jak pochopení problému ovlivnilo vaše chápání ostatních věcí?
10. Bylo vaše pochopení problému něčím, co jste museli neustále zkoumat nebo jste se to jednou naučili a už to bylo jasné?
11. Popište, jak a v jakém kontextu jste problém použili od doby, co jste se toto téma naučili.
12. Je ještě něco dalšího, co byste chtěli sdělit ohledně tématu problému?
13. Můžete popsat, jakékoliv další situace, kde jste se nejprve zasekli, ale pak se situace vyjasnila a pokračovali jste dále?

D Agregovaná konceptuální mapa



Obrázek D.1: Agregovaná konceptuální mapa

Zdroj: autor

E Myšlenková mapa

Obrázek E.1: Myšlenková mapa pro objektově orientované programování a návrh

Zdroj: autor

F Ukázka zadání úkolu Hodiny a displeje

Softwarová firma dostala za úkol naprogramovat hodinový stroj, ke kterému by bylo možné připojovat různé displeje.

Vývojáři připravili třídu *Clock* a v ní nadefinovali metodu *TimeChanged*, která je volána externím modulem vždy při aktualizaci času. K objektu třídy *Clock* se mohout připojit displeje a na takto připojených displejích se zobrazují údaje podle typu příslušného displeje (např. jen hodiny a minuty, čas i datum, přesný čas včetně milisekund, apod.). Displeje se mohou kdykoliv připojit nebo odpojit, může být připojeno i více displejů současně (nebo žádný).

Vy, jako člen vývojového týmu, máte za úkol vytvořit objektový návrh pro popsanou situaci. Metodu *TimeChanged* můžete implementovat dle svého uvážení. Důležité je, že při každém volání této metody musí dojít k aktualizaci zobrazované informace na všech připojených displejích. Displeje se připojují samy, případně se samy odpojují (iniciátorem je vždy displej, nikoliv hodiny).

Aktuální čas a datum můžete nastavit v metodě *SetNewTime* (implementujte ve třídě *Clock*), kterou zavoláte z metody *Main* třídy *Program*. Ta může být implementována např. tak, že získá systémový čas počítače a zavolá metodu *TimeChanged*.

Dodržujte principy OOD a OOP. Implementujte pomocí výstupu na textovou konzoli.

Tipy pro typy displejů (zobrazované údaje):

1. hodiny, minuty, sekundy
2. hodiny, minuty, sekundy, milisekundy
3. datum, hodiny, minuty, sekundy

G Ukázka zadání úkolu Auto konfigurátor

Prodejce aut plánuje vytvořit software, který by měl obsahovat též auto konfigurátor. V tomto konfigurátoru si zákazníci mohou vybírat z nabízených typů karoserií automobilů a zvolit z dostupných typů motorů.

Navrhněte objektový model konfigurátoru výše popsané situace.

Pro demonstraci fungujícího návrhu **vyberte 2 příklady** konfigurací z dostupných typů.

Dodržujte principy OOD a OOP. Implementujte pomocí výstupu na textovou konzoli.

Ukázka výstupu 1:

```
karoserie sedan (4 dveře), motor vznětový (diesel)
```

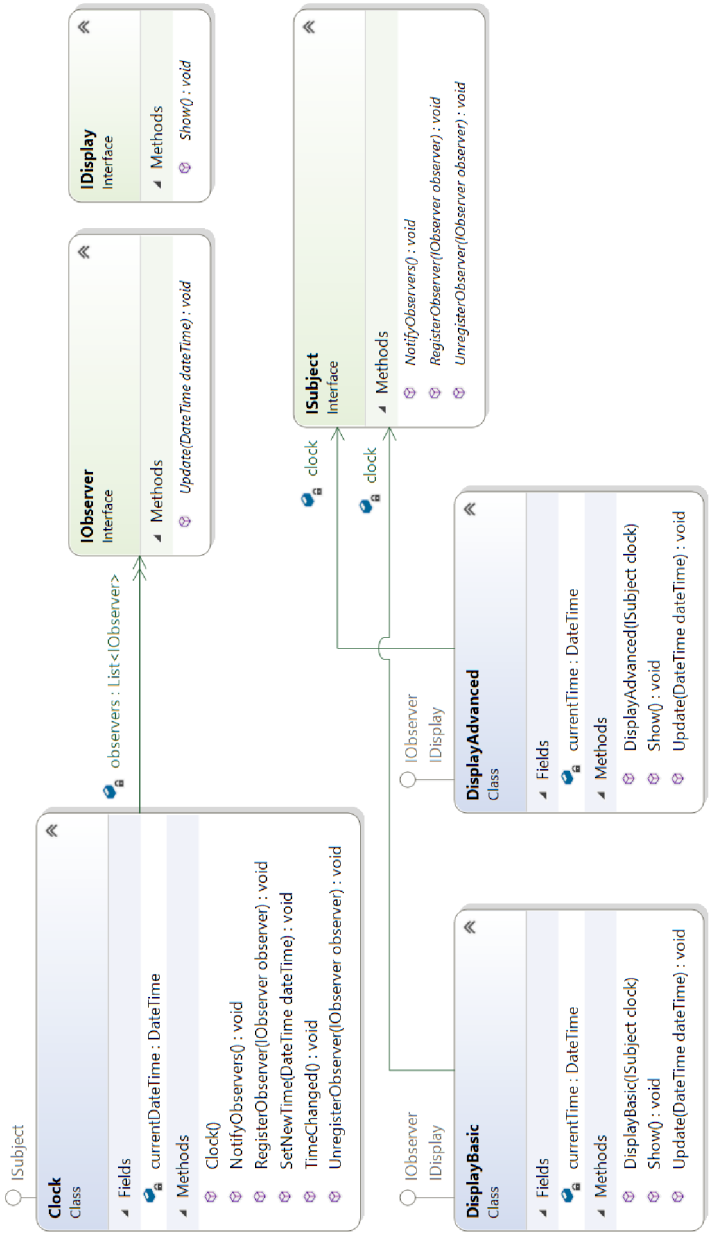
Ukázka výstupu 2:

```
karoserie kabriolet (2 dveře), motor zážehový (benzín)
```

Dostupné typy karoserií: *sedan*, *kabriolet*, *kombi*. Sledované atributy: *název*, *počet míst k sezení*, *počet dveří*.

Dostupné typy motorů: *spalovací motor* (*zážehový motor*, *vznětový motor*), *elektromotor*. Sledované atributy: *název*, *typ paliva*, *výkon v kW*.

H Ukázka řešení úkolu Hodiny a displeje



Obrázek H.1: Ukázka řešení úkolu Hodiny a displeje (diagram tříd)

Zdroj: autor

Výpis H.1: Ukázka řešení úkolu Hodiny a displeje (zdrojový kód)

```
class Program
{
    static void Main(string[] args)
    {
        Clock clock = new Clock();
        IDisplay displayBasic = new DisplayBasic(clock);
        Console.WriteLine();
        clock.SetNewTime(DateTime.Now);
        IDisplay displayAdvanced = new DisplayAdvanced(clock);
        Console.WriteLine();
        clock.SetNewTime(DateTime.Now.AddMinutes(1));
        Console.WriteLine();
        clock.SetNewTime(DateTime.Now.AddHours(1));
    }
}

public interface IObserver
{
    void Update(DateTime dateTime);
}

public interface ISubject
{
    void RegisterObserver(IObserver observer);
    void UnregisterObserver(IObserver observer);
    void NotifyObservers();
}

public interface IDisplay
{
    void Show();
}

public class Clock : ISubject
{
    private DateTime currentDateTime;
    private List<IObserver> observers;

    public Clock()
    {
        observers = new List<IObserver>();
    }

```

```
44      public void RegisterObserver(IObserver observer)
45      {
46          observers.Add(observer);
47      }
48
49      public void UnregisterObserver(IObserver observer)
50      {
51          int indexObserver = observers.IndexOf(observer);
52          if (indexObserver >= 0)
53          {
54              observers.Remove(observer);
55          }
56      }
57
58      public void NotifyObservers()
59      {
60          foreach (var observer in observers)
61          {
62              observer.Update(currentDateTime);
63          }
64      }
65
66      public void TimeChanged()
67      {
68          NotifyObservers();
69      }
70
71      public void SetNewTime(DateTime dateTime)
72      {
73          currentDateTime = dateTime;
74          TimeChanged();
75      }
76  }
77
78  public class DisplayBasic : IObserver, IDisplay
79  {
80      private DateTime currentTime;
81      private ISubject clock;
82
83      public DisplayBasic(ISubject clock)
84      {
85          this.clock = clock;
86          clock.RegisterObserver(this);
87      }
```

```
88
89      public void Update(DateTime dateTime)
90      {
91          this.currentTime = dateTime;
92          Show();
93      }
94
95      public void Show()
96      {
97          Console.WriteLine("time {0}", currentTime.ToShortTimeString());
98      }
99  }
100
101 public class DisplayAdvanced : IObserver, IDisplay
102 {
103     private ISubject clock;
104     private DateTime currentTime;
105
106     public DisplayAdvanced(ISubject clock)
107     {
108         this.clock = clock;
109         clock.RegisterObserver(this);
110     }
111
112     public void Update(DateTime dateTime)
113     {
114         this.currentTime = dateTime;
115         Show();
116     }
117
118     public void Show()
119     {
120         Console.WriteLine("time {0}:{1}:{2}.{3}",
121             currentTime.Hour, currentTime.Minute,
122             currentTime.Second, currentTime.Millisecond);
123     }
124 }
```

Zdroj: autor

I Ukázka řešení úkolu Auto konfigurátor

Výpis I.1: Ukázka řešení úkolu Auto konfigurátor (zdrojový kód)

```
class Program
{
    static void Main(string[] args)
    {
        Engine motorPetrol = new EnginePetrol();
        Engine motorDiesel = new EngineDiesel();

        Car carCabrio = new Cabriolet(motorPetrol);
        Car carCombi = new Combi(motorDiesel);

        Console.WriteLine(carCabrio.Drive());
        Console.WriteLine(carCombi.Drive());
    }
}

public class Car
{
    protected Engine motor;
    protected string name;
    protected int seats;
    protected int doors;

    public Car()
    {
        this.motor = null;
        seats = 4;
        doors = 4;
    }

    public Car(Engine motor)
    {
        this.motor = motor;
        seats = 4;
        doors = 4;
    }

```

```
37      public Engine Motor { get => motor; }
38      public string Name { get => name; set => name = value; }
39      public int Seats { get => seats; set => seats = value; }
40      public int Doors { get => doors; set => doors = value; }
41
42      public string Drive()
43      {
44          return string.Format(
45              "Car {0} ({2} doors) with engine {1} goes vroom vroom.",
46              this.Name, Motor.Name, this.Doors);
47      }
48  }
49
50  public class Sedan : Car
51  {
52      public Sedan(Engine engine) : base (engine)
53      {
54          name = "sedan";
55      }
56
57      public Sedan() : base()
58      {
59          name = "sedan";
60      }
61  }
62
63  public class Cabriolet : Car
64  {
65      public Cabriolet(Engine engine) : base(engine)
66      {
67          name = "cabriolet";
68          seats = 2;
69          doors = 2;
70      }
71
72      public Cabriolet() : base()
73      {
74          name = "cabriolet";
75          seats = 2;
76          doors = 2;
77      }
78  }
79
80  public class Combi : Car
```

```
{
    public Combi(Engine engine) : base(engine)
    {
        name = "combi";
    }

    public Combi()
    {
        name = "combi";
        seats = 5;
        doors = 5;
    }
}

public class Engine
{
    protected string name;
    protected FuelType fuel;

    public Engine()
    {
        fuel = FuelType.None;
        name = "no name";
    }


    public string Name { get => name; set => name = value; }
    public FuelType Fuel { get => fuel; }
}

public class EnginePetrol : Engine
{
    public EnginePetrol()
    {
        name = "petrol";
        fuel = FuelType.Petrol;
    }
}

public class EngineDiesel : Engine
{
    public EngineDiesel()
    {
        name = "diesel";
```

```
125         fuel = FuelType.Diesel;
126     }
127 }
128 
129 public class EngineElectric : Engine
130 {
131     public EngineElectric()
132     {
133         name = "electric";
134         fuel = FuelType.Electric;
135     }
136 }
137 
138 public enum FuelType
139 {
140     None,
141     Petrol,
142     Diesel,
143     Electric
144 }
```

Zdroj: autor

Obrázek I.1: Ukázka řešení úkolu Auto konfigurátor (diagram tříd)

Zdroj: autor

J Ukázka přepisu mluvení nahlas (úkol Postavy)

J, M – probandi

M: Nejdřív bych se dohodnul máme, tři hrdiny

J: Máme definitivně dané zadání. Jedná se o objektový návrh, potřebujeme nějaké objekty.

J: Co máme za objekty? Aspoň ze začátku…

M: Uděláme postavičku a zbraň.

J: Jo! Minimálně

M: Já to rovnou budu dělat, budeme psát v češtině?

J: Ano, můžeš

M: Myslíš, ne jako konkrétně?

J: Zatím dejme tomu nějakou postavu…

M: Jasan

J: Minimálně potřebujeme dva objekty, protože řešíme zbraň

M: A postavu… To takhle překopíruji rovnou

M: Teďka bychom mohli udělat nějaký jednotlivý

J: Teď je otázka … potřebujeme nějakou specializaci jednotlivých postav a zbraní

J: Takže nějaký dědění … u zbraní určitě

M: U postav ne

J: V jediným případě by se to mohlo hodit, že bychom měli přednastavený nějaký postavy s určitýma vlastnostma

M: Ale potřebuješ zbraň, protože každá zbraň má svůj způsob boje

J: Každá zbraň má své způsoby boje

M: Takže udělám další objekty… potomky

J: Který budou dědit

M: Dohodneme se na třech zbraní

J: Nutně by asi dědit nemusely, ale ... jo, celkově by mohly mít nějakou vlastnost, třeba jméno

J: Stačí asi dvě

M: Dáme tři

J: Dědičnost, né asociace

M: Tak

J: Možná

M: Ták, dobrý, nějaká zbraň

J: Standardně mě napadá meč

M: Luk

J: No, nevím

M: Meč, luk, sekera?

J: Joo

M: Tak, to jsou teda potomci zbraně jako takový

J: Takže co má?

M: Momentálně nic, máme zaseklý VisualStudio

J: Ahhh

J: Co máme umět, máme umět ... Zabojovat se zbraní

M: Takže to budou dědit

J: Zbraň bude mít vlastnost ... metodu

M: Útok

J: Teoreticky by jich mohlo být víc, kdybychom měli sekání sekerou ručně nebo házení sekerou

M: Hmm

J: Zatím bych to neřešil

M: Nebudeme to komplikovat ... Je to stejný princip

J: Necháme metodu pro útok ... Tady bude hlavní oni budou dědit a každá bude mít

M: Specifický útok

J: Ještě to v návrhu nemáme, postava musí mít zbraň u sebe nějak danou ... Přiřazenou

M: Kdybych takhle udělal, může být typ zbraně

J: Typ zbraně

M: Ta bude datového typu zbraň

J: Bude datový typ zbraň, protože vlastní nějakou zbraň a bude součástí postavy

M: Můžeme přiřadit jednotlivé... Ale to až v implementaci

J: To až v implementaci

M: Dobrý

J: Základní návrh asi

M: Ten je asi hotový

M: Budeme řešit ještě postavy

J: To až v implementaci

J: Ulož kdyžtak ten návrh ať to nedělá blbosti

M: Máme to uložený

J: Smaž Hello World a vytvoř dvě postavy to budeme potřebovat

M: Rytíř třeba

J: Může být rytíř

M: Musí mít defaultně nastavenou nějakou zbraň?

J: Nemusí, to pak přiřadíme ... Nejdřív potřebuji dvě postavy

M: Dáme třeba lučešník

J: Jak to máš ty, nevím, ale teď potřebuji dvě postavy

M: Právě abychom se sešli v nápadu

J: Já mám dvě postavy a zbraně až potom

J: A dvě zbraně

M: Stačí dvě, můžou si je měnit, to je jedno

J: Takže můžeš vytvořit dvě zbraně

M: Takže meč

J: A luk, ten má lepší zvuk

M: Tak takhle

M: Takže teďka jsme si vytvořili dva bojovníky a dvě zbraně

J: Ještě musíme ke každé postavě přiřadit zbraň, musí zaútočit

M: Jojo

J: To taky můžeme napsat, implementace až potom

J: Dá se to přiřadit

J: Musíš útočit u postavy... Rytíř. TypZbraně nebo co to je

J: Tak nic, teďka budeme řešit útok

J: Teoreticky obecnej útok asi implementovat nemusíme.

M: Ten nemusíme.

J: Protože nikdy obecná zbraň nebude. Leda, že by to byly jako pěsti nebo něco a to by spíš bylo zase lepší jako samostatná zbraň.

M: Ne, budeme to dávat jednotlivým zbraním jenom.

J: Takže ne, takže každou vlastní metodu útoku pro jednotlivý zbraně.

M: Takže normálně vlastně ke každý zbrani napíšu „sekám“, dejme tomu.

J: Pro jednoduchost bude metoda znova útok. Každá zbraň musí mít svojí vlastní metodu útok. Takže tady musí být metoda „public void utok“, ještě to bude muset myslím „override“ nebo „new“, já si teď nejsem jistej, co z toho, ale to je jedno.

M: No, ono nám to napíše, co tomu schází.

J: Visual Studio poradí.

M: Útok. Klíčová slova „new“.

J: Já si nejsem jistej, jestli máme nějakou chybu. Jestli nemusí bejt virtuální ta funkce útok nebo něco.

M: Ta funkce?

J: Ta horní.

M: Tadyhle hmm...

J: Nebo něco takovýho.

M: Nebo abstraktní. Zkusíme to takhle.

J: Abstraktní by bylo asi nejlepší.

M: Asi viď.

J: A zruš implementaci.

M: Jo. Takhle jo?

J: I ty hranatý, teda složený závorky, protože ty nejsou potřeba.

M: Takhlenc.

J: To bude asi takhle nejlepší a myslím že středník se dává za…

M: Jo jo.

J: Ještě si na něco stěžuje.

M: Ukaž co to píše. Metoda abstraktní…

J: Nevracíš „void“ ještě. Je tam nějaká chyba musíš ještě… Myslim, že se to píše obráceně, ale… Ale tím si jistej nejsem.

M: Tak to zjistíme.

J: To musíme zkusit.

M: Taky ne. Ukaž. Zkusíme před…

J: Ne, to psalo něco jinýho.

M: Tak počkej, já to vrátím zpátky. Tak.

J: A stěžuje si na něco jinýho.

M: Abstraktní …

J: Třída zbraň by mohla bejt celá sama o sobě abstraktní.

M: Jo už to vidim. Tak. Zmizíme … Zbraň, ne postava.

J: Zbraň nikoliv postava ano.

M: Tak.

J: Protože instance samostatně nikdy nebude.

M: Tak dobrý. A tady vlastně potom už bude jenom …

J: Moment, stěžuje si to na něco. To bude že nemá implementaci.

M: Nemá implementaci jo.

J: Ano správně to chceme.

M: Počkej, já bych teoreticky – co udělá žárovka, udělá to za mě? Ne. Vygeneruje...

J: Vygeneruje ti to funkci, ale ... Ok pořád jsme na tom že potřebujem implementaci útoku pro každou zbraň. Nebo minimálně v tomto případě zvuk zbraně. Ale..

M: Jo takhle výborně.

J: Takhle ...

M: Tak já to naimplementuju u všech.

J: Doplnilo „override“.

M: „Override“, tak výborně, to samý u poslední zbraně. Tak. Tím máme vyřešenou tuhle problematiku a už...

J: Každá ze zbraní ... Pro jednoduchost bych jenom vložil nějakej vlastní zvuk. „sek“.

M: Nějakej zvuk jenom ...

J: Meč může mít například „sek“.

M: To je meč ...

J: Luk může mít například ... To nevím ale ...

M: Luk bude mít ...

J: Výstřel. Výstřel. Control + Z je ...

M: Střílim. Střílim.

J: Například to je poněkud jedno.

M: Tak a poslední pro sekeru a tam bude ...

J: Větší sek?

M: Prosim.

J: Možná spíš sekera „sek“ a meč „šermování“ nebo něco. To je v tuhle chvíli jedno, jenom rozdílný hodnoty, mohlo by tam být cokoliv, ale...

M: „Buch buch“. Takhle nám to stačí.

J: Tak, teď by jsme měli mít základně naimplementovaný. Chybí nám ještě při-

řazení zbraně postavě. Asi…

M: Jo.

J: A samotnej útok se zbraní.

M: A potom nějaká následná výměna, kterou budeme simulovat.

J: Následná výměna by mohla bejt, ale myslim si že pro základní návrh není úplně nutná. Vlastně, vyměnit se dá i potom, ale…

M: Tak …

J: Můžeš přiřadit tady samostatně kdyžtak, ono se to dá dělat i jinak, ale museli by jsme prohodit pořadí. To mě první napadlo. Dá se do složenejch závorek, za to definovat přímo nějaká hodnota, ale uděláme to —

M: Zkusíme to obouma metodama. Hmm.

J: Ne ne. V tomto případě, kdyby jsme to chtěli definovat takhle, by muselo bejt obracený pořadí, zbraně by museli být nadefinovaný před postavama, pokud nechceš vytvářet přímo novej objekt v postavě.

M: Hmm.

J: A nebo ji tam ručně přiřadit každej postavě, záleží, co chceme.

M: Asi růčo.

J: Tak růčo. Tak tady. Rytíř "tečka" „typ zbraně" nebo jak se u nás… Ale je definovaná jenom jako privátní, nemáme veřejnou, takže přiřadit nemůžeme … vlastnost.

M: Tak to uděláme tím prvním způsobem, že si nejdřív…

J: To je v tomhle případě jedno, protože tam se nic nepřiřadí. Ty máš typ zbraně jako privátní, nemůžeš do ní mimo vlastní třídu přiřadit, takže buď ji změn na veřejnou a nebo udělej druhou veřejnou, která okolo ní se opisuje. Tady budeš muset naimplementovat stejně get a set. A nebo on se možná sám nadefinuje implicitně, to uvidíme. Pokus se přiřadit zbraň, kdyžtak se vrátíme.

M: Takže zkusíme to na rytíři, výborně už to funguje. „typ zbraně" …

J: Funguje. „Rovná se", jedná se o proměnou.

M: Tak a dáme meč. Tak a máme definováno.

J: Takhle má rytíř přiřazenou zbraň.

M: Potom tam máme druhou postavu a to je ...

J: To je nahoře v implementaci.

M: To bylo tady, lučištník ...

J: Tak lučištník.

M: A tomu přiřadíme luk. A jsme vlastně hotový.

J: Ještě musí se zbraní zaútočit. Teď ji mají jenom přiřazenou takže ...

M: Zaútočit jasně.

J: Bude rytíř „tečka“ „typ zbraně“

M a J: „Tečka“ „útok“.

J: Co. Jedná se o metodu, takže závorky.

M: Takže takhle.

J: Teď ještě lučištník.

M: Já to jenom rychle otestuju a spustim.

J: Dobrá...

M: Výborně, funguje nám to.

J: V pořádku, v pořádku.

M: Tak ještě pro druhou postavu, Lučištník.

J: Potom by ještě bylo dobrý možná vyzkoušet změnit zbraň. Ale to ...

M: Vyzkoušíme... Vyzkoušíme ...

J: Ale to až za chvíli.

M: Zaútočíme ještě s lukem.

J: Jenom myšlenka ...

M: Zkontrolujeme ...

J: Střílim, střílim. A teď třeba lučištníkovi přiřadit meč.

M: Všechno nám funguje. A teďka prohodíme ...

J: Takže znova lučištník „typ zbraně“.

M: Lučištník typ zbraně zaútočíme ...

J: A nejdřív meč, přiřadit meč. „Typ zbraně“ „rovná se“ meč. To jsi mohl nechat.

M: A zaútočíme za lučištníka.

J: Úplně si nejsem jistej, jestli je takhle volat … Ale asi jo. Asi jo, nic jinýho mě nenapadá.

M: Funguje to. Ještě by jsme mohli jako zdůraznit, kdo s tím zachází s tou zbrani.

J: To by se dalo, ale nebylo v zadání.

M: Jde to poznat vlastně tady v „main“. Ale …

J: To už je vylepšení. Základní návrh asi teda…

M: Základní návrh je hotov.

J: Asi.

K Ukázka přepisu mluvení nahlas (úkol Kavárna)

Tak. Vytvořím si teda nový projekt budu modelovat to samý jako coffeshop, takže si to nazvu coffe shop chci konzolovou aplikaci.

Takže coffeshop vytvořím. Vytvořím si class diagram. Tak budu potřebovat teda nějakou třídu, kterou si přidám do program.cs a název třídy bude beverage. Nebo já to budu psát anglicky, i když v tom, v tom je lepší, když to bude česky. Tak dám nápoj, budu to psát česky.

Pak budu potřebovat nějaký přísady, tak vytvořím si mléko přidám ho do programu.cs vytvořím si ještě jednu přísadu. Karamel třeba přidám ho do programu. Cs pak tam byla třída nějaké espreso, taky ho přidám do programu.cs pak tam bylo nějaký dávkovač. Jak to nazvu něco jako nápoj machine, to když tak pak přejmenuju. Tak nápoj bude abstraktní třída, tak to tady změním na abstrakt. Pak espreso bude děti třídy a tady z nápoje. Na nápoj nápoje. Mléko nápoj bude dědičnost a karamel bude dědit tak jí tam přidám. Nápoj přísady to pojmenuju.

Teďka přejmenovávám, tak teď mi to tady hází chybu, protože jsem to tady nepřejmenoval. Takovej detail. Tak v nápoji bude. Přidej nebo ne. Získej cenu a vlastnost tady dám cena.

Jsem tam přidal pole, ale chtěl jsem tam přidat vlastnost. Takže já tam přidám vlastnost cena typu int. Přidám jí get, set. Tak přesunu nápoj přísady asi výše. No jako zkusím. No já si myslím, že to mám špatně. No něco z nápoj přísady se dědilo do přísady. Nemyslím si že by to byla celá třída.

Tak to bych měl asi základní návrh předpokládám.

Teďka. Teďka už nevím jak dál. Vím, že tam byly nějaký.

Nevím.

V tenhle moment bych se kouknul, jak jsme to dělali, protože nevím jak dál.

Vím, že tam byly konstruktory tak tam můžu přidat konstruktory. Tak můžu

teda espreso udělat ještě konstruktor espreso tady bude zase jenom cena rovná se 30. Pak můžu si v kódu vytvořit public karamel. Cena deset, mléko cena třeba patnáct. Tady dám dvacet to je jedno.

Teďka vlastně to získej cenu to bude abstraktní a nechám jí public abstrakt void. Jo nemám. Jo hlásilo mi to chybu, když to je abstraktní metoda tak to potřebuje to nechce tělo, takže jsem ho vymazal. Teď vymažu konstruktor nápoj, protože si myslím, že ho nebudu potřebovat momentálně. Teďka mi to hlásí chybu, že nemám přidanou abstraktní třidu (asi metoda) přidaná. Tak jí tam přidám do třídy nápoj přísady a do espresa.

Tam mám na začátku nějakou blbost, jestli si dobře vzpomínám tak tady nebyla potomek nápoje tam byla nějaká vlastnost na to odkazovala nebo něco takového, ale teď na to asi nepřijdu.

Metoda získej cenu by mohla vracet int nebo by měla vracet int. To, jestli si vzpomínám tak u těch potomků se odkazoval na nápoj přísady. V konstruktoru byl nějakej nápoj přísady nápoj. A odkazoval se do base na něco. Ale opět si nevzpomenu, jak to tam bylo. Si ani nevzpomenu, jak vypadala ta syntaxe v podstatě.

Si úplně nevzpomínám. No můžu si teďka vytvořit nějaký testovací objekt. Espreso espreso. Tak si vytvořím nové mléko. Jo do toho espresa se vytvoří. Třídy espreso. Tady musím mít nějakou chybu, že buď v konstruktoru mléko má být espreso espreso. Nevím, jak to tady bylo. Já to zakomentuju. No můžu tady do tý ovveride získej cenu. No ono to je jedno. Vrátit cenu. Když budu chtít jenom karamel tak to nechce počkat proč to nechce. Získej cenu. Jo já to mám tady.

Já tu mluvím o nějakém mléku, ale vůbec ho tu nemám. Jo já ho nemám, jak se mi vymazal nebo jak jsem měnil. Jak jsem měnil. Teď nevím, jak to udělat, aby se mi zobrazovala v class diagramu. Tak já jí udělám znova v class diagramu. Já jsem jí jenom možná posunul. Mléko vytvořím do ní konstruktor bude dědit z nápoj a vytvořím tam konstruktor mléko a teď asi. A abych jí tam měl nějak tak si jí posunu výš pod karamel třeba. Tady tu espreso si posunu výš nad karamel.

Teď jsem přidal. Teď mi to hlásí chybu u mainu. Potřebuju tady. Mám tady prostě chybu. Tak se kouknu, co mi to tady nabízí. Potřebuju do konstruktoru nápoj přísady nápoj argument. Tak to chce nějaký argument, ale já žádný nemám, tak si ho vytvořím tady. Ale tak jsme to nedělali což je podle mě blbě tak já to z toho espresa vymažu.

Tak mám tady nějaký návrh, ale tam to bylo nějak jinak s tou dědičností s tím nápoj přísady s tím beverage jinak se to jmenovalo. A já nevím, jak to tam bylo. Tam bylo podle mě nějaká vlastnost nápoj přísady vázána na nápoj a nebyla to dědičnost celý třídy a teď si nejsem jistej jak to tam bylo. Nicméně teď nevím, jak pokračovat. Mám tady mlíko a tady můžu přidat cenu do konstruktoru. Získej cenu metoda mi podle mě nefunguje. A nevím, jak tam byla ta syntaxe, jak jsme to dělali tak mi to dávalo smysl, ale nemůžu si vzpomenout.

Milk...? A teďka..to celou dobu dělám česky, ale to je jedno. No já potřebuju. Já si vytvořím espreso. To se bude rovnat new espreso mléko. A to se nedá udělat, protože to spolu nekomunikuje je tu nějaká chyba nabízí mi to tady. Mléko statickou vlastnost ne statický pole a pak statickou vlastnost nebo jak se to jmenuje možná člen. Pak místo espreso. To je taky blbost. Nevím nevim nevim.

Espreso ta má přístup do získej cenu a cenu. Tak kolik mi řekne cenu. ...

Ne tak mi to neřekne nic prostě. To je divný teda, že nefunguje ani ta cena 30. To je fakt divný. Jo já jí nevypisuju. To já jsem blbej. Já ji nevypisuju do konzole. Jo 30. Ale jak získat, jak jsme to dělali. ... Nevim.

Já toho nechám, protože nevím jak dál. Vím, že by tam někde měla být dědičnost u toho konstruktoru. Nevím teda jestli se tomu říká dědičnost u toho mléka a karamelu. Jako z toho toho. Pak by tam měl být argument u těch přísad, který odkazuje na nápoj přísady. Ale po několikátý si myslím, že ten nápoj přísady s tím nápoj mají být jinak provázaný a na to nepřijdu, protože si to nepamatuju a u toho asi zakončím, protože si to nepamatuju a nevím jak dál.

Pokračování po pauze:

Tak minule jsem, jsem skončil u projektu CoffeShop. Minule jsem skončil,

protože jsem nevěděl, jak pokračovat. Teď jsem si nastudoval ten hotovej implementovanej projekt. A jdu pokračovat. Tak a hlásí mi to tady chyby, že když do espressa jsem minul a do vytvořené espressa která má třídu espresso tak chci přidat nový expresu a hází mi to chybu A to proto že mléko tečka Mám tady na implementovaný jako tečka Mám tam konstruktor v mléku a mám tam špatný argumenty nebo argument Mám tam daný espresso, ale já vlastně potřebuji. nápoj přísady tak. teďka ještě koukám, když jsem v tom mléku, že to nic nemám vlastně Tím myslím, že to nemám nic. Žádné jiné metody, a ještě jsme přidávali nebo chtěli jsme mít, aby jméno, aby dokázal a a ještě tam chybí get kost nebo o získej cenu a to potřebuju Já koukám že zjistit cenu to má Mám ji v abstraktní metodě nápoj a. Teď mě napadá, jestli vy vlastně náboj přísady neměl být taky strašný, protože jsi vlastně ty. No kouknu se do Class diagramu. Tady mi chybí konstruktor. Tak tady přidám metodu nebo konstruktor NápojPřísady.

A... Do argumetnu přidám... ten nápoj. Nápoj nápoj. A ten nápoj budu k něčemu přiřazovat. Jo tady bych měl mít něco jako použité přísady. A takže bych měl mít vlastně. Vlastnost public nápoj použité přísady. Get, set a...

No zkusím se vrátit k tomu Karamelu a mléku, protože mi to tady hází nějakou chybu. Kouknu se teda co je to za chybu, ale podle mě... Jo v tom konstruktoru to si pamatuju. By měl, být to že dědí base nápoj. Jo base nápoj mi to tady radí. A to samé bych měl mít u konstruktoru u mléka.

Tak u karamelu je to v pohodě, ale u mléka to hází nějakou chybu. Nejsem si jistý, ale myslím, že jsme to měli všude stejný. Kouknu se jestli... V mainu mám u expressa přiřadit new mleko a v závorce expresso. Tak mi to jako nápovědu vyhazuje vytvořit konstruktor, ale to jsem si myslel že mám vytvořenej.

Jo aha já tady mám. No to si nechám opravit, protože... No teď mi to zase zmizelo. No nabízelo mi to, že... Jo aha ono mi to vygenerovalo a mám tu vlastně dva. Aha ono se to vygenerovalo. Tak původní smažu, co jsem tu měl původní. Dám tu jenom cenu přiřadím... v mléku přiřadím cenu na 20. A ještě jsem měl vlastně. To můžu z nějakýho jinýho konstruktoru name. Takže v nápoji vytvořím

public string. Vytvořím vlastnost public string Název a přiřadím sem. Tak pořád mi to u mlíka hází nějakou chybu. Ale už to mám, takže sem můžu dát nějaký ten název. Tak tady přiřadím název mléko. To můžu udělat vlastně s diakritikou. To samý můžu udělat u karamelu. Tak tady mi to už chybu nehází nikde. Akorát tady mám zase, tak tady to přepíšu na nápoj nápoj, protože si tam chci přiřadit nápoj...

Tak teďka ještě bych měl mít v abstraktní třídě ZískejNázev, takže public abstract string ZískejNázev. Teďka to bude chtít někde. Teď to bude chtít z těch tříd, aby to dědilo, takže to vygeneruju. A napíšu sem return název. Název jsem to pojmenoval. To samý to bude chtít v espressu. Jo počkat ono to v tom expresu bylo asi jinak. No nechám tu ten return název. Jo já jsem sem dal rovná se to je špatně. A ještě...no to je jedno. A ještě někde? Počkat proč to...jo...

Tak..... No, když je abstraktní tak se dá taky přepisovat. Jo tak ještě vytvořím v karamelu a mléku metodu, dvě metody, takže public ovverride chci, aby vracela string. Teď se mi vygenerovala ToString, ale to nechci. Jak jsem jí pojmenoval proč mi jí nevrací. Proč mi to tady nenabídlo. Jo aha nabídlo ale asi jsem byl rychlejší. Dobrý tak mi jí tady nabízí. Takže ZískejNázev.

Jo a já chci vlastně vrátit vlastnost použité přísady to má jo to má ZískejNázev +. Jo to mi získá to co už tam je. Tady dám čárku. Plus název. A ještě teda druhou to bude v podobným duchu. Public ovveride int ZískejCenu a to bude taky použité přísady získej cenu. Plus ještě chci cenu tady toho cena a z nějakýho důvodu to háže chybu. Počkat takže, že bych. Tady chci ještě.

No to si pamatuju to tam bylo to tam bylo stoprocentně. Bylo to u všech.

Takže teďka mi to háže nějakou chybu, takže se jdu kouknout na abstraktní metodu. Tu tu mám... No tu to mám. Tak se mi to hází...

To je zajmavý, že mi to tady háže. Jo já jsem zkopíroval tu implementaci, co jsem měl předtím a mám tady vlastně v int string. To je zajímavý. Takže se ještě kouknu to, co mi to nabízelo jako opravu. Jo nenabízelo... Jo a proto mi to tady hází chybu a to samý bude tady.

Tak... Jo já tady vytvářím no já jsem to měl možná předtím...no to je jedno. A ještě mi tam chybělo to...

Já tady vlastně vytvářím espresso, ale já chci vlastně vytvořit nápoj. Proto už ten nápoj tam mám. Počkat, takže tohle smažu. Nápoj bude to třeba espresso třeba jedna se rovná new espresso. No jasně tak teď by to mohlo.

Jo tady k espressu přidám ještě název do konstruktoru. Espresso. Takže když přidám to mlíko a ještě přidám do espressa karamel. Tak by to mělo asi fungovat.

Ještě dám teda espresso1.ZískejNázev. Ještě jsem tam mohl dát tu mezeru. To dělám takhle blbě no už jsem to tak udělal. Takže...

Takže to spustím. Espresso, mléko, karamel 60. Jo 60.

Tak to bych měl mít. Měl bych to mít hotový...