Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačních technologií

Diplomová práce

Programovací jazyk Swift

Bc. Marcel Soukeník

Vedoucí práce: Ing. Jiří Vaněk, PhD.

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ DIPLOMOVÉ PRÁCE

Bc. Marcel Soukeník

Informatika

Název práce

Programovací jazyk Swift

Název anglicky

The Swfit Programming Language

Cíle práce

Primárním cílem práce je vymezení funkcionality programovacího jazyku Swift, aplikovat šíři jeho implementačního využití a kritické zhodnocení gramatiky a syntaxe daného jazyka. Další cíle této práce budou:
- představit základní vlastnosti programovacího jazyku Swift
- podrobně prozkoumat architekturu jazyka přes zdrojový kód
- vyvinout funkční části kódu v jazyce Swift vymezující nové a kritické vlastnosti jazyka
- implementovat obecné návrhové vzory v jazyce Swift
-porovnat procesní výkonnost rychlost s ostatními jazyky
- formulovat závěry a doporučení

Metodika

Teoretická část představí základy a myšlenky, jež stojí při budování programovacího jazyku Swift. Budou prozkoumány základní prvky a vlastnosti jazyka Swift, jejich syntaktická i gramatická formulace a stručné porovnání s dalšími programovacími jazyky. Dále se důkladně zanalyzuje architektura zdrojového kódu.

Praktická část se zaměří na implementaci kritických vlastností programovacího jazyka Swift, jeho důležitých návrhových vzorů, a dále nových řešení vyplývající z teoretických východisek. Poté budou jednotlivé části implementace zhodnoceny. Další částí bude porovnání výkonosti jazyka Swift s vybranými programovacími jazyky. Závěrečnou částí práce bude formulovat výsledky a vyplývající doporučení.

Doporučený rozsah práce

50 – 60 stran

Klíčová slova

Swift; Apple; Mac; Programovací jazyky; Informační technologie; Open Source

---

Doporučené zdroje informací

APPLE, The Swift Programming Language; Apple 2014; [online]. [cit. 22-04-2016].; <https://itunes.apple.com/cz/book/swift-programming-language/id881256329?mt=11>.

DeVOE Jiva; Objective C; Indianapolis: Wiley 2011; 400s.; ISBN: 9780470479223

FELLIER, Jesse, Swift™ for dummies, Hoboken: John Wiley and Sons, 2015; 365s.; ISBN: 1119022223

HARPER, Robert, Practical foundations for programming languages, Cambridge: Cambridge University Press, 2013, 471s., ISBN: 9781107029576

PECINOVSKÝ, Rudolf, Návrhové vzory, Brno: Computer Press, 2007; 527s.; ISBN: 8025115828

Předběžný termín obhajoby

2017/18 ZS – PEF (únor 2018)

Vedoucí práce

Ing. Jiří Vaněk, Ph.D.

Garantující pracoviště

Katedra informačních technologií

Elektronicky schváleno dne 1. 6. 2016

Ing. Jiří Vaněk, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 2. 8. 2016

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 24. 11. 2017

Čestné prohlášení

Prohlašuji, že svou diplomovou práci Programovací jazyk Swift jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu literatury na konci práce. Jako autor uvedené diplomové práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 28.11.2017

Poděkování

Rád bych vyjádřil poděkování svému vedoucímu práce Ing. Jiřímu Vaňkovi, PhD. za trpělivost, cenné rady a čas při konzultování zpracované práce. Také chci poděkovat mé manažerce Petře Slezákové za poskytnutí času a motivace při dokončování této práce.

Programovací jazyk Swift

The Swift Programming Language

Souhrn

Téma této diplomové práce je programovací jazyk Swift. Cílem práce je vymezení funkcionality programovacího jazyka Swift a poskytnout její kompletní přehled. Dále na praktických příkladech pak ukázat syntaxi a využití jazyka Swift. Práce je rozdělena do několika částí. Teoretická část zahrnuje přehled řešené problematiky. Praktická část analyzuje zdrojový kód a syntaxi, implementuje příklady pokrývající vlastnosti jazyka a porovnává syntaxi a výkon ostatních programovacích jazyků. Závěrečná část shrnuje a zhodnocuje veškeré získané výsledky a poznatky.

Summary

This thesis topic is about The Swift programming language. The main aim of the thesis is to provide defining functionality of the Swift language and complete research about this language. Also on practical examples demonstrate syntax and use of the Swift language. The thesis has several sections. The theoretical section is to provide all information about the topic. The practical section is to show: an analysis of source code and syntax, implementation covering features of the language and comparing syntax and performance of other languages. Final section summarizes all results and observations.

Klíčová slova
Swift; Apple; Mac; Programovací jazyky; Informační technologie; Open Source

Keywords
Swift; Apple; Mac; Programming languges; Computer Science; Open Source

Obsah

Seznam tabulek

Seznam obrázků

1 Úvod

Úlohou programovacích jazyků je interpretovat algoritmy, který dokáže zpracovat stroj. Každý programovací jazyk má 2 základní prvky: procedury a data. Existuje přes 200 různých programovacích jazyků, ale žádný nedokáže komplexně pokrýt všechna řešení pro daný úkol. Každý programovací jazyk má od počátku daný účel.

Programovací jazyk Swift byl představen firmou Apple v roce 2014 na vývojářské konferenci WWDC. Hlavní motto, jež na konferenci zaznělo a mělo by vystihnout jazyk Swift, je: „Objective-C bez zatížení jazyka C.“ Objective-C vznikl v roce 1986 jako objektová nadstavba jazyka C. V současné době, kdy převládá trend aplikací pro mobilní platformy, se Objective-C v porovnání s jazyky jako je Java, Python nebo C# jeví jako zastaralý. Firma Apple proto vyvinula nový programovací jazyk, čímž dala najevo snahu udržet si krok v oblasti inovací a rozšířit si vývojářskou obec.

Swift má ambice být moderním, bezpečným a jednoduchým jazykem, se kterým lze vytvořit jak jednoduchou aplikaci „Hello World“, tak celý operační systém. Licence jazyka Swift byla uvolněna 3. prosince 2015 jako Open Source, načež se otevřela možnost všem vývojářům přispět k vývoji jazyka Swift či použít Swift pro jinou platformu než Apple. Také byly uvolněny kompilátory pro Linux, tudíž není nutné vlastnit Mac pro využití tohoto programovacího jazyka. V červnu 2016 na vývojářské konferenci WWDC byla představena třetí verze jazyka Swift 3.0.

Tématem této práce je pokrýt a prozkoumat všechny významné vlastnosti programovacího jazyka Swift. Dále zhodnotit potenciál a ambice, kde se dá jazyk Swift využít. Velká pozornost bude věnována průzkumu a implementaci návrhových vzorů. Nebude chybět ani kritické porovnání s ostatními jazyky.

2 Cíle práce a metodika

2.1. Cíle práce

Primárním cílem práce je vymezit funkcionality programovacího jazyka Swift, aplikovat šíři jeho implementačního využití a kriticky zhodnotit gramatiku a syntax daného jazyka. Další cíle této práce jsou:

- představit základní vlastnosti programovacího jazyka Swift,
- podrobně prozkoumat architekturu jazyka přes zdrojový kód,
- vyvinout funkční části kódu v jazyce Swift vymezující nové a kritické vlastnosti jazyka,
- implementovat obecné návrhové vzory v jazyce Swift,
- porovnat procesní výkon a rychlost s ostatními jazyky,
- formulovat závěry a doporučení.

2.2. Metodika práce

Práce bude rozdělena do dvou částí. První část uvede přehled řešené problematiky. Nastíněna bude klasifikace programovacích jazyků, základní charakteristika programovacího jazyka Swift, jeho základní knihovna a stručný vývoj jazyka Swift. Dále bude následovat porovnání s ostatními rozšířenými programovacími jazyky a syntaxe jazyka Swift. Ke konci teoretické části budou vedeny přehledné informace o návrhových vzorech.

Druhá, praktická část se bude věnovat analýze zdrojového kódu jazyka Swift. Podrobně bude prozkoumána syntax, kompilace a výkon, nadto dojde k implementaci hlavních vlastností jazyka Swift a návrhových vzorů. Nakonec bude následovat porovnání s jazyky Objective-C a jazykem Python.

Výstup práce bude zahrnovat vymezení funkcionality a potenciál programovacího jazyka Swift. Rovněž bude nabídnuto kritické zhodnocení vyplývající z praktické části této práce. Závěrem dojde k formulaci doporučení při užívání jazyka Swift a nedostatky, které je nutné brát v potaz.

3 Přehled řešené problematiky

V této kapitole je zahrnuto základní rozdělení programovacích jazyků, obecný přehled a základní informace o programovacím jazyku Swift, porovnání programovacích jazyků jako C, C++, Java a Python. V dalších kapitolách se pak budeme zabývat strukturou jazyka Swift.

3.1. Klasifikace programovacích jazyků

V oboru počítačových věd neexistuje jeden všeobecně přijímaný a ucelený pohled na kategorizaci programovacích jazyků. Existují jen obecné definice a kategorie, jak lze programovací jazyk rozdělit. Ty se liší pouze hloubkou a rozsahem pohledu na jednotlivé kategorie.

3.1.1. Účel využití programovacích jazyků

Každý programovací jazyk má svůj specifický účel využití. Dává proto smysl rozlišit jazyky na různé druhy. Zde jsou některé z nich:

- Strojové jazyky – interpretované přímo v hardwaru.
- Jazyk symbolických adres (angl. Assembly languages) – nízkoúrovňový jazyk zapouzdřující strojový jazyk
- Vysokoúrovňové jazyky – jakýkoli strojově nezávislý jazyk
- Systémové jazyky – řídící úkoly počítače na nižší úrovni jako je např. správa paměti
- Skriptovací jazyky – obecně velmi účinný interpretovaný jazyk
- Vizuální jazyky – netextový jazyk

Rozdělení podle účelu není vzájemně exkluzivní, neboť jednotlivé druhy lze kombinovat. Např. jazyk C je považován za jazyk vyšší úrovně a současně i systémový jazyk. Existují ještě další druhy, jako např. ezoterické, vzdělávací, kompilované, interpretované, bezformátové, hybridní atd. [6]

3.1.2. Programovací paradigma

Programovací paradigma je styl programování počítače na základě matematické teorie nebo koherentním svazku zásad. Každé paradigma podporuje množinu konceptů, které nejlépe řeší daný druh problému. [2]

Programovací paradigma	Charakteristika
Imperativní	Řídící tok je přesně definovaná sekvence příkazů. Každý krok způsobuje změnu globálního stavu daného výpočtu.
Strukturované	Druh imperativního programování, kde řídící tok je ovládán vnořenými cykly, podmínkami a podprogramy než pomocí příkazu „goto“. Proměnné jsou zpravidla v blocích lokální.
Objektově-orientované	OOP je založené na posílání zpráv objektům. Objekty mohou mít deklarovaný stav a chování.
Deklarativní	Řídící tok v deklarativním programování je implicitní. Důležitý je požadovaný výsledek, nikoli jak získat výsledek.
Funkční	Řídící tok je vyjádřen kombinací volání funkcí než přiřazování hodnoty proměnným.
Logické	Programátor specifikuje fakta a pravidla, a výpočetní stroj vyvodí odpovědi na zadané otázky.

Tabulka č. 1 Hlavní programovací paradigmata a jejich stručná charakteristika [7]

Hlavní a nejběžnější programovací paradigmata jsou uvedena v tabulce č. 1. Existují i další i typy paradigmat. Žádné paradigma není výlučně exkluzivní, tj. programovací jazyk může být objektivně orientovaný i funkční. [7]

3.2. Charakteristika jazyka Swift

Swift je dle oficiální dokumentace navržen jako moderní programovací jazyk. Je postaven na nejlepších vlastnostech jazyka C a Objective-C bez omezení kompatibility s jazykem C. Swift tak zavádí bezpečné programovací vzory a přidává moderní funkce, jež činí programování lehčí, flexibilnější a zábavnější. [1]

Programovací jazyk Swift kombinuje několik programovacích paradigmat. Swift je jazyk, který je:

- objektově-orientovaný
- funkční
- imperativní
- generický
- řízený událostmi.

Swift patří mezi vyšší programovací jazyky a je tak problémově orientovaný. Také se řadí mezi bezpečné jazyky a znemožňují programátorovi udělat fatální chybu v systému.

Swift byl navržen firmou Apple a představen veřejnosti v roce 2014. Vývojáři jej využijí pro všechny Applem vlastněné platformy, jako je např. macOS, iOS, watchOS a tvOS. Vedoucím celého projektu Swift byl Chris Lattener, který vedl tým do začátku roku 2017. V současnosti je odpovědným vedoucím projektu Ted Kremenek. Apple začíná Swift pomalu integrovat v rámci svých operačních systémů, např. do aplikace Hudba či Dock. [22]

3.2.1. Návrh jazyka Swift

Jazyk Swift byl navržen, aby byl dostupný pro rostoucí škálou výpočetních zařízení, jako např. desktopy, mobilní zařízení či cloudové služby. Vývojář by tak měl umožněno snadnější psaní a udržování správných programů. Aby bylo možné tohoto stavu docílit, měly by být splněny tyto 3 základní předpoklady:

Bezpečnost. Při psaní kódu je hned patrné, že je třeba se chovat v určitých bezpečnostních mantinelech. Nedefinované chování je nepřítelem bezpečnosti a vývojář by měl odhalit chyby dříve, než je software uveden do produkce. Může se zdát, že Swift je příliš striktní, ale to z dlouhodobého hlediska může ušetřit spousty času při hledání těžko odhalitelných chyb.

Rychlost. Swift je považován za náhradu jazyků založených na C (C, C++, Objective-C). Těmto jazykům se Swift musí vyrovnat ve většině úkolů i výkonově. Výkon též musí být předvídatelný a konzistentní, a ne pouze rychlý jen v některých okamžicích, které vyžadují

následnou péči. Mnoho nových jazyků přidává spousty nových vlastností – rychlost však nikoli.

Srozumitelnost. Swift těží z výhod prudkého rozvoje výpočetní techniky a nabízí jednoduchou syntaxi, již je snadné použít se všemi moderními aspekty, které vývojář očekává. Vývoj Swiftu ovšem tady nekončí. Jazyk se bude dále upravovat a osvojovat si techniky, které fungují, čímž budou dělat Swift ještě lepším. [3]

3.2.2. Základní vlastnosti jazyka Swift

Swift zahrnuje možnosti, které usnadňují kód lépe číst a zapisovat, vývojáři ale dává i možnost mít plnou kontrolu nad opravdovým systémem programovacího jazyka. Odvozené typy umožňují udržet kód přehlednější a bez náchylnosti k chybám. Moduly není již nutné zahrnovat s hlavičkovými soubory. Paměť je spravována automaticky a není nutné příkazy zakončovat středníky. Swift si také vypůjčuje vlastnosti z ostatních jazyků, např. jména parametrů jsou ve Swiftu mají mnohem čistější syntaxi oproti Objective-C a dělají API čitelnější a udržitelnější.

Zde je přehled dalších vlastností, které Swift nabízí:

- Sjednocené *closures* s ukazateli na funkce
- N-tice a mnohonásobné „return“ hodnoty
- Generičnost
- Rychlé a stručné iterace přes kolekce
- Struktury, které podporují metody, *extentsions* (z angl. rozšíření) a protokoly
- Funkční programové vzory např. mapování a filtrace
- Vestavěné mocné ošetřování chyb
- Pokročilé řízení toku dat s klíčovými slovy jako: *do, guard, defer* a *repeat.* [3]

3.2.3. Verze jazyka Swift

První verze Swiftu 1.0 byla uvolněna 9. září 2014. Menší aktualizace se dočkal jazyk hned v říjnu 2014 na verzi Swift 1.1 a poté v dubnu 2015 na verzi Swift 1.2. Na WWDC 2015 byla představena verze Swift 2.0, která přináší větší změny. Obsahuje lepší ošetřování chyb pomocí try/catch, nová klíčová slova „guard“ a „defer“ a další drobné úpravy

v syntaxi. Přišla i velká licenční změna, kdy byl jazyk Swift 2.0 uvolněn jako Open Source. Současně byl vytvořen projekt na službě GitHub.

Swift 2.2, uvolněný 30. března 2016, přináší vylepšenou syntaxi a možnosti, ale zároveň vypuštění některé vlastnosti. Byla přidána kontrola verze Swiftu skrze značky kompilátoru, zabudované porovnání n-tic, odstranění C-stylové *for* cykly, spolu s ++ a --.

Swift 3.0 byl představen na konferenci WWDC 2016 a uveden 15. září 2016. Změny se týkají především API, jmenovací konvencí a syntaxe. Příklady některých změn: parametry volané funkce musí mít nápis ve výchozím stavu, zbavení se nadbytečných slov (např. *UIColor.blueColor -> UIColor.blue()* aj.), enumerace nyní začínají malým písmenem, podpora C funkcí apod.

Swift 4.0 byl představen na 5. června 2017 na konferenci WWDC 2017. Finální verze byla vydána 19. září 2017. Tato nová verze nepřináší žádné velké změny ani novinky, ale jen malé evoluční změny, jako např. úpravu *String* typu pro udržení Unicode standardu, vylepšení kolekcí *Dictionary* a *Set*, zlepšení archivace a serializace apod. Na konci listopadu 2017 je plánováno vydání nová verze Swift 4.1.

3.3. Základy jazyka Swift

Jazyk Swift přejímá základní datové typy z jazyka C, např. *Int*, *Double*, *Float*, *Bool*, *String,* a datové kolekce, např. *Array* a *Dictionary*, ale zároveň si razí vlastní cestu. Novinkou oproti Objective-C je nový typ n-tice, která dovoluje seskupovat hodnoty. Funkce a metody mohou nyní vracet více hodnot než jednu.

Tato kapitola představí základy jazyka Swift, které tento jazyk definují.

3.3.1. Základní datové typy

Netypové jazyky mají pouze jeden typ a to je „slovo“, které je uloženo v operační paměti. Typové jazyky rozlišují více typů „slov“ v operační paměti a lze určit co dané slovo reprezentuje. [5]

Swift je typově bezpečný jazyk a nedovolí programátorovi udělat chybu v nechtěném přetypování. V tabulce č. 2 jsou základní datové typy:

Datový typ	Definice
Int	*Integer* reprezentuje celá čísla. Tento typ je *signed*, tj. může dosahovat i negativních čísel. Podle číselné přípony 8, 16, 32 64, lze i zvolit bitovou formu např. *Int32* pro 32-bitové platformy. Jednoduchý zápis *Int* má takovou bitovou formu jako architektura daného systému.
UInt	Tento typ integer obsahuje nezáporná celá čísla. Předpona „U“ značí *unsigned*. Stejně jako u *Int* lze přidat bitovou formu.
Float	*Float* definuje čísla s plovoucí desetinou čárkou pro 32-bitové platformy. Přesnost výpočtu je na 6 desetinných čárek
Double	*Double* jsou čísla s plovoucí desetinou čárkou pro 64-bitové platformy. Přesnost výpočtu je na 15 desetinných čárek. Forma *Double* je preferována před *Float*.
Bool	Typ *boolean* rozlišuje logické hodnoty *true* a *false*.
String	*String* jsou textové řetězce. Značí se značí dvojitými uvozovkami např. „Hello, World“
Character	*Character* je jednotlivý znak textu. Značí se též dvojitými uvozovkami např. „C“
Optionals	*Optionals* je nový typ, který se v jazycích C a Objective-C nevyskytuje.
Tuples	*Tuples*, nebo n-tice dovoluje seskupovat více hodnot do jedné sloučené hodnoty

Tabulka č. 2 Seznam datových typů v jazyce Swift

3.3.2. Literály

Literál je součástí zdrojového kódu, jenž reprezentuje konkrétní hodnotu typu jako *string,* číslo nebo booleovskou hodnotu. Literál nemá svůj vlastní datový typ, namísto toho Swift literál analyzuje a odvodí nejlepší možný datový typ. Na obrázku č. 1 jsou ukázky příkladů některých typů literálů.

```
25              // Celočíselný literál
3.14159         // Číselný literál s desetinnou čárkou
"Ahoj"          // String literál
true            // Literál typu Boolean
```

[zdroj] Autor

Obrázek č. 1 Příklady typů literálů přijímané jazykem Swift

3.3.3. Proměnné a konstanty

Pro veškerou operaci s literály a složitějšími datovými typy slouží proměnné a konstanty. Proměnné jsou definovány klíčovým slovem *var*. Konstanty jsou deklarované slovem *let*. Klíčová vlastnost, která rozlišuje proměnné a konstanty, je ta, že proměnné umožňují změnu hodnot, zatímco konstanty nelze po prvním zápisu hodnoty měnit. Po klíčovém slově následuje pojmenování proměnné, či konstanty. Rovnítkem „ = “ se zapíše hodnota. Dvojtečkou „ : “ po pojmenování lze specifikovat datový typ, ale není to vyžadováno, je-li hodnota zadána při inicializaci. Viz obrázek č. 2.

```
var prázdnáProměnná:String
var proměnná = "Toto je proměnná"
let konstanta = "Tohle je konstanta"
var character:Character = "D"

proměnná = "Proměnnou lze změnit"
konstanta = "Konstantu nelze změnit"
proměnná = 5.3      // Proměnnou nelze přetypovat na jiný datový typ
```

[zdroj] Autor

Obrázek č. 2 Příklady zadání proměnných a konstant v jazyku Swift

Jazyk Swift je typově chráněný. Jakmile jsou tedy proměnné jednou inicializovány, nelze již změnit jejich datový typ a kompilátor ohlásí chybu, pokud by mělo dojít k přetypování proměnné. Na obrázku č. 2 je uveden příklad s proměnnou s názvem *proměnná* na posledním řádku, kde z typu *String* by se měla proměnná změnit na typ *Double*.

3.3.4. Optionals

Optionals ve významu k programování lze do češtiny přeložit jako volitelný. Tento nový typ umožňuje vyjádřit, že hodnota objektu nemusí být přítomná a může mít prázdnou hodnotu *nil*.

V jazycích C a Objective-C se tento typ nevyskytuje, neboť každý objekt v C jazycích může vrátit *nil*, a to kvůli tomu, že volaný objekt není validní, nebo že hodnota nebyla nalezena. Přidáním k objektu typ *optionals* určujeme, s jakou hodnotou přesně program pracuje, což zvyšuje typovou ochranu jazyka Swift a zbavuje interpretace prázdné hodnoty *nil* jako chybový stav.

Optionals se označují otazníkem „ ? “ na konci pojmenování. Pro přístup k neprázdné hodnotě se používá znaménko vykřičníku „ ! “, ale je-li hodnota prázdná vyvolá to *runtime chybu* . Proto je nutné objekt před vyvoláním hodnoty ošetřit. Viz obrázek č. 3.

```
var poznamka:String? = "Volitelná poznámka"  // Tato proměnná je typ optional
poznamka = nil
if poznamka != nil {
    print(poznamka!)    // Přidáním vykřičníku se zajistí přístup k hodnotě
}
```

[zdroj] Autor

Obrázek č. 3 Příklad užití optionals v jazyce Swift

3.3.5. Základní operátory

Operátor je zvláštní symbol, se kterým lze hodnoty kontrolovat, měnit či kombinovat. Swift podporuje téměř všechny standardní operátory jazyka C. Některé operátory přibyly jako například *nil-compaunding* „ ?? “ operátor, který kontroluje, zdali hodnota není prázdná. Nové jsou i intervalové operátory „ … “, který lze využít dobře ve *for-in* cyklech. V následující tabulce č. 3 jsou uvedeny základní operandy jazyka Swift.

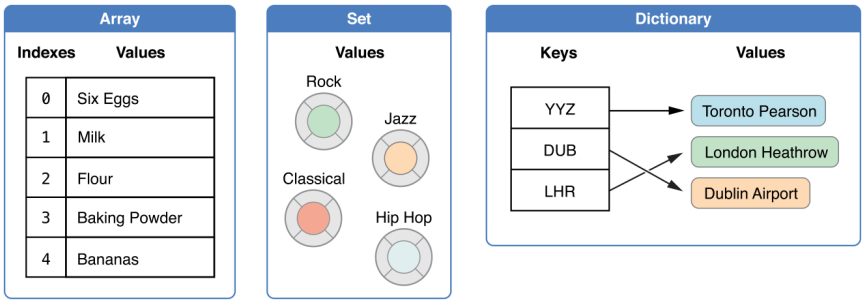
Operátor	Vlastnost	Příklad
+	Aritmetický operátor sčítání	5 + 4 // se rovná 9
-	Aritmetický operátor odčítání	7 – 3 // se rovná 4
*	Aritmetický operátor násobení	3 * 4 // se rovná 12
/	Aritmetický operátor dělení	10.0 / 2.5 // se rovná 4
%	Zbytkový operátor modulo	9 % 4 // se rovná 1
=	Přiřazovací operátor	a = b
==	Porovnávací operátor „je rovno“	1 == 1
!=	Porovnávací operátor „není rovno“	1 != 2
>=	Porovnávací operátor „větší nebo rovno“	1 >= 1

<=	Porovnávací operátor „menší nebo rovno“	2 <= 3
>	Porovnávací operátor „větší než“	2 > 1
<	Porovnávací operátor „menší než	1 < 2
-	Unární operátor negativního čísla	-1
+	Unární operátor pozitivního čísla	+1
!	Logický unární operátor negace	!a
+=	Zahrnující operátor sčítání	a += 2
-=	Zahrnující operátor odčítání	b –= 1
q ? a : b	Trojitý operátor podmínky	otázka ? řešení1 : řešení2
??	Nil-compaunding operátor	prázdné ?? hodnota
…	Uzavřený intervalový operátor	1…5
..<	Polootevřený intervalový operátor	1..<6
&&	Logický operátor AND	a && b
||	Logický operátor OR	a || b

Tabulka č. 3 Základní operátory v jazyce Swift

3.3.6. Kolekce

Swift poskytuje tři primární typy kolekcí: *Arrays* (z angl. pole), *Sets* (z angl. sady) a *Dictionaries* (z angl. slovníky). Pole je seřazený seznam hodnot. Sada je neseřazená množina unikátních hodnot. Slovník je neseřazená kolekce dvojic klíčů a hodnot. Je-li jakýkoli z těchto typů kolekcí definován jako proměnná *var*, lze po vytvoření kolekce přidávat, odstraňovat a měnit položky v dané kolekci. Definováním pole, sady či slovníku jako konstanty *let*, po vytvoření konstanty je velikost a obsah neměnitelný. Datové typy hodnoty v kolekcích jsou vždy jasně dané a nelze omylem vložit špatný typ. Detailnější informace o kolekcích lze nalézt v oficiální dokumentaci. [1]



[zdroj] upraveno z dokumentace The Swift Programming Language [1]

Obrázek č. 4 Vizualizace ukládání kolekcí do typů Array, Set a Dictionary

3.3.7. Řízení toku

Swift používá pro řízení toku klasické prvky jako jsou podmínky *if* a *switch* a smyčky *for-in*, *while* a *repeat-while* (analogický stejné jako *do-while*). Podmínka *if* má stejnou formu zápisu jako v jazycích C. Podmínky následované za klíčovým slovem není nutné uzavírat do závorek.

```
let výsledek = 67
var hodnocení: String

if výsledek < 60 {
    hodnocení = "Neprospěl"
} else {
    hodnocení = "Prospěl"
}
```

[zdroj] Autor

Obrázek č. 5 Příklad užití podmínky *if* v jazyce Swift

Switch má také převzatou formu z jazyka C s jedním podstatným rozdílem, a to, že ve výchozím stavu není nutné u každého případu *case* zapisovat příkaz *break* pro vystoupení ze *switch* podmínky. Pokud programátor chce, aby po splnění jednoho příkazu *case* pokračoval i v dalších kontrolách *case,* je nutné zadat na konci nové klíčové slovo *fallthrough*.

```
let číselnýPočet = 42
var přirozenýPočet: String

switch číselnýPočet {
case 0:
    přirozenýPočet = "Žádný"
case 1..<20:
    přirozenýPočet = "Několik"
case 20..<100:
    přirozenýPočet = "Desítky"
case 100..<1000:
    přirozenýPočet = "Stovky"
default:
    přirozenýPočet = "Mnoho"
}
```

[zdroj] Autor

Obrázek č. 6 Příklad užití podmínky *switch* v jazyce Swift

Smyčky *while*, *repeat-while* mají téměř stejný zápis i stejnou funkcionalitu jako v jazycích C. Je vytvořená podmínka, která dokud je pravdivá, opakuje blok kódu, který je zapsaný ve smyčce. *For-in* smyčka umožňuje iteraci přes danou sekvenci, jako je interval čísel, položky v poli, či znaky ve řetězci *String*. Cyklus lze opustit pomocí klíčového slova *break*. Pro přeskočení na další iteraci se použije klíčové slovo *continue*.

```swift
let ovoce = ["Pomeranč", "Jablko", "Třešně"]
var iterace = 0

// While smyčka
while iterace < ovoce.count {
    print(ovoce[iterace])
    iterace += 1
}

// Repeat-While smyčka
var opakování = 0
repeat {
    print(ovoce[opakování])
    opakování += 1
} while opakování < ovoce.count

// For-In
for druh in ovoce {
    print(druh)
}
```

[zdroj] Autor

Obrázek č. 7 Příklad užití smyček while, repeat-while a for-in v jazyce Swift

3.3.8. Funkce

Funkce jsou bloky kódu, které vykonávají specifický úkol. Dle běžné konvence se funkce pojmenovávají podle toho, co vykonávají a dle jména se i funkce volají. V jazyce Swift je sjednocená syntax dost flexibilní od vyjádření jednoduchých funkcí bez parametru ve stylu jazyků C až po komplexní metody s pojmenovanými argumenty pro každý parametr.

Možnosti funkcí v jazyce Swift:

- Návratová hodnota funkce může být jedna, žádná i několikanásobná
- Funkce může být bezparametrová i vícenásobnými parametry
- Jako parametr lze zapsat i další funkci
- Parametr může mít výchozí hodnotu
- Funkce lze pomocí intervalového operátoru „ … “ nedefinovaný počet operátorů
- Parametr může zůstat nepojmenovaný pomocí podtržítka „ _ “
- Parametr může být proměnou pomocí klíčového slova *inout*

```swift
func pozdrav(jméno: String) {
    print("Ahoj, \(jméno)!")
}

pozdrav(jméno: "Karle")     // Vypíše se "Ahoj, Karle!"

func sumaAPrůměr (_ čísla: Double...) -> (suma: Double, průměr: Double) {
    var suma: Double = 0
    for položka in čísla {
        suma += položka
    }
    let průměr = suma / Double(čísla.count)
    return (suma, průměr)
}

let (suma, průměr) = sumaAPrůměr(1, 2, 3, 4, 5) // Funkce vrátí n-tici (15, 3)
print("Suma: \(suma)\tPrůměr: \(průměr)")

func vyměňČísla(_ a: inout Int, _ b: inout Int) {
    let temporaryA = a
    a = b
    b = temporaryA
}

var číslo1 = 25
var číslo2 = 42

vyměňČísla(&číslo1, &číslo2)

print("Číslo 1 je \(číslo1)") // "Číslo 1 je 42"
```

[zdroj] Autor

Obrázek č. 8 Příklad užití funkcí v jazyce Swift

Na

Obrázek č. 8 jsou 3 funkce: *pozdrav, sumaAPrůměr, vyměňČísla*. První funkce *pozdrav* obsahuje parametr jméno typu *String*, která nemá žádnou návratovou hodnotu. Druhá funkce *sumaAPrůměr* přijímají neomezený počet parametrů typu *Double*. Návratovou hodnotu je n-tice *suma* a *průměr*, které jsou též typu *Double*. Poslední funkce *vyměňČísla* je bez návratové hodnoty s parametry typu *inout*, které vkládají proměnnou vloženou do funkce a daná funkce může změnit její hodnotu. Je nutné zadat znak ampersandu „ & “ k proměnné, které vkládáme do funkce, je-li typu *inout*. Proměnná *číslo1*, tak po vykonání funkce *vyměňČísla* má hodnotu 42.

3.3.9. Další vlastnosti jazyka Swift

Programovací jazyk Swift poskytuje řadu dalších klíčových vlastností, jako jsou třídy, enumerace, protokoly apod. V následující tabulce jsou shrnuty vlastnosti jazyka Swift s jejich stručnou definicí. Další podrobnosti a příklady užití lze nalézt v oficiální dokumentaci [1] .

Vlastnost jazyka Swift	Definice
Closures	*Closures* (z angl. uzavření) je funkcionalita uzavřena v bloku a může být vložena do kódu.
Enumerace	Enumerace seskupuje příbuzné hodnoty a lze s nimi pracovat typově-chráněné formě.
Třídy	Třída je prvkem převzatým z objektově-orientovaného programování. Swift však nevyžaduje rozdělení na rozhraní a implementaci.
Struktury	Struktury jsou podobné třídám. V jazycích C struktury typicky obsahovali pouze hodnoty, v jazyce Swift mohou obsahovat i metody.
Properties	*Property* (z angl. vlastnost) je hodnota, která je přiřazena nějaké třídě, struktuře či enumeraci a patří dané instanci. Uložená hodnota může být konstanta, či proměnná.
Metody	Metoda je funkce, která je spojena s konkrétním třídou, strukturou či enumerací a zapouzdřuje specifické úkoly a funkcionalitu dané instance.
Subscript	Třídy, struktury a enumerace mohou definovat *subscript*, který je zkratkou pro přístup k elementům kolekcí, seznamu či sekvence.
Dědičnost	Dědičnost je vlastnost, kde třída může přebírat metody, *property* a další charakteristiky z další třídy
ARC	ARC je zkratka pro *Automatic Reference Counting* (z angl. automatické počítání referencí), která řídí správu paměti. Díky ARC se programátor nemusí starat o uvolňování paměti a vše se děje automaticky.
Řetězení Optionals	Řetězení datového typu *Optional* je proces dotazování a volání *property*, metod či *subscript,* zdali hodnota může být nil.
Ošetřování výjimek	Swift pro ošetřování chybových stavů poskytuje plnou podporu. V jazyce Swift lze chybový stav vyvolat, zachytit, propagovat a manipulovat za chodu programu.
Typové obsazení	Typové obsazené umožňuje zkontrolovat typ dané instance či pracovat s určitou instancí jako její podtřídou či nadtřídou.
Vnořené typy	Vnořené typy pomáhají specifickým třídám, strukturám či enumeracím

	podpořit jejich funkcionalitu.
Extensions	*Extensions* (z angl. rozšíření) umožňuje přidat novou funkcionalitu do již existující třídě, struktuře, enumeraci či protokolu.
Protokoly	Protokol je plán metod, *properties* a dalších požadavků, které slouží specifickému účelu, nebo části funkcionality. Protokol může být adaptován třídou, strukturou nebo enumerací.
Generičnost	Generičnost neboli obecnost umožňuje psát flexibilní a znovu použitelné funkce a typy, které mohou pracovat s jakýmkoli typem, jenž je programátorem definován.
Kontrola přístupů	Kontrola přístupů zabraňuje přístupu částem zadaného kódu do dalších souborů a modulů. Pomocí klíčových slov: *open, public, internal, fileprivate, private* jsou definovány jednotlivé přístupy.
Pokročilé operátory	Pokročilé operátory umožňují komplexnější manipulace s hodnotami, jako jsou bitové operace, přesouvací operátory, známé z jazyka C a Objective-C.

Tabulka č. 4 Další vlastnosti jazyka Swift a jejich definice

3.4. Porovnání programovacích jazyků

Programovací jazyk Swift bude porovnán se systémovým jazykem C a s dalšími populárními vysokoúrovňovými jazyky, jako je Objective-C, C++ a Java. Jazyk Swift bude také poměřen se skriptovacími jazyky Python a JavaScript.

3.4.1. Jazyk C

Programovací jazyk C byl vyvinut v 70.letech 20.století při vývoji operačního systému UNIX. Je to systémový a imperativní jazyk, který nabízí vlastnosti jako jsou rekurzivní funkce, víceúrovňové ukazatele, alokace paměti za běhu, uživatelsky definované typy, ukazatele funkcí a malé, ale flexibilní sada příkazů.

Jazyk C byl navržen, aby byl výkonný, pružný a efektivní. Komunikace s prostředím je řešena přes standardní knihovnu funkcí, která se ve skutečnosti skládá ze 2 částí: *jazyk jádra* a *standardní knihovny*.

Duch jazyka C je výstižně zachycena v motivacích v knize *Rationale for the Ansi — programming language* C [28] :

- Důvěřovat programátorovi,
- Nebránit programátorovi v tom, co má být provedeno,
- Udržovat jazyk malý a jednoduchý,
- Poskytovat pouze jednu cestu pro vykonání operace,
- Rychlost je více preferovaná než portabilita. [28]

V roce 1988 byl jazyk C standardizován organizací ISO a o rok později americkou organizací ANSI. [10]

3.4.2. Jazyk C++

Jazyk C++ byl vyvinut v roce 1979 v rámci diplomové práce Bjarneho Stroustrupa. Tehdy byl tento jazyk pojmenován „C with Classes“. Hlavní motivací vytvoření nového jazyka bylo poskytnout dovednosti jazyka Simula s efektivitou a flexibilitou jazyka C. V roce 1985 byla poprvé komerčně vydána kniha „C++ Programming Language“, v níž ale jazyk nebyl ještě standardizován. Jazyk C++ byl standardizován až v roce 1991.

C++ měl od počátku navržené vlastnosti, jako např. třídy, odvozené třídy, konstruktory a destruktory, kontrola přístupů, typová kontrola a implicitní konverze argumentů funkcí. Později přibyly další vlastnosti: vnořené funkce, výchozí argumenty, přetížení přiřazovacího operátoru, virtuální funkce, přetížení jmen funkcí a operátorů, reference, konstanty a další minoritní možnosti. Obecné požadavky při návrhu jazyka C++ byly:

- Být lepší než jazyk C,
- Podporovat datovou abstrakci,
- Podporovat objektově-orientované programování,
- Podporovat generičnost

C++ v současnosti nadále zůstává jazykem pro vývoj systémů. Jazyk aproximuje ideály ostatních jazyků, ačkoli není svým ideálem. Ideálem C++ je: sada nižších programovacích jazykových mechanismů ovládajících hardware přímo v kombinaci s výkonnými kompozičními abstraktními mechanismy a minimální závislost na prostředí za běhu. [11] [12]

3.4.3. Objective-C

Objective-C vznikal v 80. letech zásluhou svou vývojářů: Toma Lovea a Brada Coxe. Vývoj jazyka Objective-C probíhal zcela nezávisle na jazyku C++ a byl významně ovlivněn jazykem Smalltalk. Roku 1986 byla zveřejněna první publikace, která popisovala nově vytvořený jazyk Objective-C. V následujících letech si tento jazyk licencovala společnost NEXT vedená Stevem Jobsem a s pomocí tohoto jazyka vyvinula operační systém NEXTSTEP. Roku 1996 firmu NEXT převzala společností Apple. Součástí akvizice byl i programovací jazyk Objective-C, který společnost Apple využila společně s knihovnami operačního systému NEXTSTEP do nově vytvářeného systému Mac OS X.

Syntaxe jazyka Objective-C byla navržena tak, aby byla malá, jednoznačná a snadno naučitelná. Objective-C je skutečnou nadstavbou jazyka C a lze využít veškerou funkcionalitu, kterou jazyk C nabízí. V porovnání s ostatními objektivně orientovanými jazyky je Objective-C velmi dynamický. Kompilátor zanechává spoustu informací o objektech samotných při použití systému za běhu. Rozhodnutí, která by jinak byla vytvořena v kompilačním čase, mohou být odložena, dokud program není spuštěný. Dynamičnost dává programům napsaným v Objective-C výkon a flexibilitu, což jsou dva velké benefity, které je těžké získat od Objective-C. [13] [14]

3.4.4. Java

Javu začal vyvíjet třináctičlenný *Green Team* ve firmě Sun v roce 1991. Tato tvůrčí skupina věřila v novou vlnu počítačové revoluce, která měla na starost digitální spotřebitelská zařízení a počítače. *Green Team* prezentoval schopnosti nového jazyka s interaktivními domácími zařízeními, který cílil na digitální televizní průmysl. Představený televizní koncept se neuchytil, ale nastupovala doba Internetu. V roce 1995 byl uveden prohlížeč Netscape Navigator Internet, který měl v sobě integrovanou technologii Java.

Jazyk Java je součástí celé platformy Java, která má na starosti kompilaci i spuštění přeloženého kódu. Součástí platformy je Java Virtual Machine (JVM), která komunikuje s jádrem operačního systému. Dále je součástí platformy API, které obsahuje veškeré knihovny pro potřebný chod aplikace. Velikost Java API se liší podle zvolené edice,

kterých v současnosti existují č: Java Card, Java ME (Micro Edition), Java SE (Standard Edition), Java EE (Enterprise Edition). Klíčové principy při návrhu Javy byly:

- Jednoduchost, objektová orientovanost a povědomost
- Robustnost i bezpečnost
- Nezávislost na architektuře a přenositelnost
- Výkonost
- Interpretovaný jazyk, podpora více vláken a dynamičnost

[15] [16]

3.4.5. Python

Python vytvořil koncem 80.let 20. století nizozemský programátor Guido van Rossum. Tento jazyk byl silně inspirován programovacím jazykem ABC. Pojmenování získal po britském komediálním pořadu *Monty Python's Flying Circus*. První verze byla vydána roku 1994.

Jazyk Python je multiplatformní, interpretovaný, objektově-orientovaný a strukturovaný jazyk. Pro svou jednoduchost v syntaxi je považován za začátečnický programovací jazyk. Python je navržen, aby byl snadno rozšiřitelný, a tak lze například vnořit novou část funkcionality do již existující aplikace. Python využívá dynamické typování, počítání referencí a *garbage collector* pro správu paměti. Primární filozofie jazyka shrnuje dokument „Zen of Python", jehož klíčové vlastnosti jsou:

- Hezké je lepší než ošklivé,
- Explicitní je lepší než implicitní,
- Jednoduché je lepší než komplexní,
- Komplexní je lepší než komplikované,
- Srozumitelnost se počítá.

[17] [18]

3.5. Návrhové vzory

Návrhové vzory (angl. *Design Patterns*) jsou doporučené postupy řešení často se vyskytujících se úloh. [9]

Obecně se vzor skládá ze 4 základních prvků:

1. **Název vzoru** je záhlavím, které lze použít k popisu návrhového problému, jeho řešení a důsledků jedním nebo dvěma slovy.
2. **Problém** popisuje, kde se má vzor používat. Vysvětluje problém a jeho kontext.
3. **Řešení** definuje prvky, z nichž se návrh skládá – jejich vztahy, povinnosti a spolupráci. Řešení však neposkytuje určitý konkrétní návrh nebo implementaci, protože vzor je jako šablona, kterou lze využít v různých situacích.
4. **Důsledky** jsou výsledky a kompromisy vzoru. [20]

3.5.1. Katalog návrhových vzorů

Zde je abecedně seřazený katalog návrhových vzorů s jejich anglickým pojmenováním a stručnou charakteristikou.

Název návrhového vzoru	Charakteristika
Abstraktní továrna (*Abstract Factory*)	Poskytuje rozhraní pro vytváření řad příbuzných nebo závislých objektů, aniž by se museli specifikovat konkrétní třídy.
Adaptér (*Adapter*)	Převádí rozhraní třídy na jiné rozhraní, které očekávají klienti. Umožňuje společnou funkci tříd, které jinak nemohou spolupracovat kvůli neslučitelným rozhraním.
Dekorátor (*Decorator*)	Dynamicky připojuje další povinnosti k objektu. Na rozdíl od tvorby podtříd, které jinak nemohou spolupracovat kvůli neslučitelným rozhraním.
Fasáda (*Facade*)	Poskytuje unifikované rozhraní pro sadu subsystémových rozhraní. Fasáda definuje rozhraní na vyšší úrovni, které usnadňuje použití subsystému.
Interpret (*Interpreter*)	V daném jazyce definuje vyjádření jeho gramatiky včetně překladače, který vyjádření použije k interpretaci vět.
Iterátor (*Iterator*)	Poskytuje možnost sekvenčního přístupu k prvkům vícečlenného objektu, aniž by odhaloval jeho vyjádření.
Jedináček (*Singleton*)	Zajišťuje, aby jedna třída měla pouze jednu instanci, a poskytuje k ní globální přístupový kód.
Most (*Bridge*)	Aby se mohly abstrakce a implementace nezávisle měnit, dělí tuto

	dvojici na dvě samostatné části.
Muší váha (*Flyweight*)	Využívá ke sdílení účinné podpory většího počtu jemnozrnných objektů.
Návštěvník (*Visitor*)	Vyjadřuje operaci, která se má provést na prvcích objektové struktury. Návštěvník umožňuje definovat novou operaci, aniž by se měnily třídy příslušných prvků
Obnovitel (*Memento*)	Aby mohl objekt obnovit svůj původní stav, zachycuje a rozšiřuje vnitřní stav objektu, aniž by se porušilo jeho zapouzdření.
Pozorovatel (*Observer*)	Definuje meziobjektovou závislost 1 : n. Změní-li jeden objekt svůj stav, všechny závislé objekty jsou automaticky upozorněny a zaktualizovány.
Prostředník (*Mediator*)	Definuj objekt, který zapouzdřuje způsoby interakce v sadě objektů. Prostředník podporuje volné spřažení tím, že brání objektům ve vzájemných explicitních odkazech, a umožňuje jejich interakce nezávisle měnit.
Prototyp (*Prototype*)	Specifikuje vytvářené druhy objektů pomocí prototypické instance. Nové objekty vytváří tak, že tento prototyp kopíruje.
Příkaz (*Command*)	Zapouzdří žádost do objektu, čímž umožňuje parametrizovat klienty pomocí různých žádostí, např. řadících nebo protokolovaných, a podporovat nevratné operace.
Řetěz odpovědností (Chain of Responsibility)	Vyhýbá se spřažení odesílatele a příjemce žádosti tím, že dává příležitost ke zpracování žádosti více objektům. Vytváří řetěz objektů a předává žádost po jeho článcích, až ji některý objekt zpracuje.
Skladba, Strom (*Composite*)	K vyjádření hierarchií část za celek, skládá objekty do stromových struktur. Umožňuje klientům jednotné zacházení s jednotlivými objekty i s jejich strukturou.
Stav (*State*)	Umožňuje objektu změnit své chování, když se změní jeho vnitřní stav. Objekt tak zdánlivě mění svou třídu.
Stavitel (*Builder*)	Aby mohl stejný konstrukční proces vytvářet různá vyjádření, odděluje sestrojení komplexního objektu od jeho vyjádření.
Strategie (*Strategy*)	Definuje řadu algoritmů, každý zapouzdří a umožní jejich

	zaměnitelnost. Strategie umožňuje algoritmu, aby se měnil nezávisle na klientech, kteří jej používají.
Šablonová metoda (*Template Method*)	Definuje kostru algoritmu v nějaké operaci a odkládá některé záležitosti na podtřídy. Šablonová metoda umožňuje podtřídám předefinovat jisté kroky algoritmu, aniž by měnily jeho strukturu.
Tovární metoda (*Factory Method*)	Definuje rozhraní pro vytváření objektu. Rozhodnutí, u které třídy se má spustit její instance, ale přenechává podtřídám. Tovární metoda umožňuje třídě, aby odložila rozhodnutí o vytvoření instance na své podtřídy.
Zástupce (*Proxy*)	Poskytuje náhradníka nebo místo držitele jiného objektu za účelem řízení jeho přístupu.

Tabulka č. 5 Katalog návrhových vzorů se stručnou charakteristikou [20]

3.5.2. Organizace katalogu

Návrhové vzory se liší svojí granularitou i úrovní abstrakce. Tento oddíl návrhové vzory klasifikuje. Tato klasifikace urychluje výuku vzorů v katalogu, a může nás také nasměrovat při hledání vzorů nových.

Účel				
Oblast	Třída	Tvořivý	Strukturální	Chování
		Tovární metoda	Adaptér (třída)	Interpret
				Šablonová metoda
	Objekt	Abstraktní továrna	Adaptér (objekt)	Iterátor
		Jedináček	Dekorátor	Návštěvník
		Prototyp	Fasáda	Obnovitel
		Stavitel	Most	Pozorovatel
			Muší váha	Prostředník
			Skladba	Příkaz
			Zástupce	Řetěz odpovědnosti
				Stav
				Strategie

Tabulka č. 6 Organizace katalogu návrhových vzorů

4 Praktická část

V následující části bude provedeno praktické zkoumání jazyka Swift. Prvně se prozkoumá zdrojový kód a jeho nejdůležitější prvky. Další část vyhodnotí syntax, tu se kromě zápisu kódu bude řešit i jmenovací konvence a kompilace kódu. Příklady užití se zaměří na silné, ale i okrajové části jazyka. Poslední částí této kapitoly bude porovnání s programovacími jazyky z hlediska syntaxe a výkonu.

4.1. Analýza zdrojového kódu

Oficiální stránky projektu Swift jsou na adrese: http://swift.org. Jsou zde veškeré informace, aktuality, dokumentace i odkaz na zdrojové kódy, které jsou k dispozici přes službu GitHub. Na vývoji Swiftu se mohou podílet všichni a Apple vítá veškeré připomínky, opravy chyb či návrhy na zlepšení, ovšem pouze Apple rozhoduje, které změny budou součástí Swiftu.

Cílem kapitoly *Analýza zdrojového kódu* je prozkoumat strukturu repozitáře zdrojového kódu Swiftu, ukázat nejdůležitější prvky pro stavbu jazyka a představit Swift komunitu.

4.1.1. Zdrojový kód Swiftu

Repozitář zdrojových kódů Swiftu se nachází na adrese: https://github.com/apple/swift. Zde je možné celý obsah repozitáře stáhnout do počítače, ovšem změny do hlavní vývojové větve *master* lze nahrát do online repozitáře pouze po schvalovacím procesu.

Ve verzi Swift 4.0 se nachází 13 adresářů a 7 souborů. V následující tabulce je uvedena stručná charakteristika adresářů, které jsou strukturovány následovně:

Položka v repozitáři	Charakteristika
apinotes	Soubory, které pomáhají mapovat Objective-C API pro kompilátor Swiftu.
benchmark	Testovací platforma pro měření výkonu. Testy jsou napsány v jazyce Swift.
bindings	XML schéma pro vygenerování dokumentace v psaném kódu.

cmake	Soubory pro vytváření binárních souborů Swiftu.
docs	Dokumentace k jednotlivým částem zdrojového kódu.
include	Nízkoúrovňová knihovna Swiftu, kde se nachází hlavičky všech souborů.
lib	Implementační soubory nízkoúrovňové knihovny Swift psané v jazyce C++.
stdlib/public	Knihovna Swiftu na vyšší úrovni. Soubory jsou psané v jazyce Swift, C++ a Objective-C.
stdlib/private	Privátní knihovna, která slouží k budování prototypu a užívání interních prvků jazyka Swift. Apple nepublikuje soubory do veřejného repozitáře.
test	Podrobné testy, které zkoušejí kvalitu jednotlivých komponent. Testy jsou napsány v jazyce Swift.
tools	Zdrojové soubory k testovacím aplikacím, které navazují přímo na knihovnu Swiftu.
utils	Podpůrné testy, data a programy. Aplikace a skripty jsou psány v jazycích Python a Bash.
unittests	Testují nižší úrovně knihovny Swift. Testy jsou psány v jazyce C++.
validation-tests	Kolekce všech testů pro validaci výstupu po vybudování jazyka Swift. Psáno v jazyce Swift.

Tabulka č. 7 Struktura repozitáře Swiftu

4.1.2. Kompatibilita verzí Swiftu

Swift jako nový programovací jazyk je v neustálém vývoji a každý rok je vydávána nová verze. Mezi hlavní cíle jazyka Swift je i kompatibilita mezi různými verzemi Swiftu. Rozlišují se 2 druhy kompatibility, které jazyk Swift podporuje:

- Kompatibilita zdrojového kódu,
- Kompatibilita za běhu binárních frameworků.

Kompatibilita zdrojového kódu znamená, že novější kompilátory umí zkompilovat starší kód. Vývojáři, tak nemusí celý projekt ve Swiftu přepisovat na novější verzi jazyka. Xcode také poskytuje nástroj *Swift Migration Assistant*, kterým může vývojář svůj projekt aktualizovat na novější verzi.

Kompatibilita za běhu a binárních frameworků umožňuje spolupráci již zkompilovaných implementací, které mají jiné verze. Swift pro tuto nízkoúrovňovou kompatibilitu využívá ABI stabilitu. ABI je anglický akronym pro *Application Binary Interface* a umožňuje navázat nezávisle zkompilované binární soubory mezi sebou na nižší úrovni systému. ABI stabilita je zajišťována tak, aby budoucí verze kompilátorů produkovaly binární soubory odpovídající stabilnímu ABI.

Mezi komponenty Swift ABI patří:

1. Typy jako struktury a třídy, které musí mít definované uspořádání v paměti pro každou instanci.
2. Typová metadata, která pokrývají programy Swift, běžící procesy Swift a jiné nástroje pro ladění či vizualizaci.
3. Jakýkoli exportovaný či externí symbol v knihovně. Tyto symboly potřebují unikátní jméno, na kterých se binární entity shodnou. Unikátní jména jsou produkována pomocí techniky nazvané *mandlování jmen* (angl. Name Mangling).
4. Funkce, které musí znát, jak se navzájem funkce volají. *Konvence volání* (*Calling Conventions*) obsahuje věci jako volací zásobník, registry, jež jsou uchovávány a vlastnické konvence.
5. Runtime knihovna, která je součástí Swiftu, operuje např. s dynamickým obsazováním paměti či počítáním referencí. Tato runtime knihovna je Swift ABI samotné.
6. Standardní knihovna Swiftu, která definuje všechny běžné typy struktury a operace. Každá nová verze standardní knihovny musí mít zafixované API, aby Swift ABI bylo kompatibilní se staršími verzemi.

4.1.3. Podporované platformy

Swift je denně budován denně pomocí open-source nástroje Jenkins na stránkách https://ci.swift.org/. Mezi podporovanými platformami jsou všechny Apple platformy: *macOS, iOS, watchOS* a *tvOS*. Dále sem paří i Linux platforma, konktrétně distribuce Ubuntu. Výstupem je kompilátor a pomocné nástroje, které umožňují vytvářet programy na daných platformách.

Jako oficiální cestu pro vývoj aplikací je nutné použít vývojové prostředí Xcode, které je kompatibilní pouze s platformou macOS. S aplikací Xcode je standardně dodáván balíček SDK pro všechny Apple platformy. SDK umožňuje vytvářet plnohodnotné aplikace s UI. V červnu 2017, kdy proběhla světová konference WWDC 2017, bylo k dispozici 250 000 aplikací na App Store, které byly vyvinuty s pomocí jazyka Swift. [21]

4.1.4. Komunita Swift

Swift je otevřený veřejnosti a všichni vývojáři mohou přispět k vývoji jazyka Swift. Komunikovat přímo s vývojáři Swiftu lze pomocí specializovaných emailových seznamů. Všechny emailové adresy a jejich účel jsou k dispozici na adrese: https://swift.org/community/#mailing-lists. Pomocí emailové skupiny Swift Evolution (swift-evolution@swift.org) je možné i diskutovat o nápadech a návrzích pro další vývoj jazyka Swift. Dostupná je možnost zaregistrovat si příjem emailu, které sdělují novinky a různá oznámení. Apple také zveřejňuje hlavní členy týmu odpovědný za vývoj Swiftu. Vedoucím týmu v roce 2017 je Ted Kremenek, kremenek@apple.com.

Komunikace má také svá pravidla sepsaná v dokumentu *Code of Conduct* (z angl. pravidla chování). Je zakázáno jakékoli nežádoucí chování, jako např.: jakákoli diskriminace, obtěžování, osobní útoky a neetické a neprofesionální vystupování.

Nahlašování chyb lze provádět prostřednictvím webové aplikace JIRA na webové adrese https://bugs.swift.com. Pro správnou registraci chyby do systému JIRA, je nutné:

- Prozkoumat stávající hlášení chyb v systému, zdali neexistuje duplikát,
- Sepsat srozumitelně popis problému,
- Napsat postup, jak chybu reprodukovat,
- Vypsat verzi použitého výstupu Swiftu spolu s verzemi operačního systému a SDK.

Kontribuce do kódu Swiftu jsou doporučovány pomocí menších inkrementálních změn, kdykoli je to možné. Dlouhodobé změny potřebují vlastní vývojovou větev, nezávislou na hlavní větvi, a v takovém případě je nezbytné kontaktovat Swift team. *Commit* zprávy do Gitu musí splňovat pokyny, které jsou sepsány na adrese: https://swift.org/contributing/#contributing-code. Na zmíněné adrese jsou i doporučené

hlavičky souborů zdrojových kódu, které definují, pod jakou licencí je tato práce chráněna a co daný kód zpracovává.

4.1.5. Licence Swift

Swift je licencovaný pod Apache 2.0., což je svobodná softwarová licence, která od uživatele vyžaduje zachování autorství a zřeknutí se odpovědnosti. Tato verze je kompatibilní s *GNU Genaral Public Licence* verze 3.0.

Swift licence má ještě výjimku *Runtime Libray Exception*, která ujasňuje koncovým uživatelům, že kompilátor Swiftu nemusí přisuzovat licenci Apache na jejich koncový produkt, který používá jazyk Swift, a mohou se svým produktem nakládat dle svého uvážení.

4.2. Syntaxe jazyka Swift

Tato kapitola představí skladbu jazyka Swift jak z pohledu člověka, který kód píše, tak z pohledu člověka, který kód reviduje. Podstatný je i pohled počítače, resp. kompilátoru, který jazyk překládá na spustitelný soubor.

4.2.1. Zápis kódu

Jazyk Swift je na první pohled velice jednoduchý a přehledný. Kombinuje prvky známé ze starších jazyků C a modernějších jazyků, jako jsou Java, Python a Ruby. Pro zkušené programátory tak nepředstavuje velkou komplikaci přejít na jazyk Swift. Nováčci bez zkušenosti jiných programovacích jazyků také nebudou mít problém pochopit Swift. Apple se snaží popularizovat tento jazyk mezi širší publikum s aplikacemi jako je *Playgrounds*, které jsou dostupné pro iPady. V oficiálních obchodech Apple je možné se i účastnit přednášek, na nichž se lze tento jazyk naučit.

```swift
enum Enumerace {
    case jedna
    case dva
}
protocol Protokol {
    var proměnná: Any { get set }
    func funkce(parameter: Any) -> Any
}
class Třída : Protokol {
    let konstanta: Any
    var proměnná: Any

    init (proměnná: Any, konstanta: Any) {
        self.proměnná = proměnná
        self.konstanta = konstanta
    }
    func funkce(parameter: Any) -> Any {
        return parameter
    }
}
let pravda = true
if pravda {
    let mojeTřída = Třída(proměnná: "a", konstanta: "b")
    mojeTřída.funkce(parameter: "c")
}
```

[zdroj] Autor

Obrázek č. 9 Ukázka syntaxe základních typů Swiftu

Samotná syntaxe jazyka Swift je velice přehledná a srozumitelná. Má jasně definovaný rámec, jak lze kód zapsat. Jako příklad lze uvést obrázek č. 9, který ukazuje zápis základních prvků, jako je třída, protokol a volání metod. Tento zápis se podobá v některých ohledech zápisu pseudokódu, který zobrazuje pouze neformální způsob zapsání kódu. To ulehčuje zápis algoritmu a díky vývojovému prostředí Xcode jej lze i rychle otestovat.

Ovšem komplexnost narůstá se složitostí vyvíjeného projektu. Spousta Applem dodávaných knihoven a API byly původně napsány v Objective-C a některá rozhraní se spoléhala čistě na rozhraní jazyka C. Jako konkrétní příklad lze uvést *Grand Central Dispatch* API, které bylo upraveno až ve verzi Swift 3.0, kde získalo kompatibilní Swift rozhraní. Viz obrázek č. 10, kde je stejné rozhraní pro vytvoření asynchronního procesu na frontě *gcdQueue*. Ve verzi Swift 2.0 je forma zápisu de facto stejná jako v Objective-C. Ve verzi Swift 3.0 má *Grand Central Dispatch* API plnou Swift podporu a k tomu odpovídající zápis.

```
// Swift 2.0
func oldAsyncThread() {
    let gcdQueue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0)
    dispatch_async(gcdQueue) {
        // Some async process on gcdQueue
    }
}

// Swift 3.0
func newAsyncThread() {
    let gcdQueue = DispatchQueue.global(qos: .default)
    gcdQueue.async() {
        // Some async process on gcdQueue
    }
}
```

[zdroj] Autor

Obrázek č. 10 Příklad rozdílné syntaxe mezi Swift 2.0 a Swift 3.0

Vývojáři již existujících aplikací pro Apple platformy, kteří se rozhodli zapojit Swift do projektu, mají možnost si určit v jakém rozsahu budou Swift do projektu implementovat. Apple vydal elektronickou knihu *Using Swift with Cocoa and Objective-C*, kde jsou přesně sepsány postupy a ukázky příkladů. Kniha je rozdělena do tří částí, jež pokrývají nejdůležitější aspekty kompatibility Swift a Objective-C:

- **Interoperabilita**, která dovoluje mít jedno rozhraní napsané buď ve Swiftu, nebo Objective-C, a poté umožňuje rozhraní použít v obou jazycích.
- **Mix and match** dovoluje použít oba jazyky v rámci jedné aplikace a vzájemnou komunikaci Swift a Objective-C souborů.
- **Migrace** z Objective-C na Swift.

[23]

Z mé programátorské zkušenosti je vzájemná kompatibilita obou jazyků na dobré úrovni. Lze psát kód v Objective-C, který je pak možné použít ve Swiftu, a naopak napsat kód ve Swiftu, který je plně kompatibilní s Objective-C. Ovšem pro vzájemnou spolupráci je v mnoha případech nutné přidat klíčová slova pro výjimky a přesnou specifikaci chování, aby kompilátor při překladu neohlašoval chybu. Jako příklad lze uvést, že Objective-C nemá volitelné (*optionals*) proměnné, jelikož každá proměnná může být *nil*. Proto má Objective-C nové klíčové slovo *nullable*, které specifikuje, že daná proměnná může mít prázdnou hodnotu.

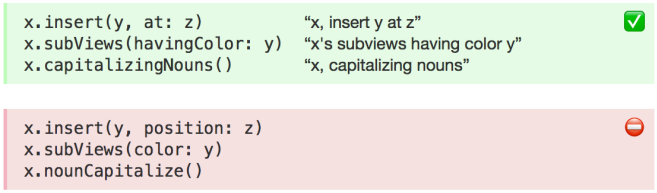
4.2.2. Jmenovací konvence

Pro lepší čitelnost a srozumitelného psaného kódu, má každý programovací jazyk jmenovací konvence, což je soubor pravidel, dle kterých by se měl programátor řídit. Tato

pravidla nejsou vymahatelná kompilátorem a neoznačují se ve většině případech jako chyby. Ovšem jmenovací konvence jsou komunitou vývojářů brány jako dogma; pro jazyk Swift jsou jmenovací konvence zdokumentovány na stránce https://swift.org/documentation/api-design-guidelines/.

Základní principy, jichž by se měl programátor Swift držet, je být přímočarý v místě použití. Při návrhu API je nutné mít na mysli, že entity jako *property*, nebo metody jsou často volány a mají být při použití stručné a jasné. Stručné názvy by však neměly být za jakoukoli cenu krátké. Dále každá deklarace by měla mít komentář.

Většina programovacích jazyků používá anglickou terminologii a Swift není výjimkou. Swift nabádá, aby jména funkcí používala angličtinu gramaticky správně, jak je vidět na obrázku č. 11. Zkratkám by se měl vývojář vyhnout, stejně tak nezřetelným termínům. Jména funkcí by se měla také zdůraznit, aby bylo jasné, zdali se při volání mění, nebo se jím jen vrací již hotová neměnná hodnota.



[zdroj] upraveno z dokumentace API Design Guidelines [24]

Obrázek č. 11 Příklad správného a špatného pojmenování funkcí

Mezi další konvence Swiftu patří:

- Třídy, protokoly a datové typy se píší typem *Camel Case,* a to tzv. *UpperCamelCase,* tudíž se zvádějí slova bez mezer a všechna slova mají na začátku s velké písmeno.
- Ostatní názvy používají *lowerCamelCase*, tudíž začínají malým písmenem na začátku slova.
- Metody lze přetěžovat, tj. mohou sdílet stejné jméno s jiným parametrem, pokud vykonávají stejnou operaci.
- Zdokumentovat komplexnost operací, je-li větší než O(1).
- Jména parametrů funkce by měly sloužit dokumentaci.
- Všechny argumenty musí být pojmenované. [24]

Jmenovací konvence Swiftu nejsou pro zkušené programátory nic nového. Dokumentace jen ulehčuje hlavní body a příklady správného i špatného použití a také vyzdvihuje výhody a novinky jazyka Swift.

4.2.3. Kompilace kódu

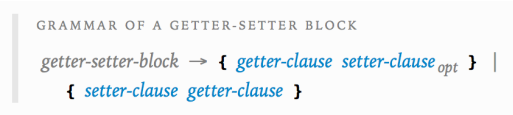
Kompilace kódu převádí zdrojový kód na strojový kód. Architektura kompilátoru Swiftu má 7 hlavních komponent:

- **Parsování** je nástroj, který sestupně rekurzivně prochází kód a vytváří tzv. *Abstract Syntax Tree* (AST) bez sémantické či typové informace a upozorňuje na chyby v gramatice zdrojového kódu.
- **Sémantická analýza** je odpovědná za transformaci zparsovaného AST na plně typově zkontrolovanou formu AST. Je-li analýza úspěšná a bez chyb, indikuje že je bezpečné vygenerovat kód.
- **Clang importer** vkládá moduly Clang a mapuje C a Objective-C API.
- **SIL generation**. *Swift Intermediate Language* (SIL) je vysokoúrovňový specifický jazyk, který slouží pro další analýzu a optimalizaci Swift kódu. Tato fáze vygeneruje tento *SIL*.
- **SIL garantované transformace** vykonává přidanou diagnostiku pro celkovou správnost programu. Výstupem je kanonický SIL.
- **SIL optimalizace** spouští specializované vysokoúrovňové optimalizace jako např. ARC optimalizace.
- **LLVM IR generace**. IR, neboli *Instruction Reference* vygeneruje nízkoúrovňový SIL pro LLVM kompilátor, který vygeneruje strojový kód.

[24]

Swift má v oficiální dokumentaci podrobně sepsanou gramatiku, dle které se zdrojový kód zpracovává. Lexikální struktura se dělí na: mezery, identifikátory, literály, operátory, typy, výrazy, deklarace, vzory, parametry a argumenty. Tyto struktury se poté dělí na další konkrétnější subtypy a ty mají mezi sebou propojené vztahy. Na obrázku č. 12 je vidět, jak je v dokumentaci znázorněn vztah *getter-setter-block*, kde první varianta musí mít *getter-*

clause a pak následuje *setter-clause* kvůli označení *opt*, nebo druhá varianta musí mít *setter-clause* a po ní následuje *getter-clause*.



[zdroj] upraveno z dokumentace The Swift Programming Language [1]

Obrázek č. 12 Gramatika pro getter-setter blok

S kompilátorem samotným se mnoho programátorů přímo nesetkává. Díky dobře zdokumentované gramatice a syntaxi jazyka Swift může Xcode lépe ukazovat chyby a upozornění na přesná místa kódu v reálném čase, a nikoli až po spuštění kompilátoru. Syntaxe moderního jazyka by měla být jednoduchá, ale přitom flexibilní a bezpečná, což Swift zvládá velmi dobře.

4.3. Příklady užití

Tato kapitola na základě autorem sestavených příkladů představí charakteristiky jazyka Swift v běžné praxi. Příklady jsou vybrány tak, aby ukázaly hlavní výhody jazyka a také jeho limity. Příklady budou také doplněny komentářem, zda dané řešení splňuje hodnoty jazyka Swift, jako je jednoduchost, rychlost a srozumitelnost.

4.3.1. Funkce

Funkce ve Swiftu jsou jedny z fundamentálních prvků jazyka, které charakterizují jazyk Swift jako takový. Tvar zápisu funkce není převzat z dosud známých programovacích jazyků, nicméně určitá podobnost je zachována. Zápis je jednoduchý a srozumitelný, avšak nabízí velkou flexibilitu využití.

Funkce již není pouze operace, ale validní datový typ. Tím splňuje vlastnosti pro funkční paradigma, které přebírá trend objektově-orientovaného programování. Jako příklad slouží obrázek č. 13, který definuje funkci s parametrem *tajemství*, která vrací typ *String* a proměnná *operace* má jako datový typ určenou funkci, jež přijímá i vrací *String*, která funkce *pošliTajemství* splňuje.

```
func pošliTajemství(tajemství: String) -> String { return "Posílam tajemství: \(tajemství)" }
var operace: (String) -> (String) = pošliTajemství(tajemství:)
operace("Mýdlo")
```

[zdroj] Autor

Obrázek č. 13 Funkce jako proměnná

Funkce získávají nový rozměr jako proměnné, ale funkce mohou i samotným parametrem, jak můžeme vidět na obrázku č. 14. Funkce *proveďOperaci* má tři parametry, přičemž první je funkce, která přijímá dva parametry typu *Double* a vrací jeden typ *Double*. Zbývající dva parametry jsou čísla *a* a *b*. Funkce *sečti* a *odečti* mají také parametry, *a* a *b,* a vrací výsledek typu Double. Volání *proveďOperaci* tak může mít jiný výsledek, zvolíme-li jinou funkci, ale číselné parametry zůstanou stejné.

```
func sečti(_ a: Double, _ b: Double) -> Double { return a + b }

func odečti(_ a: Double, _ b: Double) -> Double { return a - b }

// Funkce jako parametr
func proveďOperaci(_ operace: (Double, Double) -> Double, _ a: Double, _ b: Double) -> Double {
    let result = operace(a, b)
    print(result)
    return result
}
proveďOperaci(sečti, 3.4, 2.1 )     // 5.5
proveďOperaci(odečti, 3.2, 2.1 )    // 1.1
```

[zdroj] Autor

Obrázek č. 14 Funkce jako parametr jiné funkce

Funkce může být i výstupem další funkce viz obrázek č. 15, kde *posunoutSe* vrací funkci *jítDopředu*, *jítDozadu* nebo *zůstat* na základě vstupního parametru *početKroků*.

```
// Funkce jako výstup
func posunoutSe(početKroků: Int) -> () {
    if početKroků > 0 { return jítDopředu(kroky: početKroků) }
    else if početKroků < 0 { return jítDozadu(kroky: početKroků) }
    else { return zůstat() }
}

func jítDopředu(kroky: Int) { print("Dopředu o \(kroky)")}
func jítDozadu(kroky: Int) { print("Dozadu o \(kroky)")}
func zůstat() { print("Zůstávám na místě") }
```

[zdroj] Autor

Obrázek č. 15 Funkce jako návratová hodnota

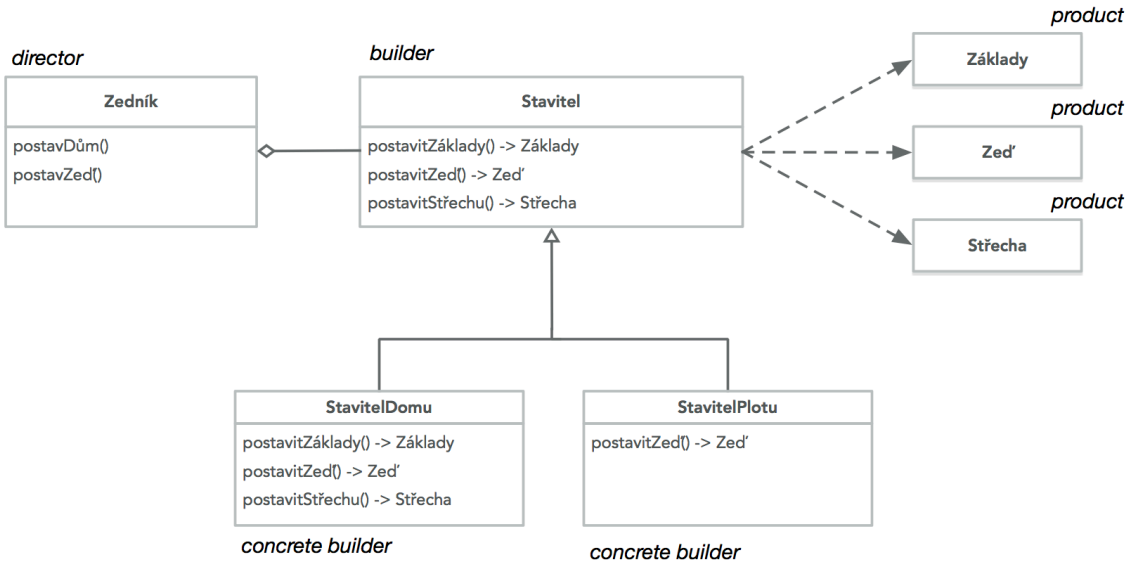
V jiných nefunkčně programovacích jazycích takové využití funkcí není bez složitých obstrukcí možné. Novější verze Javy a C# již podporují výše ukázanou implementaci,

ovšem Swift má tuto vlastnost zabudovanou již od počátku a zápis není složitě vykonstruovaný.

4.3.2. Návrhové vzory

Návrhové vzory ve Swiftu mohou být navrženy a implementovány stejně jako u jazyka Objective-C. Swift jako multiparadigmatický jazyk umí zápis objektově-orientovaných návrhových vzorů bez větších problémů. Proto se kapitola nazvaná Návrhové vzory zaměří pouze na vzorové příklady, které reflektují možnosti jazyka Swift, a nikoli samotné implementace návrhových vzorů.

Při programování návrhových vzorů je nejdůležitější návrh řešení. Postup při implementaci návrhových vzorů je volný, pokud splňuje návrh rozhraní. Jako první příklad bude implementace návrhového vzoru Stavitel – *Builder*, který je znázorněny a obrázku č. 16 pomocí UML diagramu tříd.



[zdroj] Autor

Obrázek č. 16 UML diagram tříd návrhového vzoru Stavitel

Návrhový vzor *Stavitel* řeší, aby stejný konstrukční proces mohl vytvářet různá vyjádření. To znamená, že objekt *zedník* může pomocí funkcí v abstraktní třídě *Stavitel* postavit různé objekty. Funkce *postavitZeď* může sloužit k postavení plotu nebo zdi domu. Abstraktní třídy ve Swiftu ani v Objective-C nejsou k dispozici jako v Javě. Místo toho lze využít

protokolů (*protocol*) a rozšíření (*extension*). Na obrázku č. 17 je úryvek implementace návrhové vzoru.

```
/* Builder */
protocol Stavitel {
    func postavitZáklady() -> Základy?
    func postavitZeď() -> Zeď?
    func postavitStřechu() -> Střecha?
}

extension Stavitel {
    func postavitZáklady() -> Základy? { return nil }
    func postavitZeď() -> Zeď? { return nil }
    func postavitStřechu() -> Střecha? { return nil}
}
/* Concrete Builder */
class StavitelPlotu: Stavitel {
    func postavitZeď() -> Zeď { return Zeď() }
}
```

[zdroj] Autor

Obrázek č. 17 Implementace návrhového vzoru stavitel

Jelikož ve Swiftu nelze označit funkce v protokolu jako volitelné (*optional*), jak to bylo možné dříve v Objective-C, stejnojmenné rozšíření *Stavitel* k protokolu přidává výchozí implementaci, která ve všech případech vrací prázdnou hodnotu *nil*. Třída *StavitelPlotu*, pak dědí jak protokol, tak i rozšíření, ovšem pouze metoda *postavitZeď* vrací žádaný produkt. Zbývající metody lze přes třídu *StavitelPlotu* pořád volat, ale nevrací žádný výsledek.

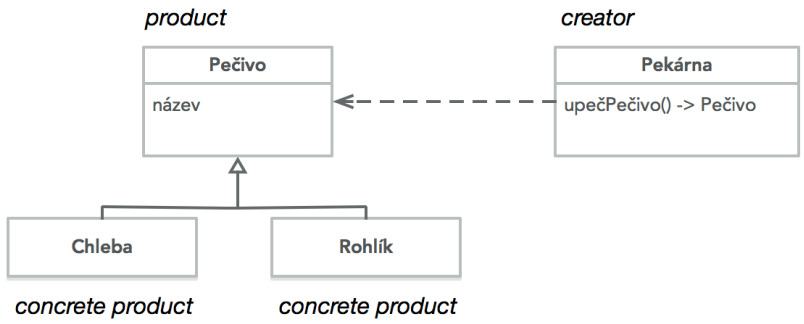
Swift jako jazyk s funkčním paradigma má nové možnosti, jak přistupovat k řešení návrhových vzorů. Přednáška Scotta Wlaschina, týkající se návrhových vzorů ve funkčním programování, představila možnosti, jak používané návrhové vzory v objektově-orientovaném programování lze zcela nahradit funkcemi. [26]

OO pattern/principle	FP pattern/principle
• Single Responsibility Principle	• Functions
• Open/Closed principle	• Functions
• Dependency Inversion Principle	• Functions, also
• Interface Segregation Principle	• Functions
• Factory pattern	• Yes, functions
• Strategy pattern	• Oh my, functions again!
• Decorator pattern	• Functions
• Visitor pattern	• Functions

[zdroj] převzato z prezentace Functional Programming Design Patterns [26]

Obrázek č. 18 Stránka z přednášky Scotta Wlaschina o nahrazení návrhových vzoru funkčním programováním

Následující příklad demonstruje použití objektově-orientovaného přístupu a funkčního přístupu v návrhovém vzoru *Tovární metody – Factory Method*. Na obrázku č. 19 je tovární metoda *upečPečivo*, která vytváří produkt *Chleba*, nebo *Rohlík*.



[zdroj] Autor

Obrázek č. 19 UML diagram tříd Tovární metody

Pro objektově orientované řešení ve Swiftu existuje několik možností, jak tento návrh tříd implementovat. Pokud tovární metoda nesdílí nějaký společný kód, lze celou třídu *Pekárna* předělat na protokol. Další možností je využít šablon. Třetí možností je vytvořit enumeraci produktů, která vytvoří daný produkt.

```swift
enum TypPečiva {
    case chleba
    case rohlík
}
/* Creator */
class Pekárna {
    func upečPečivo(typ: TypPečiva) -> Pečivo {
        var pečivo: Pečivo
        switch typ {
        case .chleba:
            pečivo = Chleba()
        case .rohlík:
            pečivo = Rohlík()
        }
        return pečivo
    }
}
var pekárna = Pekárna()
var rohlík:Pečivo = pekárna.upečPečivo(typ: .rohlík)
```

[zdroj] Autor

Obrázek č. 20 Implementace Tovární metody pomocí principu OOP

Následující obrázek č. 20 ukazuje implementaci tovární třídy *Pekárna* a vytvoření produktu *Rohlík*. Pro rozlišní produktu, který bude vytvořen je využita enumerace *TypPečiva*. Voláním posledního řádku se vytvoří produkt *rohlík*.

Funkční programování ve Swiftu se však může zcela zbavit enumerace. Na obrázku č. 21 je jako parametr tovární metody použitá funkce namísto enumerace. Na předposledním řádku je definovaný typ zvláštní typ funkce *makeChleba*, která vrací produkt. Pro vytvoření produktu *chleba* pak vložíme tuto zvláštní funkci jako parametr, jak znázorněno na posledním řádku obrázku č. 21.

```swift
/* Creator */
class Pekárna {
    func upečPečivo(typPečiva: () -> Pečivo) -> Pečivo {
        print("Upečen \(typPečiva().název)")
        return typPečiva()
    }
}
var mojePekárna = Pekárna()
var makeChleba = { return Chleba() as Pečivo }
var chleba: Pečivo = mojePekárna.upečPečivo(typPečiva: makeChleba)
```

[zdroj] Autor

Obrázek č. 21 Implementace Tovární metody pomocí principu funkčního programování

Funkční princip programování lze ve Swiftu efektivně využít. Tento princip však stále není v obecném povědomí vývojářů. Swift proto velmi efektivně kombinuje oba myšlenkové světy objektově-orientovaného programování a funkčního programování. Dle mého názoru nelze návrhové vzory zcela nahradit funkcemi, neboť k tomu nejsou uzpůsobena API ani

jiné systémy. Swift však dává nové možnosti při navrhování architektury nových systémů a funkce dokáží zefektivnit systém mnohem více.

4.3.3. Práce s ukazateli

Ukazatel je datový typ, který slouží ke zpřístupnění dat na dané paměťové adrese. V jazycích C, C++ či Objective-C jsou ukazatele označovány znakem hvězdy (*) a často slouží jako reference pro daný datový typ. Proměnné a konstanty ve Swiftu nevyžadují zvláštní označení pro vytvoření reference, ovšem nejsou přímým ukazatelem na adresu v paměti.

Hlavní důvod, proč reference ve Swiftu neukazují na adresu paměti, je ochrana před fatálními chybami způsobenými programátorem. Swift si totiž řídí veškerou operace s pamětí automaticky pomocí ARC, automatického počítání referencí. Ovšem některé nízkoúrovňové systémy vyžadují přímý zásah do paměti a Swift to umožňuje pomocí speciálních typů nazvaných *UnsafePointer*. Existuje několik typů ukazatelů *UnsafePointer*, které slouží pro plnou podporu operací z jazyka C.

Jako příklad poslouží funkce, která vymění ukazatele na číslo typu *Int*. Pro porovnání zápisu je stejný příklad zapsán v jazyce C a ve Swiftu. Na obrázku č. 22 je zápis v jazyce C. Můžeme si povšimnout, že proměnná *ukazatel* je označena hvězdičkou a ampersand u proměnné *cislo* vrací adresu. Voláním funkce *vymenCislo* se přehodí ukazatel na proměnnou *cislo2*, aniž by funkce měla návratovou hodnotu. Takovým způsobem se často obcházelo řešení, že funkce má jen jednu návratovou hodnotu.

```c
int cislo = 16;
int *ukazatel = &cislo;

void vymenCislo(int **pointer) {
    int cislo2 = 32;
    *pointer = &cislo2;
}

printf("Ukazatel ukazuje na číslo: %d\n", *ukazatel);   // 16
vymenCislo(&ukazatel);
printf("Ukazatel ukazuje na číslo: %d\n", *ukazatel);   // 32
```

[zdroj] Autor

Obrázek č. 22 Příklad práce s ukazateli v jazyce C

Příklad zapsaný ve Swiftu (viz obrázek č. 23) je již mnohem komplikovanější na zápis. Pro konstantu *ukazatel* je zvolen typ *UnsafeMutablePointer* typu *Int* a je alokován na jednu adresu v paměti. Voláním funkce *initialize* s parametrem *číslo* se adresa uloží do konstanty *ukazatel*. Tím, že je *ukazatel* měnitelný, lze vyměnit adresu ve funkci *vyměňČíslo*, ale pořád zůstává konstantou, neboť se nemění ukazatel samotný, pouze adresa.

```
let číslo: Int = 16
let ukazatel: UnsafeMutablePointer<Int> = UnsafeMutablePointer<Int>.allocate(capacity: 1)
ukazatel.initialize(to: číslo)

func vyměňČíslo(pointer: UnsafeMutablePointer<Int>) {
    let číslo2: Int = 32
    pointer.initialize(to: číslo2)
}

print("Ukazatel ukazuje na číslo: \(ukazatel.pointee)")  // 16
vyměňČíslo(pointer: ukazatel)
print("Ukazatel ukazuje na číslo: \(ukazatel.pointee)")  // 32
```

[zdroj] Autor

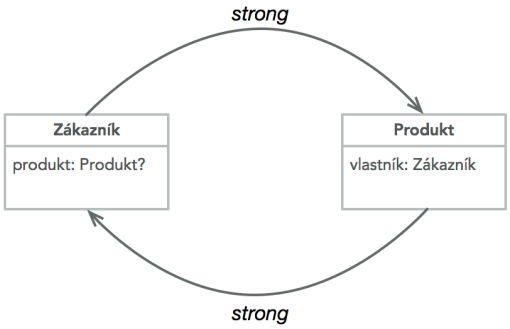
Obrázek č. 23 Příklad práce s ukazateli ve Swiftu

Ukazatele ve Swiftu jsou překvapivě plně podporované, ačkoli jejich zápis je velmi netypický. Pro programátory využívající jazyk C je však toto řešení nepraktické. Pro vytváření nízkoúrovňových systémů jako jsou např. ovladače hardwaru, se Swift nehodí, ale Swift nabízí plnohodnotné řešení, pokud je nutné ukazatele použít.

4.3.4. Práce s pamětí

Swift si řídí veškerou práci s pamětí automaticky. Používá k tomu technologii nazvanou ARC (angl. *Automatic Reference Counting)*, tj. automatické počítání referencí. Jakmile vznikne reference na objekt, systém automaticky započítá objekt do systému a přidělí mu optimální množství paměti. Na jeden objekt může během běžícího procesu vzniknout spousty referencí a každá reference je započítána do ARC. Jakmile poslední reference na objekt zanikne, systém objekt zahodí a uvolní paměť. Programátor tak ve Swiftu nemá žádnou starost s alokováním a uvolňováním paměti.

Mohou však nastat situace, kdy se paměť neuvolní. Například pokud jsou dva objekty *Zákazník* a *Produkt* na sobě navázané, jak je uvedeno na obrázku č. 24, a je k němu vytvořen kód na obrázku č. 25. Každá reference je ve výchozím stavu typu *strong* neboli silný typ reference.



[zdroj] Autor

Obrázek č. 24 Diagram označující silné reference na sebe navazující objekty

Pokud reference *zakaznik* uvolní objekt, tedy má hodnotu *nil*, pak objekt bez jakýchkoli jiných vazeb zanikne. Ovšem objekt *zakaznik* ještě silně drží reference na objekt *prod*. Na posledním řádku je vidět, že objekt *Zákazník* pořád existuje. Problém by nastal, pokud by i reference *prod* byla nastavena na *nil*, neboť tím, že objekty na sebe navazují, jejich vazby nebyly vůbec narušeny. Tím by došlo k nechtěnému úniku paměti.

```
class Zákazník { var produkt: Produkt? }

class Produkt {
    let vlastník: Zákazník
    init(owner: Zákazník) { vlastník = owner }
}

var zakaznik: Zákazník? = Zákazník()
var prod: Produkt? = Produkt(owner: zakaznik!)
zakaznik?.produkt = prod!

zakaznik = nil
prod?.vlastník
```

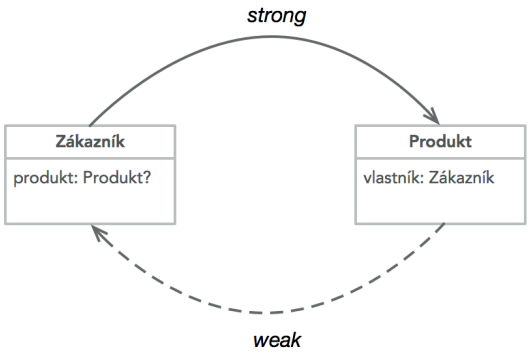
```
Zákazník
Produkt
()

nil
Zákazník
```

[zdroj] Autor

Obrázek č. 25 Implementace silných referencí na sebe navazující objekty

Aby se předešlo nechtěným únikům paměti, existuje i slabý typ referencí, které mají klíčové slovo *weak*. Na obrázku č. 26 je diagram stejného příkladu, který, kde objekt *Produkt* má slabou vazbu na objekt *Zákazníka*.



[zdroj] Autor

Obrázek č. 26 Diagram slabé reference na sebe propojené objekty

Na obrázku č. 27 je kód příkladu se slabou referencí. Můžeme si povšimnout, že proměnná *vlastník* má nyní v definici označení *weak*. Také již tento objekt není konstanta *let*, ale proměnná *var* a *optional*, neboť slabé reference mohou být pouze proměnné. Na předposledním je opět objekt *zakaznik* nastaven na *nil*. Tentokrát však již objekt je uvolněn z paměti, což dokazuje poslední řádek a reference *vlastník* na stejný objekt je také *nil*.

```
class Zákazník { var produkt: Produkt? }

class Produkt {
    weak var vlastník: Zákazník?
    init(owner: Zákazník) { vlastník = owner }
}

var zakaznik: Zákazník? = Zákazník()
var prod: Produkt? = Produkt(owner: zakaznik!)
zakaznik?.produkt = prod!

zakaznik = nil
prod?.vlastník
```

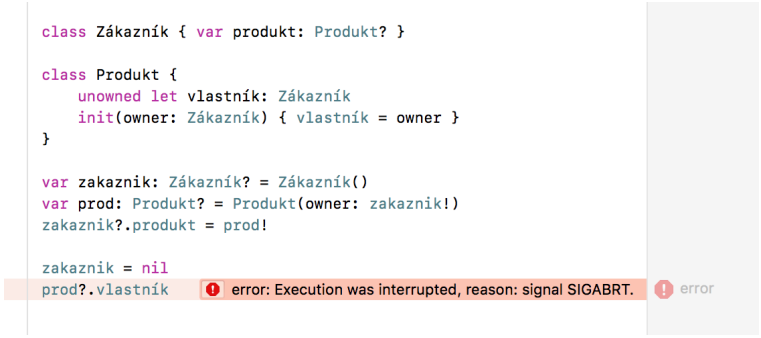
```
Zákazník
Produkt
()

nil
nil
```

[zdroj] Autor

Obrázek č. 27 Implementace slabé reference na sebe navázané objekty

Existuje ještě třetí typ referencí, který se označuje *unowned*. Tento typ reference nenavyšuje referenční počet. Reference *unowned*, na rozdíl od *weak* reference, nemůže mít hodnotu nil. Je důležité upozornit, že reference *unowned* nemůže ukazovat na uvolněný objekt, neboť to vede k chybě za běhu programu, jak lze vidět na obrázku č. 28. Pokud již objekt *zakaznik* neexistuje, voláním reference *vlastník* na neexistující objekt vyvolá fatální chybu, která vede k pádu programu.



[zdroj] Autor

Obrázek č. 28 Implementace reference unowned na sebe navázané objekty

Pro lepší přehlednost, jaký typ referencí lze v konkrétních případech použít, je sestavena tabulka č. 8, kde sloupce označují typ, zdali je proměnná (*var*), či konstanta (*let*), a zdali je hodnota volitelná (*optional*).

	Var	Let	Optional	Non-Optional
Strong	✓	✓	✓	✓
Weak	✓	✗	✓	✗
Unowned	✓	✓	✗	✓

Tabulka č. 8 Povolené typy referencí pro různé držitele dat

Swift jako vysokoúrovňový jazyk obsluhuje veškeré řízení paměti automaticky, ovšem programátor se nemůže zcela spolehnout na systém řízení paměti. Jak bylo ukázáno na příkladu, pořád jsou možné úniky paměti, což může běžící systém negativně ovlivnit. Jazyk Java nerozlišuje typ referencí a vše řídí tzv. *Garbage Collector*, který se spravuje automaticky, tudíž programátor nemůže nic ovlivnit. ARC je tak optimální kompromis, kde programátor může ovlivnit uvolnění paměti a řízení nevyžaduje komplikované zásahy, které by destabilizovaly běh programu.

4.3.5. Swift jako skriptovací jazyk

Swift je také skriptovací jazyk. Ačkoli zmínky o Swiftu jako o skriptovacím jazyku v oficiální dokumentaci nejsou k dispozici, tato možnost existuje, a dokonce není potřeba kód kompilovat. Shebang neboli řádek, který označuje cestu, kde skript bude interpretován, je pro Swift takovýto: ***#!/usr/bin/swift***. Ve skriptu lze i importovat API, které jsou k dispozici pro platformu, na které skript běží.

Následující příklad na obrázku č. 29 je skript, který vymazává soubor pomocí API *FileManger* přijatý jako argument v příkazové řádce. Je-li podáno více argumentů či žádný argument, skript ohlásí chybu.

```
#!/usr/bin/swift

import Foundation

let arg = CommandLine.arguments  // Store first argument.
if arg.count == 2 {
    let cesta: String = arg[1]
    let cestaJeValidní: Bool = FileManager.default.fileExists(atPath: cesta)
    if cestaJeValidní {
        try FileManager.default.removeItem(atPath: cesta)
    } else {
        print("Chyba. Cesta k souboru: \(cesta) je nevalidní.")
    }
} else {
    print("Chyba. Skript \(arg[0]) vyžaduje pouze jeden argument.")
}
```

[zdroj] Autor

Obrázek č. 29 Příklad skriptu napsaném v jazyce Swift

Pro zkušené programátory Swiftu, tak může být využití tohoto jazyka pro skripty mnohem přijatelnější, pokud nemají zkušenosti se skriptovacími jazyky jako Python či Ruby. Nedoporučoval bych však Swift jako náhradu za skriptovací jazyky. Syntaxe Swiftu se může v budoucím vývoji změnit a je vyžadován kompilátor, který není standardní součástí distribuce operačního systému. Proto je lepší vytvářet kompilované programy ve Swiftu než interpretované skripty.

4.4. Porovnání s programovacími jazyky

Tato kapitola bude porovnávat jazyky Objective-C a Python s jazykem Swift. Objective-C byl pro srovnání vybrán záměrně, neboť Swift má ambice ho zcela nahradit. Python je velmi populární programovací jazyk, který je vhodný pro začátečníky a je plně podporovaný operačním systémem macOS. Ostatní jazyky jako Java a C++ nejsou přímou součástí platformy Mac a jejich porovnání nebude dávat v kontextu na kritéria velký smysl.

Hlavními kritérii srovnání budou porovnání syntaxe a jednoduchost zápisu. Také bude bráno v potaz využití v rámci platformy a dostupných API. Samotná kapitola Výkon porovná čas srovnání objektů pomocí algoritmu Quick Sort.

4.4.1. Objective-C

Objective-C je již přes tři dekády starý jazyk. Nikdy nedosáhl takové popularity jako dnes, kdy téměř každá iOS aplikace vznikla za pomocí Objective-C. Ovšem Apple a jeho předchůdce NEXTSTEP jsou jediné firmy, které implementovaly tento jazyk do své platformy. Objective-C tak byl do nástupu iPhone zcela minoritní jazyk.

V současnosti již Objective-C nesplňuje předpoklady pro moderní vysokoúrovňový jazyk. Hlavním důvodem, proč je Objective-C zastaralý, je právě syntax, která není podobná žádnému z rozšířených jazyků. Následující obrázek č. 30 demonstruje proč.

```
#import "MyClass.h"

// Rozhraní třídy MyClass
@interface MyClass ()
// Hodnota uložená v objektu MyClass
@property NSString *myProperty;
// Metoda
- (NSString *)myMethod:(NSInteger)parameter;
@end

// Implementace třídy MyClass
@implementation MyClass

// Implementace metody My Method
- (NSString *)myMethod:(NSInteger)parameter {
    // Příklad volání metody stringWithFormat
    NSString *text = [NSString stringWithFormat:@"Vložený parametr %li", parameter];
    // Uložení hodnoty do myProperty
    _myProperty = text;

    return text;
}

@end
```

[zdroj] Autor

Obrázek č. 30 Příklad syntaxe v Objective-C

Třída v Objective-C je rozdělena do dvou různých souborů: rozhraní a implementace. V souboru rozhraní jsou veřejné metody a hodnoty, které jsou přístupné i pro jiné třídy a objekty. V implementačním souboru, který je vyobrazen na obrázku č. 30, je zapsán kód a privátní metody. Tato logika rozdělení dává smysl a je převzatá z jazyka C a C++. Zápis volání metod je však unikátní, neboť volání je uzavřeno do hranatých závorek jako například *stringWithFormat*. Až nová verze Objective-C umožňuje použití tečky k přístupu hodnot na daném objektu jako je tomu u Javy, avšak pro volání metod se tento zápis neujal.

Swift a Objective-C nemají rozdíl ve funkcionalitě. S oběma jazyky lze vytvořit plnohodnotné aplikace, systémy, knihovny apod. Liší se pouze přístup, kde jazyk Swift přebírá moderní přístupy zápisu kódu, což je pouze rozhraní a základy pořád běží na

jazyku C a upraveném LLVM kompilátoru. Apple podporuje oba jazyky stejně a v souvislosti s vývojem Swiftu se nepatrně změnil i Objective-C.

4.4.2. Python

Python je interpretovaný jazyk, který se kompiluje až při spuštění. Na současném operačnímu systému macOS High Siera 10.13 se nachází verze 2.7, ačkoli současná verze je 3.6. Rozdíl mezi verzemi Pythonu je v odlišných knihovnách. Starší verze systému macOS stále spoléhají na stabilní starší verzi.

Python je všeobecně populární jazyk díky své jednoduchosti a čitelnosti. Není potřeba u proměnných definovat datový typ, u funkcí se nezadává návratová hodnota a kód lze snadno spustit. U menších skriptů a programů to zjednodušuje psaní programů. Při komplexních implementacích systémů v Pythonu by však udržování kódu a ladění vyžadovalo enormní úsilí. Důvodem je, že Python není typově chráněn a programátor může snadno udělat chybu, která se zjistí až za běhu programu.

```
#!/usr/bin/env python

class MyClass:
    myProperty = ""

    def myMethod(self, parameter):
        text = 'This is my parameter {}'.format(parameter)
        self.myProperty = text
        return myProperty
```

[zdroj] Autor

Obrázek č. 31 Příklad syntaxe jazyka Python

Na obrázku č. 31 je ukázka zápisu kódu v Pythonu. Třída ani funkce nejsou vyhraněné složitými závorkami, ale pouze dvojtečkou. Nový řádek je odsunutý o čtyři místa vpravo. Zápis je čitelný a jasný. U parametru funkce *myMethod* není definovaný typ ani návratová hodnota. Pouze programátor ví, co je vstupem a výstupem metody. Pokud není kód řádně zdokumentovaný, což není běžná praxe, tak se velmi komplikuje kontrola a úprava kódu pro další participanty.

Swift a Python jsou svojí jednoduchostí podobné jazyky, ale Swift přistupuje k bezpečnosti mnohem seriózněji. Python je výborný jazyk, který seznamuje nováčky s programováním a na spoustu malých projektů a pomocných utilit je to ideální nástroj. Swift má ambice stát se jazykem pro nováčky, který naučí programovat novou generaci vývojářů. Python však

v současnosti nemůže Swift ohrozit, a to díky široké obci uživatelů Pythonu a multiplatformní podpoře na Windows, Mac i Linux.

4.4.3. Výkon

Pro porovnání výkonu Swiftu, Objective-C a Pythonu byl vytvořen stejný algoritmus *Quick Sort*, který porovná 10 000 objektů vlastní třídy *Číslo*. Cílem je zjistit rychlost inicializace třídy a rychlost vzestupného seřazení objektů. Všechny tři programy byly spuštěny z jednoho počítače této konfigurace: *MacBook Pro 13,2; 3,1 GHz Intel Core i5; 16 GB RAM, macOS 10.13.2*. Na obrázku č. 32 je stejná metoda *quickSort* vyjádřena v různých jazycích

Python

```python
def quickSort(list):
    if len(list) <= 1:
        return list
    else:
        pivot = list[0].hodnota

        smallerList = []
        equalList = []
        biggerList = []

        for x in list:
            if x.hodnota < pivot:
                smallerList.append(x)
            elif x.hodnota == pivot:
                equalList.append(x)
            elif x.hodnota > pivot:
                biggerList.append(x)

        return quickSort(smallerList) + equalList + quickSort(biggerList)
```

Objective-C

```objective-c
+ (NSArray *)quickSort:(NSArray<Cisla*>*)list {
    if (list.count <= 1) {
        return list;
    }
    NSInteger pivot = list[0].hodnota;

    NSMutableArray<Cisla *> *smallerList = [[NSMutableArray alloc] init];
    NSMutableArray<Cisla *> *equalList = [[NSMutableArray alloc] init];
    NSMutableArray<Cisla *> *biggerList = [[NSMutableArray alloc] init];

    for (Cisla *x in list) {
        if (x.hodnota < pivot) {
            [smallerList addObject:x];
        } else if (x.hodnota == pivot) {
            [equalList addObject:x];
        } else if (x.hodnota > pivot) {
            [biggerList addObject:x];
        }
    }

    NSMutableArray *result = [[NSMutableArray alloc] init];

    [result addObjectsFromArray:[Cisla quickSort:smallerList]];
    [result addObjectsFromArray:equalList];
    [result addObjectsFromArray:[Cisla quickSort:biggerList]];

    return result;
}
```

Swift

```swift
func quicksort( list: [Číslo]) -> [Číslo] {
    var list = list
    if list.count <= 1 {
        return list
    }

    let pivot = list[0].hodnota

    var smallerList = [Číslo]()
    var equalList = [Číslo]()
    var biggerList = [Číslo]()

    for x in list {

        if x.hodnota < pivot {
            smallerList.append(x)
        } else if x.hodnota == pivot {
            equalList.append(x)
        } else if x.hodnota > pivot {
            biggerList.append(x)
        }
    }

    return quicksort(list: smallerList) + equalList + quicksort(list: biggerList)
}
```

[zdroj] Autor

Obrázek č. 32 Implementace algortimu Quick Sort napsané v různách jazycích

Po spuštění kódu je v tabulce č. 9 uveden čas trvání jednotlivých operací v milisekundách pro programovací jazyky Objective-C, Python 2.7 a Swift 4.0.3. Nejmenší hodnota je lepší.

	Inicializace 10 000 objektů (ms)	Seřazení 10 000 objektů (ms)	Celkový čas (ms)
Swift	5,31	115,77	121,53
Objective-C	2,07	36,17	38,33
Python	4,82	49,23	54,15

Tabulka č. 9 Výsledky výkonnostního skriptu

Překvapivě jazyk Swift se jeví jako nejpomalejší. I Python je rychlejší v inicializaci objektů než Swift. Při seřazení objektů je Swift více než dvakrát pomalejší než Python a třikrát pomalejší než Objective-C.

Toto zjištění vyvrací tvrzení, že Swift je výkonnější než ostatní jazyky. Vytvořený seřaďovací skript je však pouze jeden z mnoha testů, jak otestovat rychlost. Tento skript není běžný v programovacím pracovním postupu a Swift kompilátor si s optimalizací neumí poradit. Swift má mimo jiné vlastní algoritmy na seřazení objektů, které jsou optimalizované a rychlé. Skript byl ale napsaný stejně pro všechny jazyky a závěr je jasný, že Swift je při alokaci a vlastním seřaďovacím algoritmu nejpomalejší.

5 Zhodnocení výsledků a doporučení

Swift je skutečně jednoduchý a flexibilní jazyk a zároveň je komplexní a funkční. Swift je vhodný k výuce programování i vytváření komplexních aplikací s UI, podpůrných knihoven či jednoduchých programů pro příkazovou řádku. Na platformách od Apple má tento jazyk perspektivní budoucnost. Vývoj a podpora jazyka je konzistentní. Jazyk Swift má vytvořené stabilní základy a syntaxe je logická a snadno pochopitelná.

Zdrojový kód Swiftu je otevřený, stejně tak je otevřená komunikace s komunitou vývojářů Swiftu. Apple drží pod kontrolou hlavní vývojovou větev a rozhoduje, které změny do jazyka přidá. Licence umožňuje vytvořit si vlastní vývojovou verzi a implementovat vlastní změny do Swiftu podle svých potřeb bez právních či finančních restrikcí.

Funkce ve Swiftu prošly významnou inovací. Jazyk byl navržen tak, aby umožňoval objektově-orientované programování i nové funkční programování. Jiné jazyky, jako např. C# či Java, umožňují také programovat v souladu s funkčním paradigma, ale styl zápisu je velmi komplikovaný a nezvyklý. Ve Swiftu je funkční programování přirozené.

Swift plně nahrazuje Objective-C. S jazykem Swift lze bez jakýchkoli komplikací implementovat objektově-orientované návrhové vzory. Swift má i přístup k ukazatelům, tudíž i k operacím na nízké úrovni systému, ovšem pro programování (např. ovladačů) tento jazyk není vhodný. Swift je také skriptovací jazyk a lze jím vytvořit kód, který se interpretuje až při spuštění, což je zajímavé zjištění, neboť toto využití není oficiálně zdokumentované.

Výkonnostní test nepotvrdil, že by jazyk Swift byl rychlejší než Objective-C či Python. Výsledky tohoto testu ale nelze interpretovat tak, že je Swift pomalý ve všech aspektech běžícího procesu, neboť test zkoumal pouze jeho nepatrnou část. Nicméně to jasně demonstrovalo současný stav, který je dle mého názoru nevyhovující; výkon nebyl v posledních aktualizacích hlavní prioritou.

Swift znamená pro Apple dlouhodobou investici. Programovacím jazykům zabere velmi mnoho času, než proniknou do vývojářské obce. Apple podnikl veškeré kroky k tomu, aby

jazyk uspěl. Především vypracoval solidní základy a syntaxi, které mají konzistentní a jednotnou ideu. Navázal na Objective-C a zachoval jeho funkcionalitu, kterou rozšířil o nové vlastnosti z jiných jazyků. Od první verze došlo k přeprogramování celého API, aby bylo možné naprogramovat plnohodnotné produkty stejně jako s Objective-C. Xcode obsahuje nástroje, které umožňují přechod na novější verze Swiftu a kompilátor umí přeložit i starší verze Swiftu pomocí ABI. Vedle toho byla vydána aplikace *Playgrounds*, kde lze kód vyzkoušet. To vše dává pádné důvody pro rozhodnutí psát aplikace ve Swiftu.

Swift je také na platformě Linux, ovšem nemá žádné API či vývojové prostředí jako je Xcode na Macu. Swift na Linuxu obsahuje pouze kompilátor a standardní knihovnu. V současnosti zatím není rozšířený žádný projekt, který by využíval Swift na Linuxu.

Pro vývojáře, kteří vytvářejí projekty pro některou z Apple platforem, má určitě význam investovat svůj čas učením jazyka Swift. Je to plnohodnotný, jednoduchý a stabilní jazyk, který časem nahradí zastaralé Objective-C. Na Linux platformě je situace komplikovanější, neboť tam je potřeba začít od úplného začátku a vyžadovalo by to spoustu času vyvinout stabilní API. Čtyřletý vývoj Swiftu je však pro masivní adaptaci ještě krátká doba. Je však jen otázka času, kdy se jazyk Swift rozšíří i mimo Apple platformu.

6 Závěr

V této práci, věnované programovacímu jazyku Swift, byla na základě dostupné dokumentace, odborné literatury a odborných článků provedena klasifikace a charakteristika jazyka Swift, dále rozebrány jeho ideové základy, provedeno srovnání s jinými programovacími jazyky a nastíněn úvod do objektově-orientovaných návrhových vzorů.

Tyto informace pak navázaly na praktickou část. V ní byla provedena analýza zdrojového kódu z veřejně přístupného repozitáře. Na vlastním příkladu pak byla zhodnocena syntaxe jazyka Swift a prozkoumána architektura kompilátoru jazyka Swift. Na autorem vytvořených příkladech byly okomentovány a zhodnoceny funkce, návrhové vzory, ukazatele, řízení paměti a vytváření skriptů v jazyce Swift.

S jazykem Swift byly porovnány jazyky Objective-C a Python, které jsou výchozí podporované jazyky na operačním systému macOS. Došlo ke srovnání syntaxe, využití a výkonu. Pro změření výkonu všech jazyků byl vytvořen vlastní skript ve všech porovnávaných jazycích.

Poslední část práce formulovala výstup všech informací, pozorování a zjištěné výsledky, které byly doplněny o komentář autora. Nakonec bylo provedeno vyhodnocení současného stavu jazyka Swift a navrženo doporučení pro vhodné využití jazyka Swift.

Slovník pojmů

ABI – *Application Binary Interface*, zajišťuje aby za běhu byly všechny programy funkční.

API – *Application Programming Interface*, neboli aplikační programovací rozhraní umožňuje využít veškeré knihovny, třídy a funkce pro tvorbu programů.

AST – *Abstract Syntax Tree*, strom vazeb při překladu zdrojového kódu

Argument – Konkrétní hodnota daného parametru funkce.

Camel Case – způsob zápisu kódu, kde slova jsou bez mezer a každé nové slovo je velké. Rozlišují se na lowerCamelCase a UpperCamelCase podle počátečního písmena.

Commit – Zapsání změny do Git repozitáře.

Closures – Alternativa blokům kódů, či lambdám.

Funkce – Pojmenovaný blok kódu, který zpracovává data.

Framework – Aplikační rámec, který obsahuje podpůrné programy, knihovny API aj.

Garbage Collector – funkce, virtuálního stroje Javy, který řídí paměť.

Generičnost – Vlastnost, kdy není nutné definovat přesný datový typ.

GNU – Rekurzivní zkratak *GNU's Not Unix*. Je to označení pro svobodný operační systém.

GPL – *General Public Licence*. Licence pro svobodný systém.

IR – Instruction reference

Literál – Hodnota, která je určena pro zápis ve zdrojovém kódu.

Metadata – Data obsahující informace o vlastních datech

Metoda – Funkce, která je definované ve třídě

Nil – Prázdná hodnota

N-tice – Jedna proměnná, či konstanta, která udržuje více hodnot.

OOP – Objektově-orientované programování

Parametr – Proměnná, která vstupuje do funkce

Playgrounds – Aplikace pro testování Swift kódu.

Property – proměnná či konstanta definována ve třídě

Reference Counting – počítání referencí, které pomáhá s uvolňováním paměti.

Runtime – za běhu systému.

Shebang – ozančení cesty k souboru, kterým je skript interpretován

SIL – *Swift Intermediate Language*, slouží pro analýzu a optimalizaci Swift kódu.

SDK – *Software Development Kit*, neboli softwarový vývojový balíček

UI – *User Interface*, uživatelské rozhraní

Seznam literatury

[1] APPLE, *The Swift Programming Language,* [online]. [cit. 16-08-2016]. <https://swift.org/documentation/TheSwiftProgrammingLanguage(Swift3).epub >.

[2] Peter Van ROY, Programming Paradigms for Dummies: What Every Programmer Should Know, [online]. [cit. 17-08-2016]. <https://www.info.ucl.ac.be/~pvr/VanRoyChapter.pdf >.

[3] APPLE, *About Swift*. [online]. [cit. 20-08-2016]. <https://swift.org/about/#swiftorg-and-open-source>.

[4] Paul HUDSON, *What's new in Swift 2*. [online]. [cit. 21-08-2016]. <https://www.hackingwithswift.com/swift2>.

[5] Luca CARDELLI, Peter WEGNER, *On Understanding Types, Data Abstraction, and Polymorphism*, Computing Surveys, Vol 17 n. 4, pp 471-522, December 1985, [online]., [cit. 26-08-2016]. <http://lucacardelli.name/Papers/OnUnderstanding.A4.pdf >.

[6] Ray TOAL, *Classifying Programming Languages*. Loyola Marymount University, Los Angeles. [online]. [cit. 23-12-2016]. <http://cs.lmu.edu/~ray/notes/pltypes/>.

[7] Ray TOAL, *Programming Paradigms*. Loyola Marymount University, Los Angeles. [online]. [cit. 23-12-2016]. <http://cs.lmu.edu/~ray/notes/paradigms/>.

[8] Kurt NØRMARK, *Functional Programming in Scheme*. Department of Computer Science, Aaborg University, Denmark, September 2003, [online]. [cit. 27-12-2016]. <http://people.cs.aau.dk/~normark/prog3-03/pdf/all.pdf>.

[9] Rudolf PECINOVSKÝ, *Návrhové vzory*, Computer Press, Brno 2007, vydání první, 528 s., ISBN: 978-80-251-1582-4.

[10] Lars Ole ANDERSEN, *Program Analysis and Specialization for the C Programming Language*. DIKU, University of Copenhagen, 1994. [online]. [cit. 24-1-2017]. <http://www-ti.informatik.uni-tuebingen.de/~behrend/PaperSeminar/Program%20Analysis%20and%20SpecializationPhD.pdf >.

[11] Bjarne STROUSTRUP, *Evolving a language in and for the real world: C++ 1991-2006*. Texas A&M University. 2006. [online]. [cit. 24-1-2017]. <http://stroustrup.com/hopl-almost-final.pdf>.

[12] Charles WALLACE, *The Semantic of the C++ Programming Language*. The Pennsylvania State University, 1993. [online]. [cit. 24-1-2017]. <http://cteseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.35.6692&rep=rep1&type=pdf>.

[13] Jiva DeVOE, *Objective-C Developer Reference.* Indianapolis, Wiley Publishing, Inc, 2011, 382s., ISBN: 978-0-470-47922-3

[14] APPLE, *Why Objective-C.* 2010. [online]. [cit. 26-01-2017]. < https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/OOP_ObjC/Articles/ooWhy.html >.

[15] ORACLE, *The History of Java Technology*. [online]. [cit. 26-01-2017]. <http://www.oracle.com/technetwork/java/javase/overview/javahistory-index-198355.html >.

[16] ORACLE, The Java Language Environment. 1997. [online]. [cit. 27-01-2017]. <http://www.oracle.com/technetwork/java/intro-141325.html >.

[17] Michael H. GOLDWASSER, David LETSCHER, *Object-Oriented Programming in Python.* New Jersey, 2008 Pearson Education, Inc. 666s. ISBN: 978-0-13-615031-2

[18] Tim PETERS, *Zen of Python*. PEP Index, 2004. [online]. <https://www.python.org/dev/peps/pep-0020/ >.

[19] Bill VENNERS, The Making of Python, A Conversation with Guido van Rossum.

[20] Erich GAMMA, Richard HELM, Ralph JOHNSON, John VLISSIDES, *Návrh programů pomocí vzorů – Stavební kameny objektově orientovaných programů.* Praha : Grada, 2003. 386s. ISBN: 80-247-0302-5

[21] APPLE, *Platforms State of the Union*, WWDC 2017 – Session 102. [záznam z konference]. [online]. [cit. 23-09-2017]. <https://developer.apple.com/videos/play/wwdc2017/102/ >.

[22] Mikey CAMPBELL, *Former Apple executive Chris Lattner leaves Tesla after 6 months on the job*. AppleInsider.com. 2017-06-20. [online]. [cit. 2017-11-09]. <http://appleinsider.com/articles/17/06/21/former-apple-executive-chris-lattner-leaves-tesla-after-6-months-on-the-job>.

[23] Apple, *Using Swift with Cocoa and Objective-C*. 2014-06-02. [online]. [cit. 2-10-2017]. < https://itunes.apple.com/cz/book/using-swift-with-cocoa-and-objective-c-swift-4/id888894773?l=cs&mt=11>.

[24] SWIFT.ORG, *API Design Guidelines*. [online]. [cit. 2017-10-30]. <https://swift.org/documentation/api-design-guidelines/>.

[25] Maxime DEFAUW, *ARC and Memory Managment in Swift*. raywenderlich.com 2016-09-16. [online]. [cit. 2017-11-1]. <https://www.raywenderlich.com/134411/arc-memory-management-swift >.

[26] Scott WLASCHIN, *Functional Programming Design Patterns.* NDC London 2014. [online]. [cit. 2017-11-6]. <https://fsharpforfunandprofit.com/fppatterns/>.

[27] Neal FORD, *Functional thinking*. 2011-08-30. [online]. [cit. 2017-10-25]. < https://www.ibm.com/developerworks/java/library/j-ft5/index.html >.

[28] ANSI, Rationale for the ANSI C Programming Language. Sillicon Press. [online]. < http://www.open-std.org/jtc1/sc22/wg14/www/C99RationaleV5.10.pdf >.