Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Diplomová práce

Vývoj webového informačního systému pro výzkum myší

Bc. Zdeněk Jirásek

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ DIPLOMOVÉ PRÁCE

Bc. Zdeněk Jirásek

Informatika

Název práce

Vývoj webového informačního systému pro výzkum myší

Název anglicky

Development of web information system for mouse research

Cíle práce

Cílem diplomové práce je návrh, vývoj a následná implementace webového informačního systému pro evidenci a výzkum myší.

Dílčím cílem je zpracování literární rešerše v oblasti dané problematiky, která bude sloužit jako teoretický podklad při řešení hlavního cíle.

Metodika

Při zpracování teoretické a praktické části budou použity převážně anglické literární zdroje. Teoretická část bude také čerpat z aktuálních dokumentací od tvůrců použitých technologií.

Informační systém bude ve formě webové aplikace, která bude vyvinuta v programovacím jazyku Python, konkrétně jeho frameworku, založeném na architektuře Model-view-controller, Django.

Framework budou doprovázet webové technologie HTML, CSS a Javascript. Databáze bude zvolena na základě analýzy požadavků kladených na systém. Vývoj bude probíhat v jednotlivých cyklech dle agilní metodiky TTD (Test-Driven Development).

Nasazení bude provedeno formou kontejnerizace v nástroji Podman.

Doporučený rozsah práce

60-80 stran

Klíčová slova

python, webový informační systém, Django framework, výzkum, HTML, CSS, Javascript

---

Doporučené zdroje informací

DUCKETT, J. HTML and CSS: Design and Build Websites. Indianapolis: John Wiley & Sons, 2011. ISBN 978-1-118-00818-8.

KLEPPMANN, M. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. Sebastopol: O'Reilly Media, 2017. ISBN 978-1-449-37332-0

MARTIN, R. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Londýn: Prentice Hall, 2018. ISBN: 978-0-13-449416-6

PERCIVAL, H. Test-Driven Development with Python: Obey the Testing Goat: Using Django, Selenium, and JavaScript, 2nd Edition. Sebastopol: O'Reilly Media, 2017. ISBN 978-1-491-95870-4

Předběžný termín obhajoby

2023/24 LS – PEF

Vedoucí práce

Ing. Jiří Brožek, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 28. 11. 2023

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 9. 2. 2024

doc. Ing. Tomáš Šubrt, Ph.D.

Děkan

V Praze dne 30. 03. 2024

Čestné prohlášení

Prohlašuji, že svou diplomovou práci "Vývoj webového informačního systému pro výzkum myší" 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 použitých zdrojů 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 31. března 2024 _____

Poděkování

Rád bych touto cestou poděkoval svému vedoucímu doktorovi Jiřímu Brožkovi za rychlou a cenou zpětnou vazbu při konzultacích. Dále své sestře Markétě za podporu během celého studia. Vděčnost si také zaslouží má nejdražší přítelkyně Ania. Bez její emotivní podpory by dokončování této práce bylo mnohem obtížnější.

Vývoj webového informačního systému pro výzkum myší

Abstrakt

Moje diplomová práce se zabývá vývojem informačních systémů. Hlavním cílem mé práce je vývoj a implementace laboratorního systému pro správu a výzkum myší s využitím webových technologií. Konkrétně webovým frameworkem Django založeným na jazyku python. Dílčím cílem, který by měl podpořit hlavní cíl, je vytvoření srozumitelné literární rešerše o informačních systémech a metodách jeho vývoje. V praktické části mé práce zaměřené na analýzu systému je zvolena databáze PostgreSQL, a to z důvodu nejlepší podpory daného webového frameworku. Při vývoji se ukázalo jako velmi užitečné šablonování HTML s podporou modulů JavaScriptu. Přestože se zvolená metoda testově orientovaného vývoje ukázala jako užitečná při tvorbě kvalitnějšího kódu, zvýšila však složitost vývoje. V důsledku toho musely být některé požadavky během vývoje zrušeny. Jako například modul statistiky, který by umožnil správcům větší grafický přehled o tom, co se v laboratoři děje. Závěrem lze říci, že vyvinutý systém by mohl být používán v laboratoři. Složitější funkce, jako je zmíněný modul statistiky, by mohly být snadno přidány s opětovným využitím již vytvořených návrhových vzorů.

Klíčová slova: python, webový informační systém, Django framework, výzkum, HTML, CSS, Javascript

Development of web information system for mouse research

Abstract

My thesis is about developing information systems. Main goal of my thesis is developing and implementing a mouse research laboratory system using web technologies. Specifically, python-based web framework Django. The sub-objective, which should support the main goal, of my thesis is making a comprehensible study into information systems theory and methods of development. In practical systems analysis part of my thesis a PostgreSQL database is chosen, because of best support for given web framework. The HTML templating with support for JavaScript modules was proven very useful while development. Although chosen development method test-driven development proved useful in producing code of higher quality it increased complexity to develop. As a result, some requirements had to be scrapped during development. Like statistics module, which would enable more graphical insight to administrators into what's going on in the lab. In conclusion the developed system could be used in a lab. More complex functionality like afford mentioned statistics module could be added easily reusing already created design patterns.

Keywords: python, web information system, Django framework, research, HTML, CSS, Javascript

Obsah

1 Úvod

Informační systém je celek složený z počítačového hardwaru a softwaru, který slouží k podporování nějakého uceleného cíle. Na teoretické rovině je systém definován daty na svém vstupu, informacemi a způsobu jejich zpracování na svém výstupu. V organizacích se informační systémy používají k podporování nějaké jeho agendy. Například účetnictví, sklady či majetek a tak podobně.

Postupem času, jak se informační technologie vyvíjí jsou více náročná na data, než na výpočetní výkon. Mezi důležité vlastnosti datově náročných informačních systémů patří spolehlivost, udržovatelnost a škálovatelnost. Díky spolehlivosti je systém funkční i při střetu s nepříznivými elementy. Zatímco udržovatelnost je schopnost systému adaptovat se na změny. V poslední řadě škálovatelnost řeší vypořádávání se s postupným růstem systému a jeho užívání.

Vývoj informačních systémů je komplexní proces, který se skládá z mnoha činností. Projekty v IT odvětví, jako je vývoj informačních systémů, se stejně jako v každém jiném dají řídit metodami projektového managementu. Dle trojimperativu se každý projekt řídí s ohledem na 3 omezení – rozsah, čas a náklady.

Vývoj není pouze o mechanickém převodu návrhů do jazyku počítačů, ale i kreativitě a úsudku. I když neformálně se vývoj z velké většiny týká pouze programování neboli konstrukce. V podnikové sféře se na celém životním cyklu informačního systému může podílet velké množství pracovních rolí. Životní cyklus informačního systému rozděluje vývoj na logické etapy. Typicky se jedná o 6 etap: analýzu, design, vývoj, testování, implementaci a údržbu. Vývojové procesní modely definují jednotlivé fáze a jejich pořadí.

2 Cíl práce a metodika

2.1 Cíl práce

Cílem diplomové práce je návrh, vývoj a následná implementace webového informačního systému pro evidenci a výzkum myší.

Dílčím cílem je zpracování literární rešerše v oblasti dané problematiky, která bude sloužit jako teoretický podklad při řešení hlavního cíle.

2.2 Metodika

Při zpracování teoretické a praktické části budou použity převážně anglické literární zdroje. Teoretická část bude také čerpat z aktuálních dokumentací od tvůrců použitých technologií.

Informační systém bude ve formě webové aplikace, která bude vyvinuta v programovacím jazyku Python, konkrétně jeho frameworku založeném na architektuře Model-view-controller, Django.

Framework budou doprovázet webové technologie HTML, CSS a Javascript. Databáze bude zvolena na základě analýzy požadavků kladených na systém. Vývoj bude probíhat v jednotlivých cyklech dle agilní metodiky TTD (Test-Driven Development).

Nasazení bude provedeno formou kontejnerizace v nástroji Podman.

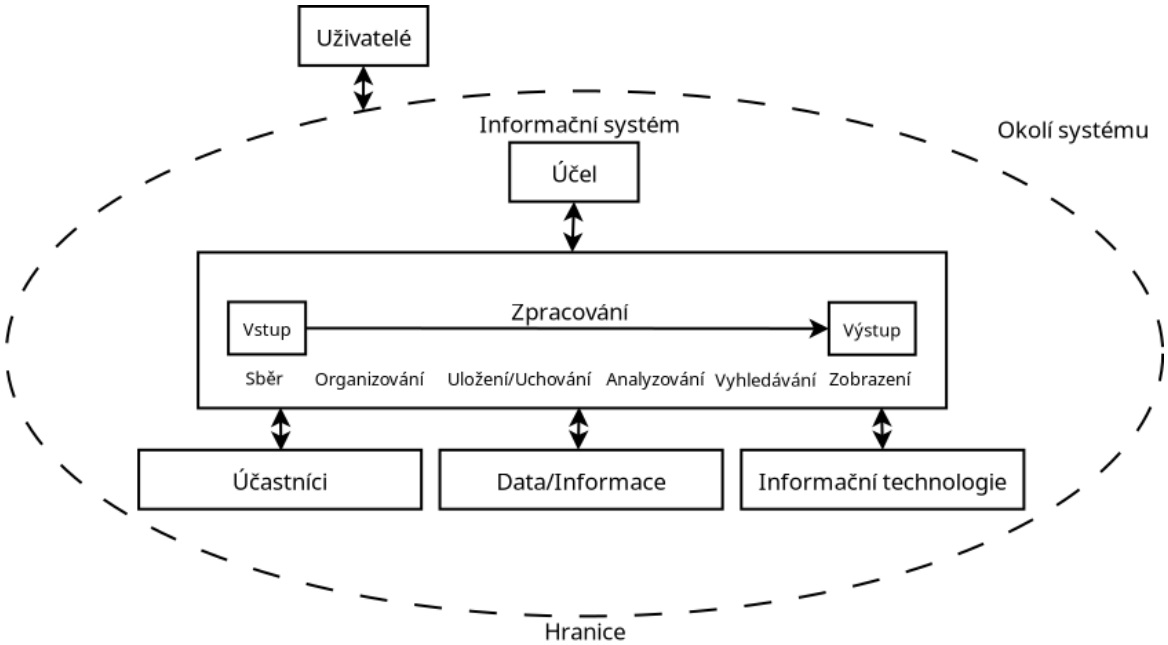
3 Teoretická východiska

3.1 Informační systémy

„Informačním systémem obecně nazýváme organizaci údajů vhodnou pro systémové zpracování dat: pro jejich sběr, uložení a uchování, zpracování, vyhledávání a vydávání informací o nich, to vše pro rozhodování v běžné praxi.“ [1, s. 8]

Výše uvedená definice od Šarmanové bere informační systémy spíše ze širšího pohledu. Autoři se na přesnější definici v užším slova smyslu neshodují. Každý upřednostní takovou vlastnost informačních systémů, která je dle jeho pochopení celé oblasti významná. Ke skutečnému pochopení smyslu informačních systémů je třeba porozumět úzce souvisejícím pojmům:

- Data – množina nezpracovaných hodnot, které popisují stav objektu v určitý moment
- Informace – množina zpracovaných hodnot vycházejících z dat
- Systém – ucelený soubor prvků se vzájemnými vazbami
- Model – obraz popisující systém (dle pohledu autora) [2]



Obrázek č. 1: Schéma informačního systému a jeho komponent (vlastní zpracování dle zdroje) [3]

Informační systémy nepracují s reprezentací systému, nýbrž s jeho modelem. Ačkoliv model popisuje charakteristiky systému, nikdy nebude schopen být jeho absolutní kopií. V každém systémů figuruje určitá úroveň ohraničenosti, kterou si lze představit téměř jakkoliv. [2]

Například je logické, že systém objednávek v kavárně by nebral v potaz baristu vykonávající danou objednávku. Přesto zahájení evidence této informace může umožnit měřit výkony zaměstnanců, podle kterého se následně vyplácí prémie za nadstandardní práci. Otázkou pak je, zda by se po implementaci této funkce nadále jednalo o objednávací systém či nikoliv. Z jiného pohledu by na vykonání této funkce měly spolupracovat dva systémy: docházkový a objednávkový. Na tyto a příbuzné otázky by měla odpovídat pevně stanovená hranice systémů.

V poslední době se stává, že mnoho informačních systémů (aplikací) je náročných spíše na data než na výpočetní techniku. Za limitující faktor systémů se už zřídka považuje výpočetní výkon jako rychlost centrální procesní jednotky (CPU). Větší problém je množství, složitost a rychlost zpracování dat. [4]

„Datově náročná aplikace je obvykle tvořena ze standartních stavebních bloků, které poskytují běžně potřebné funkce. Například mnoho aplikací potřebuje:

- ukládat data, aby šla v daném nebo jiném systému najít později (databáze)
- pamatovat si výsledek náročné operace, aby se urychlilo čtení (mezipaměť)
- umožnit uživatelům hledat data podle klíčového slova či filtru různými způsoby (indexace)
- odesílat zprávu jinému procesu, aby je zpracoval asynchronně (proudové zpracování)
- periodicky zpracovávat velké množství nahromaděných dat (dávkové zpracování)“ [4, s. 3]

Způsobů, jakými lze splnit výše uvedené funkcionality, je mnoho, protože existuje spoustu typů technologií, každá se svým specifickým případem užití. Nicméně je efektivní některé požadavky na systém splnit spojením vícero technologií. Samotná spolupráce těchto technologií je následně definována prostřednictvím aplikačního kódu. [4]

Při designu a následném vývoji informačních systémů se zabýváme 3 základními vlastnostmi. Tyto vlastnosti jsou spolehlivost, škálovatelnost a udržovatelnost. Spolehlivost zajišťuje nepřetržitou správnou funkčnost, a to i v případě ovlivnění nepříznivými elementy. Chování systému při rozšiřování nejen v komplexnosti, ale také v množství dat a provozu, je řešena úrovní škálovatelnosti. Kdežto udržovatelnost zajišťuje stálé chování a schopnost systému se přizpůsobit novým požadavkům v průběhu času. Tyto důležité vlastnosti informačních systémů budou probrány více do detailu v následujících podkapitolách. [4]

3.1.1 Spolehlivost

Pod pojmem spolehlivost si každý může představit něco jiného, nicméně ani při spojení se softwarem není ucelená představa. Například běžný uživatel systému by řekl: „vše funguje, jak má“. Naopak softwarový inženýr se může soustředit na specifičtější proměnné jako rychlost odpovědí databáze pod zátěží a tak podobně.

Od spolehlivého softwaru se očekává, že bude splňovat požadavky uživatele. Zároveň počítá s možností chyby na uživatelově straně včetně používání systému všemi možnými (špatnými) způsoby. Dále, výkon by měl být adekvátní specifickému případu užití s předpokládanou zátěží a objemem dat. V poslední řadě se očekává, že je odolný vůči zneužití a umožní přístup pouze oprávněným entitám. [4]

Pokud uvedené vlastnosti systém splňuje i přes výskyt závady, kterou dokáže předvídat a řešit, nazýváme ho odolným proti selhání. I když teoreticky není možné vyvinout software, který je připravený na výskyt každého druhu závady. [4]

I přes výskyt závady by software, který je spolehlivý, neměl činit fyzickou nebo ekonomickou škodu. Celková spolehlivost systému závisí na spolehlivosti jeho komponent. Komponenty z pohledu spolehlivosti jsou hardware, software a obsluha. Mezi spolehlivostí jednotlivých komponent existuje závislost. [5]

Závada jedné komponenty může ovlivnit činnost ostatních komponent a tím snížit celkovou spolehlivost systému. Jak závady jedné části systému ovlivní ostatní komponenty je těžké odhadnout. Celkovou spolehlivost systému je tím pádem z dat o stabilitě jednotlivých komponent téměř nemožné odhadnout. [5]

Počáteční řešitelná závada může postupem času přerůst v kritickou, což může vést až ke zkolabování celého systému. Například závada na hardwarové straně bude odesílat nesprávné vstupy, na které následně softwarová vrstva nebude dostatečně připravena, a tak se zachová abnormálně a vygeneruje špatné výstupy. Výstupy poté zpozoruje obsluha, ve které to evokuje stres, tím se zvyšuje pravděpodobnost, že se závada bude dále rozvíjet. [5]

„Lidé navrhují a vyvíjí softwarové systémy a obsluha, která tyto systémy udržuje v chodu, jsou také lidé. I když mají ty nejlepší úmysly, lidé jsou nespolehliví.“ [4, s. 9]

Autor výše uvedené citace Kleppmann na důkaz svého tvrzení uvádí studii v oblasti internetových služeb, ve které chyby na straně obsluhy způsobily většinu výpadků. Kdežto hardwarové potíže jako servery či síť pouze z 10-25 %. [4, 6]

Od prostředí, ve kterém je systém používán, se odvíjí i jeho spolehlivost. Vybrat okolí, ve kterém je systém maximálně spolehlivý, není možné, protože prostředí nelze s určitostí stanovit. Je obzvlášť obtížné klást omezení na prostředí provozních systémů. Ve stejném prostředí se může každý systém neočekávatelně chovat při střetu s problémy, čímž může ovlivnit všechny ostatní související systémy. [5]

3.1.2 Udržovatelnost

Systémy musí být udržovány, což znamená, že by měly být schopny se adaptovat na změny v čase. Velká část rozpočtu na software jde právě do průběžného udržování. Pod udržováním informačního systému si představíme úkony jako opravy a zkoumání příčiny vzniklých chyb, přizpůsobování novým prostředím či přidávání nových funkcionalit. [4]

Postupem času, co je software používán, vznikají nové požadavky. Aby si software zachoval svou užitečnost, měl by se těmto požadavkům přizpůsobit. V případě, že se jedná o software, který je udržovaný, nemělo by zavedení změn vyvolat nové chyby. [5]

Jednou ze zásad při vývoji softwaru je vytvořit ho takovým způsobem, aby se z něj eventuálně nestal tzv. historický systém. Historické systémy se stále používají v některých organizacích, protože nadále splňují požadované funkce. Nicméně jejich modifikace je obtížná kvůli zastaralé platformě, čistotě kódu a nedostatečné dokumentaci. Tím pádem je i jejich udržovatelnost značně limitovaná, a proto s nimi vývojáři neradi pracují. [4]

3.1.3 Škálovatelnost

Škálovatelnost vyjadřuje funkčnost systému z hlediska zátěže. Neboli to, jak dobře systém zvládá zvýšenou náročnost jeho používání. Otázka často nebývá, jak systém funguje právě teď, ale jak systém bude pracovat v budoucnu. Zátěž se přirozeně zvyšuje postupným růstem systému. Například počtem uživatelů nebo objemem uložených dat. [4]

Zátěž by měla být popsána pomocí přesně určených parametrů. Zvolené parametry by se pro nejlepší výsledek měly vybírat dle použité systémové architektury. Například u webových systémů odpovědí za sekundu či počtem zpracovaných operací u účetního softwaru. Bez specifického popsání zátěže a jeho metrik, je obtížné efektivně sledovat výkon systému. [4]

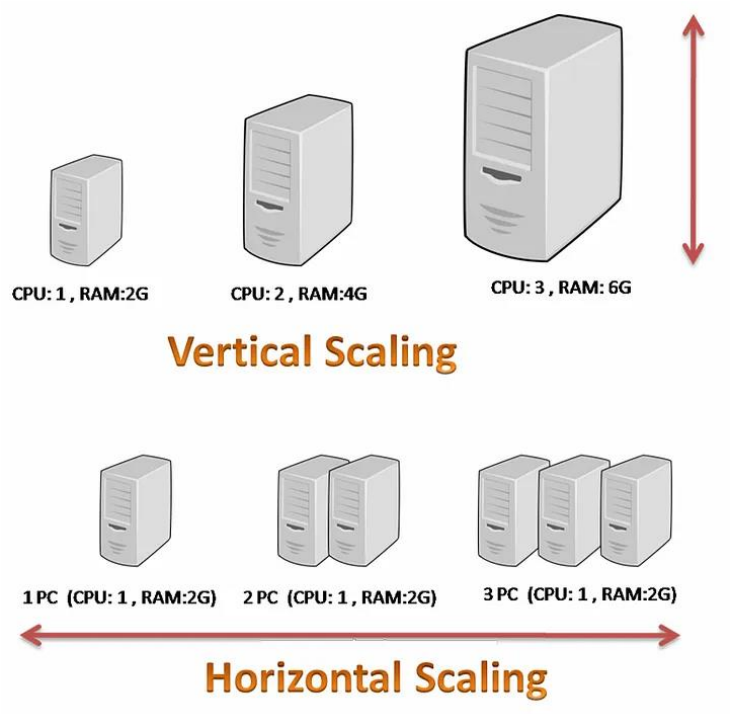
Důležité je zachování minimálně takového výkonu, aby odpovídal aktuálním požadavkům kladených na systém. Avšak, stejně jako u spolehlivosti, je obtížné určit adekvátní výkon už při navrhování. Z tohoto důvodu jsou tyto požadavky zjistitelné až při uvedení systému do nějaké formy provozu. [5]

Rozdělením systému na velké množství malých komponent se zlepšuje udržovatelnost, ale snižuje výkon. Naopak použitím velkých komponent zvyšujeme výkon, ale snižujeme udržovatelnost. Pokud požadavku na výkon a udržovatelnosti systému dáváme stejnou váhu, je potřeba naleznout kompromis. [5]

Abychom se zbavili nadměrné zátěže a udrželi adekvátní výkon systému musíme navýšit zdroje. Avšak, pokud aplikační kód není dostatečně optimalizován, může využívat hardwarových zdrojů až přebytečně, i když to není nutně potřebné. Ideálně by tedy systém měl hardwarové zdroje efektivně využívat a zbytečně s nimi neplýtvat. [5]

V běžné praxi se používají dva způsoby škálování informačních systémů:

- Horizontální – přidávání fyzických či virtuálních aplikačních instancí, mezi které se zátěž rozděluje
- Vertikální – přidávání hardwarových zdrojů:
    - Výpočetní – nejdůležitější zdroj, který provádí operace v jednotlivých instrukčních cyklech centrální procesní jednotky
    - Úložné – zařízení jako HDD či rychlejší varianta SSD uchovávají data, kdežto operační paměť RAM spolupracuje s daty, která jsou aktivně využívána procesorem
    - Síťové – používány v komunikaci s ostatními prvky v síti, při odesílání velkého množství dat může maximální kapacita přenosu představovat úzké hrdlo. [7]



Obrázek č. 2: Vertikální a horizontální škálování [8]

3.2 Projektování IS

Projekt je množina souvisejících aktivit, které mají společný cíl. Zavádění informačního systému se dá označit za projekt, který je převážně technického rázu. Organizace musí před schválením projektu zvážit nejenom výhody splnění a cenu, ale i případné technické potíže, které může v průběhu vyvolat. Od organizace ve které je informační systém implementován vyžaduje určitou změnu. Teoreticky se za projekt dají považovat i všední věci jako například úklid pokoje či stěhování. [9]

IT projekty se vyznačují tím, že používají hardware, software a počítačové sítě k vytvoření unikátní služby nebo produktu. Vytvořením služby, produktu a případným finálním zhodnocením projekt končí. O nepřetržitý chod těchto služeb či produktů se dále stará oddělení IT provozu. V poslední době je kladen důraz na spolupráci týmů vývoje a provozu, aby byl zachován co nejplynulejší přechod z vývojového prostředí do ostrého. [10]

„DevOps je poměrně nový termín, který je používán k popisu kultury spolupráce týmů softwarového vývoje a provozu, aby byl vytvořen, otestován a rychleji vydán spolehlivý software“ [11, s. 4]

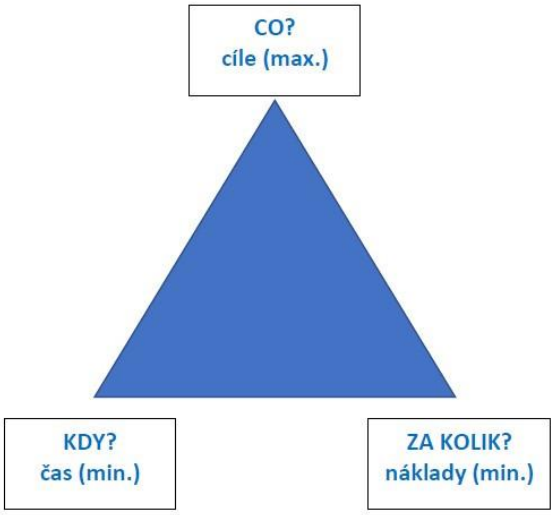
IT projekty jsou například:

- „Vysoká škola zdokonalí technologickou infrastrukturu, aby poskytovala bezdrátový internet po celém kampusu s online přístupem k akademickým a studentským službám
- Celosvětová banka odkoupí jiné finanční instituce a potřebuje sjednotit systémy a postupy
- Skupina studentů vyvine aplikaci na chytré telefony a prodává jí online“ [11, s. 5]

Projekt může být malého či velkého rozsahu. Vyhotovení může zabrat pár dní nebo trvat nepřetržitě roky. Účastnit se ho mohou týmy dvaceti lidí nebo pouze jeden klíčový člověk. Vše se odvíjí od vlastností projektu a rozhodnutí projektového manažera. [11]

3.2.1 Trojimperativ

Obecně jsou všechny projekty řízeny s ohledem na 3 omezení – rozsah, čas a náklady. V oboru projektového managementu se můžeme setkat s pojmem trojimperativ (v aj. the triple constraint nebo iron triangle). Model trojimperativu vyjadřuje vzájemnou propojenost a závislost těchto 3 omezení. Pro úspěšný projekt by teoreticky měly být tyto tři vlastnosti vyváženy. [12]



Obrázek č. 3: Trojimperativ projektu [13]

Je logické, že pozdě dokončený projekt s překročeným rozpočtem a jiným rozsahem, než bylo naplánováno, je považován za neúspěch. Z toho vyplívá, že řízení projektu je úzce spjato s trojimperativem. S touto myšlenkou souhlasí experti, kteří nazývají koncept vyvažování mezi rozsahem, časem a náklady jako jeden z nejdůležitějších v historii projektového managementu. Ačkoliv ne každý projekt s překročeným rozpočtem a dlouhým zpožděním je předurčen ke zhroucení. [12]

Budova opery v Australském městě Sydney je známým příkladem. Počátek stavby byl v roce 1959 s odhadovanou dobou trvání 4 let a rozpočtem 7 milionů australských dolarů. Nakonec byla slavnostně otevřena královnou Alžbětou II až v roce 1973 s reálnými výdaji 102 milionů australských dolarů. Z pohledu projektového managementu by se dalo jednat o naprosto neúspěšný stavební projekt. Přesto o 2 roky později v roce 1975 samotná budova vyrovnala rozpočty a je dnes celosvětově uznávanou památkou. [12]

I když popis důležitých aspektů trojimperativu je v oboru projektového managementu považován za významný, má své limitace. Jeho pohled je zaměřen dovnitř, a tím nezohledňuje ostatní prvky potřebné k úspěchu. Projektoví manažeři musí v této době také brát v potaz i vnější prvky. Jako je přínosnost a zapojení účastníků projektu. [12]

3.2.2 Projektový management

„Projektový management je aplikace znalostí, dovedností, nástrojů a technik na projektové činnosti s cílem splnit požadavky projektu.“ [14, s. 9]. Projektový manažer (někdy hovorově označován jako projekťák) zodpovídá za plánování, organizování, řízení a kontrolu realizace projektu. [10]

V poslední době vzrostl zájem o projektový management u mnoha lidí a organizací. Projektový management se v 80 letech převážně zaměřoval na předávání informací o zdrojích a plánování vrcholovému vedení. Nicméně již v této době byl používán v IT odvětví. Dnes je situace již jiná a prostřednictvím projektového managementu jsou vedeny projekty v téměř každém odvětví všude na zemi. [11]

Kvůli možnosti vedení projektů v předvídatelné podobě vznikla množina procesů. Tyto množiny se nazývají metody projektového řízení. Metody projektového řízení se dají použít na všelijaké druhy projektů. Přičemž metody vývoje jsou použitelné na specifické projekty s konkrétními typy výstupů. Důvodem je různorodost vývojových úkolů v závislosti na cílech projektu. [10]

Například vývoj bankovního softwaru a modernizace klientských stanic se dají považovat za projekty. Na oba projekty je aplikovatelný odlišný způsob vývoje, ale oba se dají vést stejnou metodou projektového řízení. Známou metodou projektového řízení ve Velké Británii, která se hodí k vývojovým metodám, je PRINCE2. Úspěch projektu netkví pouze v dodržování určité metodologie. Avšak důkladné uplatnění těchto metodik může poskytnout prostředky k úspěšnému projektu. [11]

3.2.3 Životní cyklus projektů

Projekt od jeho zárodku až k jeho konci prochází řadou fází. Tyto fáze a jejich popis se týkají každého projektu, nehledě na jeho konkrétní zaměření. Jedna nebo více cyklů projektu souvisí s vývojem produktu, služby nebo výsledku. Fáze projektu mohou být prediktivní nebo adaptivní. Prediktivní znamená, že rozsah, čas a zdroje jsou určeny v raných fázích projektu. Kdežto adaptivní stanovují rozsah v každém cyklu. [14]

Obecně se životní cyklus projektu rozděluje na tyto čtyři fáze:

- „Zahájení projektu
- Organizace a příprava
- Provádění prací
- Dokončení projektu“ [11]

Míra nejistoty je nejvyšší v počátečních fázích projektu. Naopak požadavky na zdroje jsou v raných fázích nejnižší. Ve střední fázi se nejistota snižuje díky přesnějším informacím o požadavcích. Na konci projektu je kladen důraz na přesné splnění požadavků a schválení projektu sponzory. O změnách vize produktu, služby nebo výsledků by mělo být rozhodnuto co nejdříve, protože změny v pozdějších fázích jsou více nákladné. [11]

3.3 Vývoj IS

Vývoj informačních systémů je komplexní proces od začátku až do konce. Skládá se z více činností. Předpokladem pro úspěšný výsledek vývoje je dobré plánování, monitorování změn a efektivní řízení. Cílem vývoje softwaru v podnicích je dosáhnutí obchodního cíle za určité náklady v určitém časovém limitu. [9, 10]

Navzdory veřejným míněním vývoj není pouze o mechanickém převodu návrhů do jazyku počítačů, ale i o kreativitě a úsudku. Je možné, že programátoři samouci, kteří ještě neměli tu příležitost se podílet na oficiálním vývoji informačních systémů, se nikdy s formálním rozdělením níže nesetkali a místo toho by množinu těchto činností shrnuli pod jedním pojmem „programování“. [9]

Dle McConnella za posledních 25 let vědci definovali tyto činnosti související s vývojem:

- „Definice problému
- Vývoj požadavků
- Plánování konstrukce
- Architektura softwaru nebo návrh z vyššího pohledu
- Detailní návrh
- Kódování a ladění
- Testování jednotek
- Integrační testování
- Integrace
- Testování systému
- Opravná údržba“ [9, s. 3]

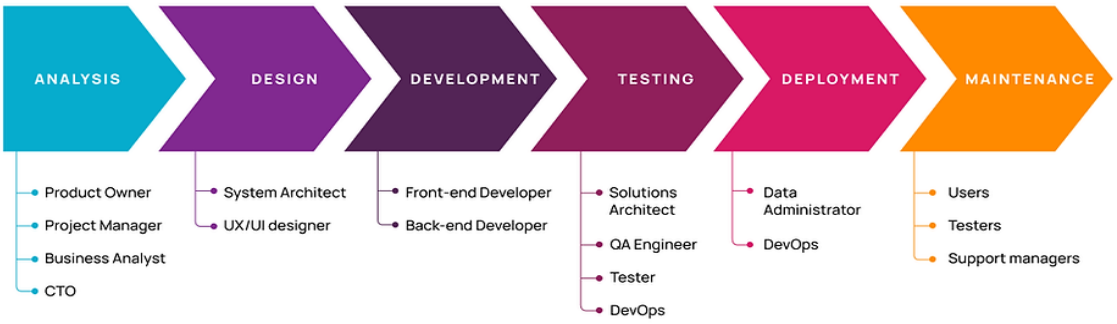
Konstrukce je jedna z hlavních činností, která je vykonávána při vývoji softwaru. Samotný termín „konstrukce“ obecně vyjadřuje proces budování. Například budování stavby dělníky. Plánování, navrhování a kontrola provedené práce může být součástí konstrukčního procesu. Ovšem „konstrukce“ se většinou týká vytváření něčeho v praxi. [9]

Kódování a ladění jsou hlavní dílčí činnosti konstrukce. Dále konstrukce úzce souvisí s detailním návrhem, plánování konstrukce, testování jednotek, integrací a integračním testováním. V reálném světě se mnoho fází vývojového procesu vynechá. Avšak bez konstrukce se vývoj nedokáže obejít. [9]

Pro efektivní konstrukci by měly být před začátkem programování tvořeny požadavky a architektura. Po naprogramování je správně provedená konstrukce ověřena systémovým testováním. Konstrukce může podle velikosti projektu zabrat od 30 do 80 procent celkového času vývoje. [9]

3.4 Životní cyklus IS

Vývoj systémů je popisován životním cyklem informačních systémů – také známý pod zkratkou SDLC (z aj. System Development Life Cycle). Díky SDLC je možné informační systémy logicky a metodicky navrhovat, vyvíjet a implementovat. Tento rámec je široce používaný mezi organizacemi. Každá společnost může mít svůj vlastní způsob pojmenování a rozdělení etap. [10]



Obrázek č. 4: Životní cyklus informačního systému. [15]

Na obrázku výše je vidět typické rozdělení SDLC na 6 fází: analýzu, design, vývoj, testování, implementaci a údržbu. Pod jednotlivými etapy jsou pracovní pozice, které se na této činnosti podílejí v organizacích. Pozicí je celkem 16. Pozice testera je využívána ve dvou fázích: testování a údržba. Z takového počtu je možné si odvodit komplexnost celého vývojového procesu v oblasti podnikání. Hughes v porovnání rozděluje etapy SDLC následovně:

- „Inicializace
- Identifikace obchodního případu
- Zahájení projektu
- Sběr a analýza požadavků
- Návrh
- Vývoj
- Akceptační testování
- Implementace/instalace
- Hodnocení a údržba“ [10, s. 6-7]

Může se zdát, že fáze vývojového procesu jsou vykonávány postupně. Nicméně opak je pravdou, protože jednotlivé komponenty vyvíjeného informačního systému se mohou nacházet v různých stádiích. Přičemž v každém stádiu jsou tvořeny hmatatelné výsledky. Výsledky poté mohou sloužit ke stanovení milníků, dle kterých lze posoudit pokrok a životaschopnost projektu. [10]

Většina účastníků vývoje se obává jejich komplexnosti. Říká se, že každý projekt potřebuje unikátní pojetí. Avšak v praxi tomu tak není. Implementační postupy a nástroje by měly zůstat stejné nehledě na povahu projektu. Ovšem individuálnost jednotlivých projektů by měla být zohledněna v přístupu analytika. [16]

Většinu problémů, které vznikají při vývoji, se týkají dvou fundamentálních principů:

- „Lidé se pokoušejí vyřešit špatný problém neboli identifikovaný problém není opravdu to, co je špatně
- Řešení skutečného problému je většinou mnohem jednodušší, než se muže zdát“ [16, s. 4]

I když se tyto záležitosti mohou zdát jako elementární, nadále zužují průmysl už více jak 25 let. Příčina vzniká často již v zárodku vyhotovením nesplnitelných plánu dle nerealistických očekávání. Tudíž je stěžejní brát v úvahu reálné prostředí, ve kterém se vývoj odehrává. [16]

3.4.1 Zahájení

Než může začít analýza, jakožto první fáze vývojového cyklu, musí se zrodit myšlenka. Manažeři se musí shodnout, že v organizaci existuje nějaká potřeba či problém. Naplnění této potřeby spočívá v zahájení nějaké formy projektu. Potřeba může být například přidání nové funkcionality do již stávajícího systému nebo identifikace nového způsobu, jak zvýšit hodnotu organizace. [10]

Důležité je ověřit, zda danou potřebu je opravdu žádoucí splnit. Toto ověření většinou proběhne rychle. Nakonec osoby zainteresované v projektu učiní konečný verdikt, jestli se

myšlenku vyplatí dále prozkoumat. Pokud ano, dalším krokem je vyhotovení studie proveditelnosti a následně obchodního případu. [10]

Studie proveditelnosti je o tom, jestli projekt dává smysl z pohledu organizace. Součástí studie je tak zvaný odhad rozpočtu z vysoké úrovně. Tento rozpočet zohledňuje náklady na vývoj z nejhoršího i nejlepšího možného pohledu. Organizace si do rovnice také zavádí rentabilitu investic. Vypočtením rentability investic se matematicky dokáže, zda výsledek vývoje poskytne v budoucnu potřebné peněžní výnosy. [16]

Podniku uškodí, když se bude soustředit pouze na ekonomické hledisko. I nefinanční výnosy mohou přinést mnoho výhod. Z tohoto důvodu se studie proveditelnosti zaměřuje také na dosažitelnost z hlediska času, techničnosti a provozu. Výstupem studie je tedy rozhodnutí, které má zohlednit více druhů cílů. [16]

3.4.2 Analýza

Analýza je fáze, během které se shromažďují požadavky kladené na systém. Aby se zjistili všechny požadavky jsou prováděny seance s uživateli. Je samozřejmě možné, že některé požadavky byly již identifikovány během vyhotovení obchodního případu. Shromážděné informace jsou zapsány analytikem do přehledného dokumentu. Tento dokument se nazývá specifikace softwarových požadavků a vývojáři ho používají jako podklad. Samotný dokument by měl vývojářům stačit a neměli by se muset obracet zpět na zadavatele kvůli upřesnění požadovaných vlastností. [10, 16]

Každá funkcionalita systému se dá odvodit zpětně z fyzických požadavků uživatelů. Obzvlášť v podnicích vytváří lidé, a jejich interakce, základ pro každý systém. Když uživatelé vysvětlují své požadavky na systém, většinou popisují věci ve fyzickém světě. Nicméně informační systémy fungují v prostředí počítačů a na bázi logiky. Fyzické procesy pracují jinak a míň efektivně než jejich logický protějšek. Například způsob, jakým zákazník nakupuje ve fyzickém obchodě v porovnání s internetovým. [16]

Software můžeme nazývat logickou verzí fyzického světa. Analytik musí být schopen převést fyzické požadavky, které získá od uživatelů do automatizovaného logického světa.

Tento proces je stavebním kamenem analýzy a nejedná se vůbec o lehký úkol. K jeho splnění jsou potřeba nejenom analytické, ale i interpersonální dovednosti. [16, 17]

„Analytické dovednosti umožňují identifikovat problém, vyhodnotit klíčové prvky a rozvíjet užitečné řešení. Interpersonální dovednosti jsou obzvlášť užitečné systémovým analytikům, kteří musí pracovat s lidmi na všech organizačních úrovní, vyvažovat protichůdné potřeby uživatelů a efektivně komunikovat“ [17, s. 143]

Zjištění informací o systému, který má být vytvořen, od uživatelů pomocí rozhovorů, průzkumů či dotazníků není zdaleka jediná náplň práce ve fázi analýzy. Ze sesbíraných informací je potřeba porozumět podstatě navrhovaného systému. Z tohoto pochopení totiž analytik identifikuje jednotlivé požadavky, jejichž kombinací vzniká celistvý systém. Z požadavků analytik zjišťuje skutečnosti jako:

- Vstupy – data, která jdou do systému, a to buď manuálně nebo automaticky. Například osobní číslo, jméno a pozice zaměstnance, který přikládá svou elektronickou kartu k docházkovému terminálu.
- Výstupy – data, která opouštějí systém, stejně tak jak do systému vstoupila či v nějakém způsobu procesně zpracována na informaci.
- Procesy – logická pravidla transformující data
- Výkonnost – definuje vlastnosti systému jako rychlost, objem, kapacity, dostupnost a spolehlivost
- Bezpečnost – schopnost obrany systému a jeho dat před vnitřními a vnějšími hrozbami [17]

Určování prvků navrhovaného systému z požadavků je součástí činnosti, která se nazývá modelování. Touto aktivitou se v životním cyklu informačních systémů zabývají nejen analytici, ale i návrháři. Modelováním analytik konzultuje své pochopení systému s uživateli. Pokud uživatel objeví chybu, může být model ještě předělán, než se stane základem pro další návrhové a implementační činnosti. [18]

Model popisuje systém z určitého hlediska. Analytik má k dispozici velké množství různých druhů modelů. Formální podoba realizovaných modelů bude variabilní v závislosti na

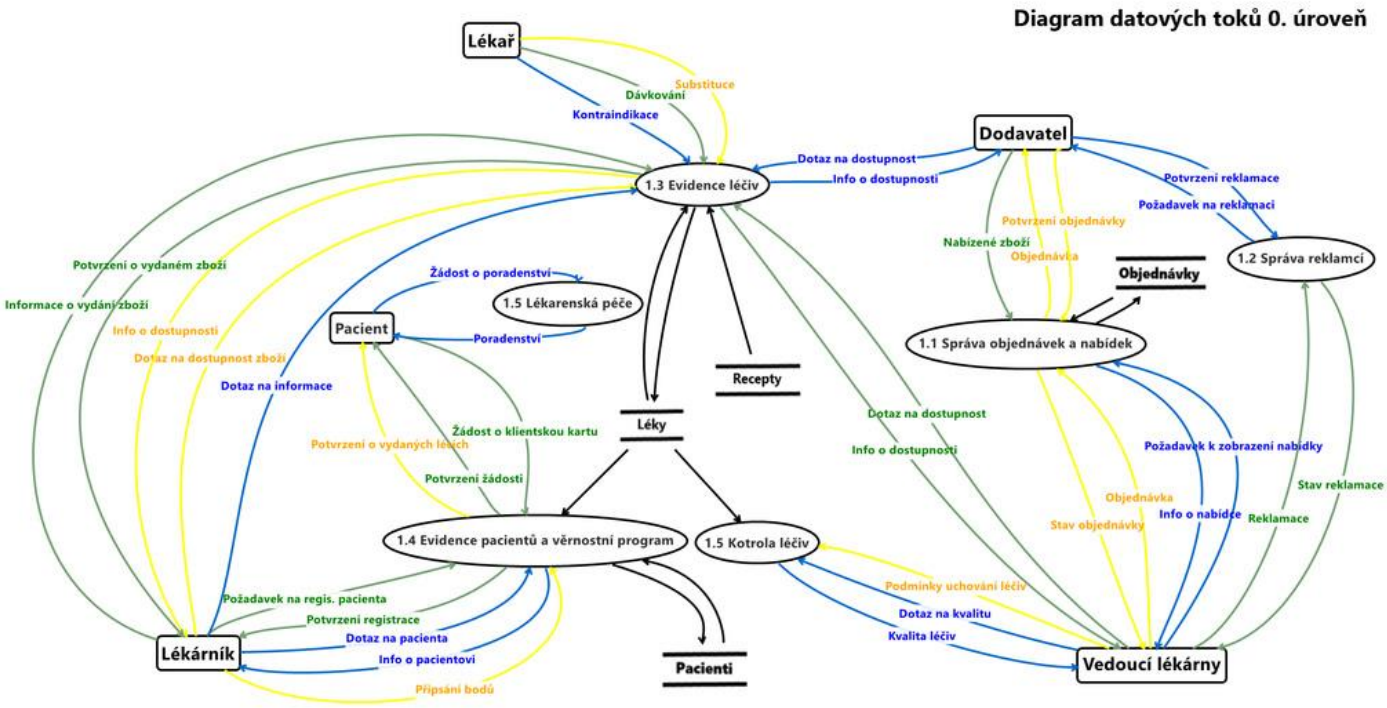
specifikách jednotlivých projektů. Menší projekty nepotřebují modely, které detailně popisují každý aspekt systému. [19]

Modely jsou vytvářeny v diagramech specializovaným softwarem. Tyto ilustrace se většinou řídí uznávaným standardem softwarových diagramů UML (v aj. „Unified Modeling Langugage“). I když někdy mohou být vytvořeny neformálně za malou chvíli. Například na kus ubrousku během firemního oběda. [18]

Podstata činností analýzy je řešení problémů. Obecně se každý problém dá řešit vícero způsoby. Tudíž i systémová analýza má více metod a postupů které si lze zvolit. Zpravidla se vybraná metoda systémové analýzy bude odvíjet od povahy samotného projektu. Ačkoliv se tak může zdát, tyto metody nejsou vylučující se alternativy. Kombinace těchto metod je možná a v některých případech žádoucí, protože se mohou navzájem doplňovat. [18, 20]

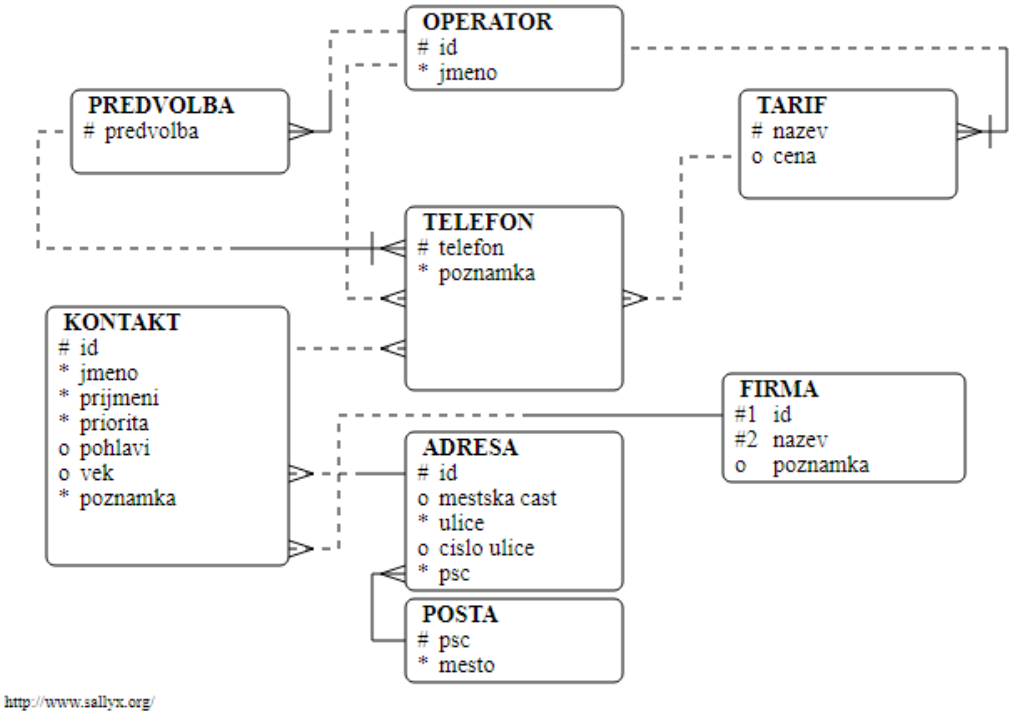
Tyto přístupy můžeme rozdělit na tyto druhy:

- Modelově orientované – založené na grafickém zobrazování. V současnosti je tato metoda často doprovázena využitím automatizovaných nástrojů jako je Microsoft Visio nebo Enterprise Architect.
    - Strukturovaná analýza – jedna z nejstarších metod (70. léta 20. století), která je používaná dodnes. Analyzuje procesy a tok dat, který mezi nimi probíhá. Hlavním modelem této metody je diagram datových toků. Tento diagram může sloužit jako předloha pro implementaci podnikových procesů a softwaru. [20]



Obrázek č. 5: Diagram datových toků lékárny. [21]

- Informační inženýrství – tradiční metoda zaměřující se na struktury uložených dat. Metoda využívá entitně-vztahový model (E-R diagram). E-R diagram je nadále široce používán při návrhu relačních databází. Původem alternativa strukturované analýzy, ale postupem času se začal používat v kombinaci se strukturovanou analýzou. [20]

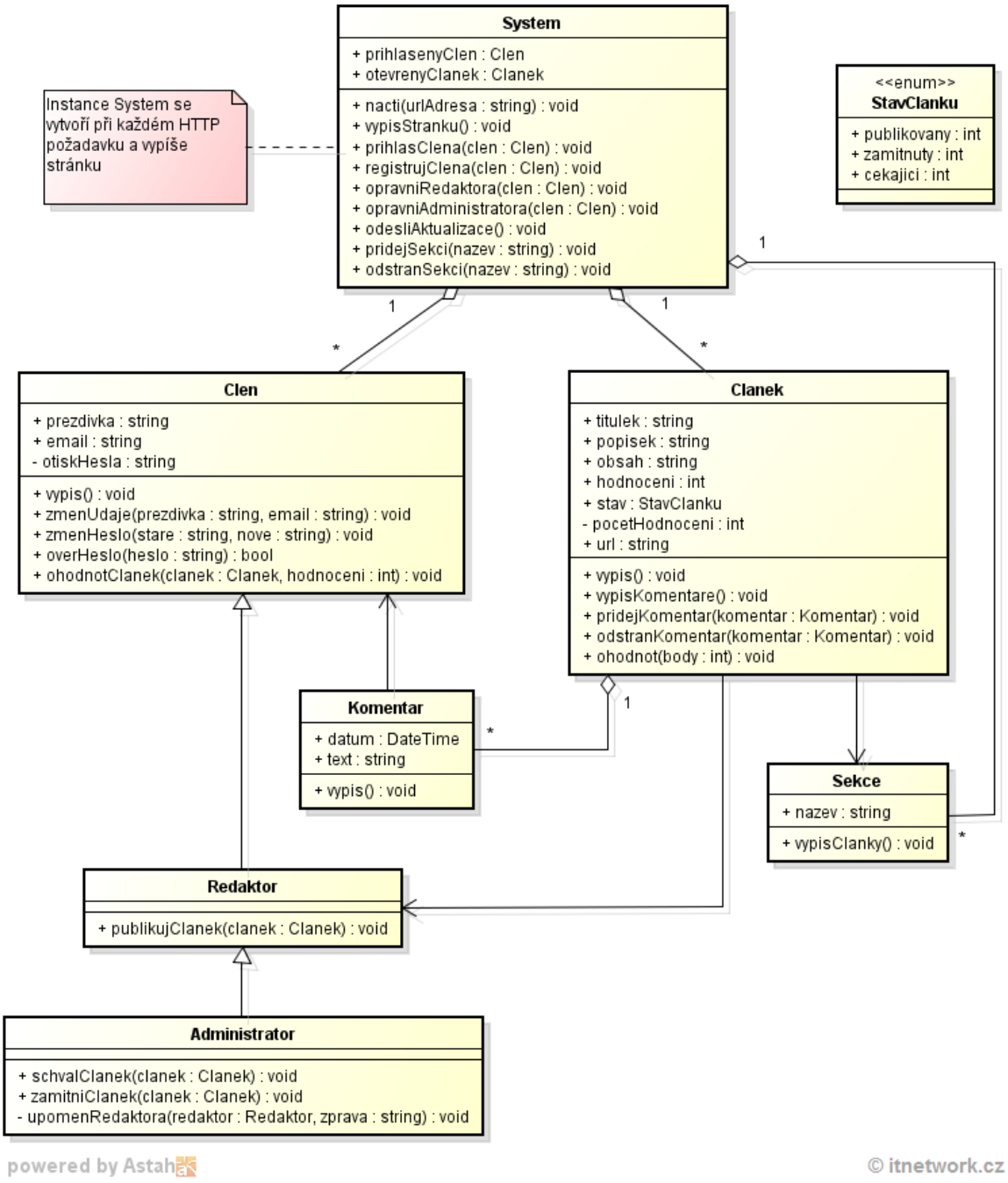


Obrázek č. 6: E-R diagram operátora. *[22]*

- Objektově orientovaná analýza – na rozdíl od dvou předchozích metod se tato, místo separátního zaměření na data a procesy, soustředí na množinu objektů, ve kterých jsou tyto informace zapouzdřeny, což umožňuje znovu použitelnost kódu. [20]

  Chování systému je definováno metodami, kterými jednotlivé objekty disponují. Analýza objektů probíhá staticky i dynamicky. V statické analýze je používán diagram tříd. Díky třídám se nemusíme opakovat, protože jsme schopni seskupit objekty, které sdílí stejnou definici. Zatímco dynamická část zkoumá prostřednictvím diagramů chování statických objektů v průběhu času. Úspěšným provedením dynamické analýzy se dá ověřit, že statické objekty podporují požadované chování. [19]

Jako příklad diagramu tříd je na obrázku níže uveden redakční systém. Na diagramu lze vidět jednotlivé třídy a jejich atributy, metody a typy vazeb mezi nimi. Třídy také obsahují datové typy svých atribut, což se dá považovat spíše za implementační detail. Je diskutabilní, zda je tato informace potřebná v analýze. Nicméně je vyžadována databázovým schématem, který při navrhování vychází ze specifikace modelu statické analýzy. [19]



Obrázek č. 7: UML diagram tříd redakčního systému. *[23]*

- Zrychlená analýza – metody zrychlené analýzy si zakládají na konstrukci tak zvaných prototypů. Prototyp si můžeme představit jako vzorovou část požadovaného systému. [19]
    - Objevování prototypů – technika, pomocí které se zjišťují uživatelské požadavky. Podstata je, že se uživatelům předkládají velice jednoduché a graficky nedodělané prototypy. Smyslem je odradit uživatele od posuzování pouze na bázi vzhledu. Tato technika by neměla být použita jako kompletní substitut modelově orientovaného návrhu. [21]
    - Rychlá architektonická analýza – prostřednictvím technologií reverzního inženýrství, které nabízejí automatizované nástroje generuje modely systému z prototypů či existujícího software. Výsledek je využíván jako základ, ze kterého analytik a dotazovaní uživatelů vychází. Tímto způsobem je tato metoda modelově orientovaná, přičemž si zachovává výhody zrychlené analýzy. [21]
- Agilní analýza – tato metoda vznikla jako reakce na analytiky, kteří argumentovali pro použití pouze jedné, dle jejich názoru, nejlepší metody. Principem je kombinace různých metod a technik tak, aby co nejlépe splnili zadání. Agilní analýza by teoreticky měla být kompromisem mezi produktivitou a kvalitou analýzy. [21]

	Strukturovaná analýza	Objektově-orientovaná analýza	Agilní/Adaptivní metody
Popis	Pohled na systém, co se týče dat a procesů, které s nimi pracují. Využívá tradiční vodopádový vývojový procesní model.	Data a procesy slučuje do reálných objektů a jejich metod. Objekty jako lidé, věci, události apod. Více interaktivní než tradiční metody.	Zaměřují se na spolupráci týmů. Rozdělení práce do jednotlivých cyklů nebo iterací, ve kterých postupně vyvíjejí a tím sníží risk. Většinou používá spirální vývojový procesní model
Nástroj modelování	Diagram datových toků/Modelování obchodních procesů	UML diagramy jako diagram tříd, objektů, aktivit, případů užití atd.	Kolaborativní nástroje, které zlepšují komunikaci. Dále techniky zlepšující kreativitu jako brainstorming či myšlenkové mapy. Lze využít i modelování obchodních procesů
Klady	Dlouhodobě ověřená metoda. Je hodně závislá na dokumentaci. V porovnání s ostatními metodami nabízí stejnou flexibilitu při častém střídání iterací. Dále je kompatibilní s metodami projektového řízení	Jednoduše se aplikuje s objektově-orientovanými programovacími jazyky. Při použití těchto jazyků, je kód modulární, znovupoužitelný a udržovatelný, což může snížit dobu vývoje a náklady.	Díky flexibilitě a efektivitě se lépe přizpůsobuje změnám. Časté produkování výstupů projektu a jejich validace redukuje risk.
Zápory	Změny jsou nákladnější obzvlášť v pozdějších fázích projektu. Definice požadavků probíhá v rané fázi a může se během vývoje změnit. K plnému pochopení uživatelských požadavků může být vyžadováno ukázání pochopitelnějších příkladů různých funkcionalit	Méně známá metoda. Komplexnější interakce mezi objekty a třídami ve velkých systémech.	Od členů týmu je vyžadována vyšší technická a komunikační schopnost. Možnost nastání rizikového faktoru v případě špatné strukturovanosti a dokumentace. Rozsah projektu se mění díky častějším změnám požadavků uživatelů.

Tabulka č. 1: Porovnání metod systémové analýzy. *[17]*

3.4.3 Návrh

Ve fázi návrhu se vychází ze softwarových specifikací, které byly stanoveny ve fázi analýzy. Cílem je navrhnout nejefektivnější řešení, které bude tuto softwarovou specifikaci splňovat. V předchozí fázi analýzy se vyvíjí logický model systému. Dalším krokem v návrhové části je vytvořit konkrétně definovaný fyzický návrh kterým se ve fázi vývoje programátoři řídí. Navrhuje se uživatelské rozhraní, datová struktura a architektura systému. [17]

Dalo by se konstatovat, že hranice mezi analýzou a návrhem se prolínají. Nicméně jedná se o odlišné fáze, které by měly být odlišně pojaty. Důraz na rozlišení je obzvlášť doporučován, protože prolínání těchto fází, ať už nechtěně nebo záměrně, může být důsledkem špatného softwarového vývoje. Je důležité, aby byla problematika pochopena dříve, než se přemýšlí nad řešením. Z tohoto důvodu by analýza měla odpovídat na otázku „Proč?“, kdežto návrh „Jak?“. Jakým způsobem probíhá přechod mezi analýzou a návrhem nelze přesně stanovit. Toto se odvíjí od lidského faktoru a vynaložené kreativitě. [19]

Úkoly spojené s návrhem informačních systémů se zaměřují na vytvoření komplexní specifikace výpočetního řešení. Toto řešení nazýváme fyzickým návrhem. Návrh systémů se tedy soustředí převážně na techniku a implementační stránku. Zatímco analýza se zabývá systémem spíše z podnikové perspektivy. [20]

Součástí systémového návrhu je rozdělení do logických a fyzických subsystémů. Logické subsystémy jsou procesy. Fyzické subsystémy jsou výpočetní technika a sítě. Dále je potřeba určit, jakým způsobem budou tyto subsystémy vzájemně komunikovat a zvolit správnou technologii pro tuto příležitost. Výběr technologií jako je programovací jazyk, systémy řízení báze dat (DBMS) či protokol. [19]

Jeden z nejdůležitějších aspektů v této fázi je návrh samotné systémové architektury. Architekturu můžeme vnímat jako tvar, který daný softwarový systém má. Tvar, jenž je definován způsobem rozložení, uspořádání, komunikace systémových komponent. I když má architektura, překvapivě, pouze malý vliv na funkčnost systému, je stěžejní. Hlavně proto, že podporuje životní cyklus systému. [24]

C. Martin rozděluje zásady systémových komponentů dle dvou aspektů:

- Chování
    - REP (The Reuse/Relase Equivalance Principle) – třídy a moduly by měly být rozděleny do komponent dle společných znaků. Nikoliv náhodně sestaveny. Třídy a moduly v komponentu sdílí stejný cyklus vydávání.
    - CCP (The Common Closure Principle) – třídy komponenty by měly být neoddělitelné, což znamená, že by komponenta neměla obsahovat nezávislé třídy.
    - CRP (The Common Reuse Principle) – pouze třídy, které mají tendenci spolu interagovat by měly být společně zakomponovány.
- Vazby
    - ADP (Acyclic Dependancies Principles) – Na jednotlivých komponentech by měly pracovat individuální vývojářské týmy. Jinými slovy každý komponent je oddělené pracovní prostředí. Grafy závislosti těchto komponentů nemohou být cyklické.
    - SDP (Stable-Dependency Principle) – komponenty nemohou být zcela statické, součástí změn je i jistá úroveň nejistoty. Nicméně na komponenty, od kterých se očekává změna (například během vývoje) nemá být tvořena závislost. Závislosti existují pouze mezi stabilními komponenty.

    - SAP (Stable-Abstractions Principle) – komponenty by měly být abstraktní podle své stability. Systém zapouzdřuje politiky vysoké úrovně jenom do stabilních komponentů. [24]

Správnou definici softwarové architektury je těžké v IT odvětví najít. Architekty může být softwarová architektura označována jako schéma systému. Nicméně někteří jí zase charakterizují jako plán vývoje systému. Dalo by se konstatovat, že ani jedna z těchto tvrzení nejsou nesprávná, jelikož skutečné pochopení softwarové architektury tkví v kombinaci těchto a dalších pohledů. [25]

„Dobrá architektura dělá systém snadno pochopitelný, jednoduchý k vývoji, jednoduchý k údržbě a jednoduchý k nasazení. Ultimátní cíl je minimalizovat celkové náklady a maximalizovat produktivitu programátorů“ [24, s. 148]

Richards definuje architekturu na základě 4 různých rozměrů, které se vzájemně doplňují:

- Struktura – jedná se o použitý styl architektury v daném systému. Je velké množství opakujících se různých architektonických vzorů, které lze použít. Například monolitická či vrstvená architektura.
- Vlastnosti architektury – ovlivňují kritéria pro úspěšný systém. Vlastnosti jako výkonnost, odolnost vůči chybám, škálovatelnost a tak podobně. Všechny tyto vlastnosti nesouvisí se samotnou funkcionalitou systému. Nicméně fungování systému se bez něj neobejde.
- Rozhodování o architektuře – pravidla pro vyhotovení systému. Například pravidlo, které z důvodu zabezpečení zakazuje přímou komunikaci prezentační a datové vrstvy.
- Zásady navrhování – podobné jako pravidlo v rozhodování o architektuře s rozdílem, že je více konkrétní a nejedná se o pevné pravidlo, ale spíše situační. [25]

Můžeme se setkat s tím, že termín „architektura“ se používá zaměnitelně se slovem „design“. Architektura vyjadřuje pohled z vysoké úrovně, který nezohledňuje detaily na úrovni nižší. Přičemž design implikuje struktury a rozhodnutí převážně na nižší úrovni. Nicméně tyto definice nedávají smysl z perspektivy návrhu softwaru, protože systém je jednotný celek, který ke svojí funkčnosti potřebuje jak nízkoúrovňové detaily, tak i vysokoúrovňovou strukturu. [24]

Hranici mezi architekturou a detailním návrhem je problematické určit. Poněvadž i změny architektury na vysoké úrovni mohou ovlivnit, jak se chová programovací kód v nejnižší vrstvě. Architektura je tedy svým způsobem určitá forma designu, protože se zabývá rozsáhlými rozhodnutími i miniaturizovanými prvky jako jsou moduly a komponenty. [26]

Je důležité, aby si v organizacích vývojářské týmy uměly obhájit stěžejnost architektury. Jelikož čím více se architektura systému zanedbává tím je větší pravděpodobnost, že porostou i náklady na vývoj. Toto tvrzení je zejména platné na softwarové architekty. Ačkoli užitečnost systému se odvíjí od jeho vlastností a funkcí samotný architekt není ten co je produkuje. Nicméně snadnost implementace pro programátory těchto vlastností a funkcí úzce souvisí s kvalitou použité architektury. [24]

Historicky se domnívalo, že vývojový procesní model má minimální vliv na softwarovou architekturu. Tento výrok je částečně pravdivý dodnes – procesní vývojový model, který vývojářský tým používá má nepatrný vliv na architekturu v oblasti inženýrských postupů. Nicméně v posledních dobách se v podnicích vliv zvyšuje díky adopci agilního vývoje. Jehož způsob vývoje zvyšuje četnost zpětných vazeb, což architektům umožňuje aktivnější experimentování. Další výhodou agilní metodiky, která je architektům prospěšná je tak zvaná restrukturalizace. Jinými slovy, v případě, že je potřeba migrovat mezi architekturami (například z monolitické na modernější alternativu) agilní přístup nabízí lepší podporu této a podobných změn. [25]

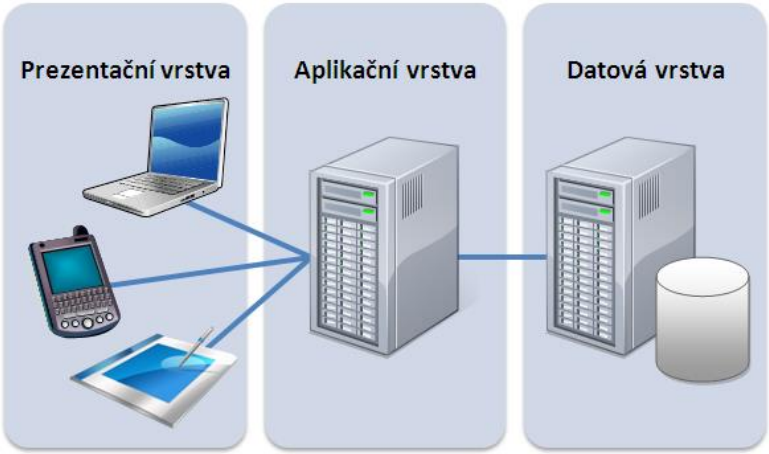
Každý typ architektury klade na vyvíjený systém určitá omezení, což znamená, že výběrem architektury omezujeme možnosti systému. Toto není neobvyklé, ve skutečnosti omezení nabízí řadu benefitů. Jako podpoření koncepční integrity, snížení složitosti a pochopení chování systému během chodu. [26]

Architektonický styl určuje, jakým způsobem programovací kód z uživatelské a back endové strany komunikují. Dále jak probíhá interakce kódu se zdrojem dat. Přičemž architektonický vzor je strukturou nižší úrovně, kterou se implementují konkrétní architektonické styly. [25]

Na malé systémy není potřeba klást takový důraz co se týče architektury. Šance na negativní dopad je na rozdíl od velkých podnikových systémů nízká. Nicméně na architekturu by se měl brát zřetel i v případě systémů malého rozsahu, které mají vysoké nároky na požadavky či jsou implementovány v nové oblasti se kterou ještě dotyčný nemá zkušenost. Nehledě na rozsah systému se tvrdí, že by architektura měla při návrhu dostávat co nejvíce pozornosti, protože přispívá k celkovému riziku projektu. [26]

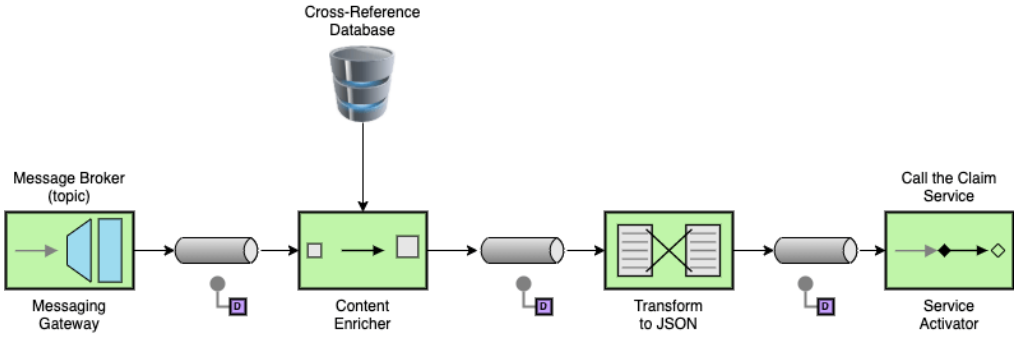
Existují dva základní typy architektonických stylů, které se dále rozdělují:

- Monolitické – tradiční styl, který se vyznačuje tím, že celý systém je jediným prvkem, který poskytuje veškerou funkcionalitu systému. Vše obvykle pracuje pod jediným procesem, jenž je připojen k jedné centrální databázi. [25]
    - Vícevrstvá architektura – rozděluje komponenty systému do několika vrstev, které mezi sebou komunikují, což podporuje modifikovatelnost, přenositelnost a znovupoužitelnost. Nejvíce používaná je tzv. třívrstvá. [26]



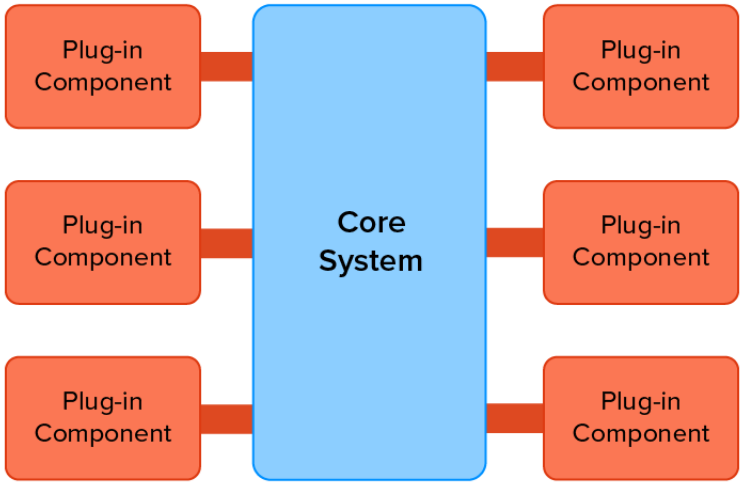
Obrázek č. 8: Třívrstvá architektura. *[27]*

    - Pipeline architektura – využívána pro zpracování velkých dat od jednoho datového úložiště do druhého. Data jsou postupně modifikovány přes filtry od vstupu do výstupů. Filtry je doporučováno řetězit, tak aby každý filtr zpracovával data pouze v jedné určité formě. [26]



Obrázek č. 9: Pipeline architektura. [28]

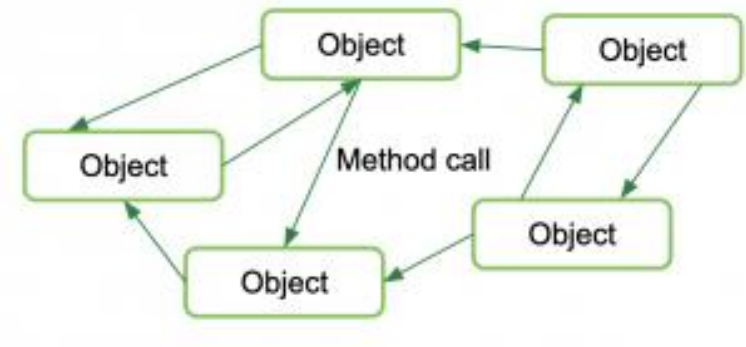
- Mikrojádrová architektura – typ architektury, který rozděluje systém na dvě části. První částí je mikrojádro, které nabízí základní minimální funkcionalitu. Druhá část, respektive části jsou zásuvné komponenty, které omezené funkce jádra rozšiřují. Aplikační logika mikrojádra a zásuvných komponentů je mezi sebou nezávislá. Tudíž hlavní přednosti této architektury spočívají v rozšiřitelnosti, přizpůsobivosti, izolací funkcí a samotném způsobu zpracování informací. [25]



Obrázek č. 10: Mikrojádrová architektura. [29]

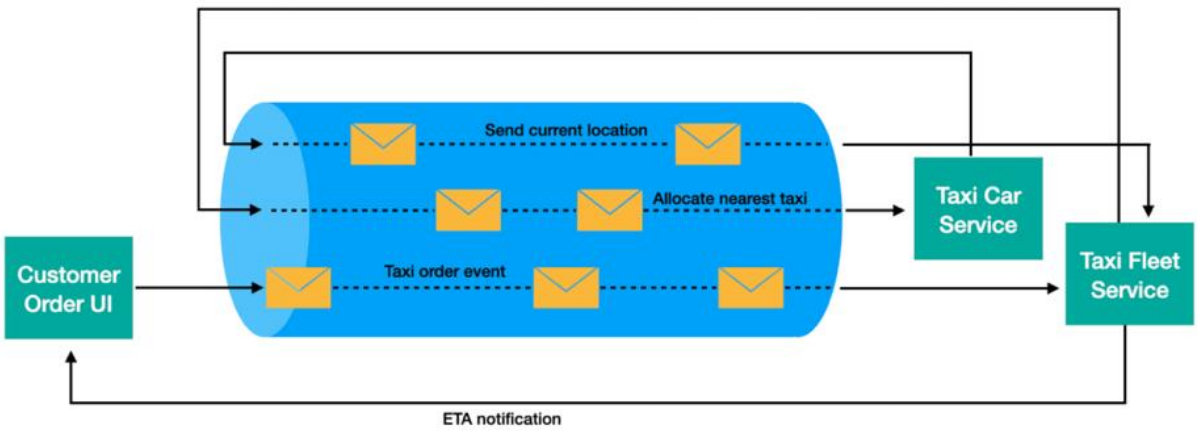
- Distribuované – opak monolitického stylu. Systém je rozdělen na více služeb, které komunikují po síti přes protokoly. Oproti monolitickým architekturám nabízí plynulejší nasazení a údržbu, protože vývojářské týmy mohou na jednotlivých částech systému pracovat nezávisle. [25]

Objektově (servisně) orientované architektury – každý prvek v systému reprezentuje objekt. Jejich komunikace probíhá po počítačové síti prostřednictvím vzdálených volání. Individuální objekty představují fyzická zařízení, která jsou kompletně nezávislá vůči svému prostředí. V důsledku distribuovaná varianta objektu umožňuje umístit své rozhraní a instanci na dvě separátní stanice v počítačové síti. [30]



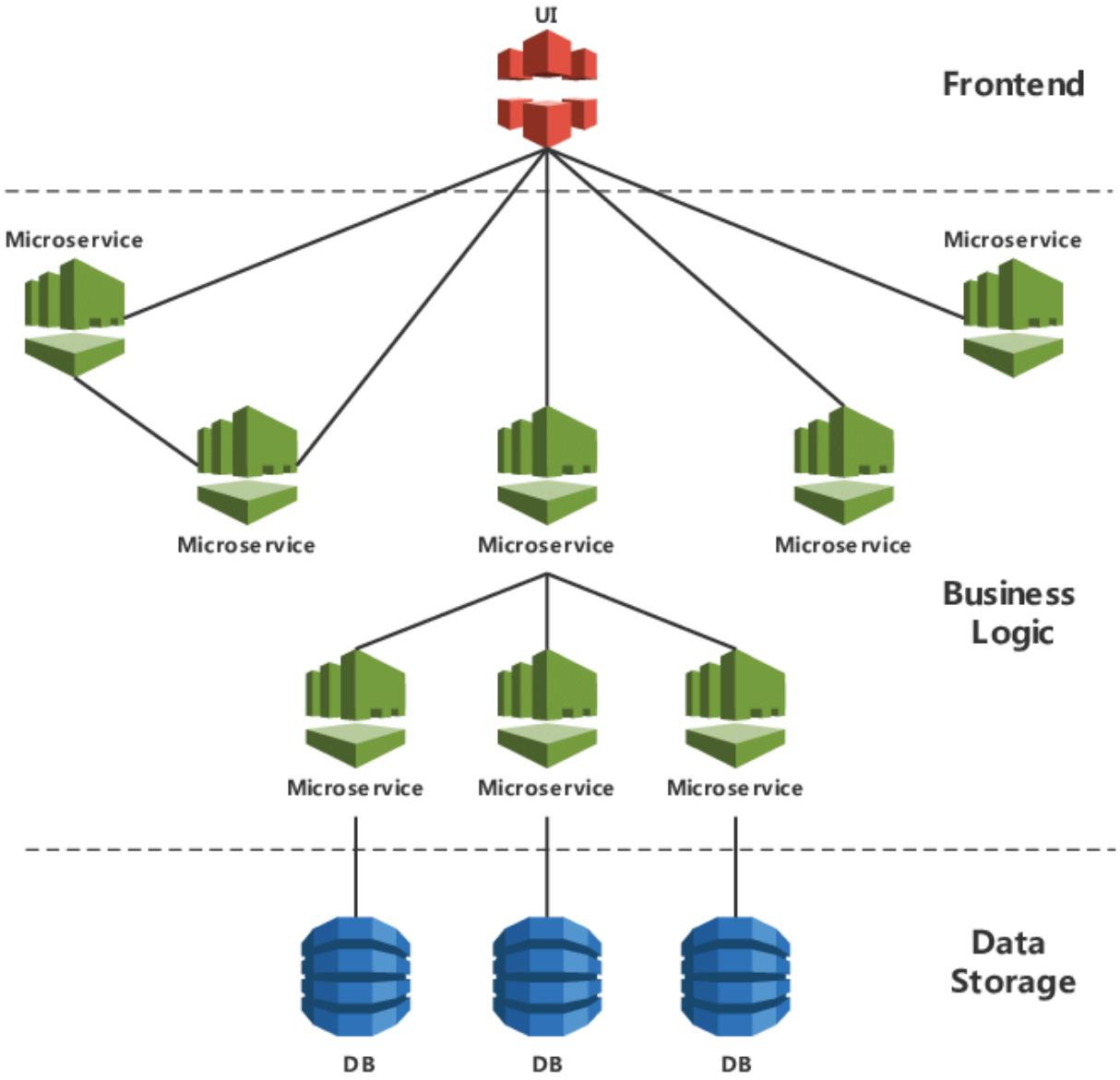
Obrázek č. 11: Objektově orientovaná architektura. [31]

Událostmi řízená architektura – používá ke svému chodu služby. Tyto služby mezi sebou vyměňují informace produkcí a spotřebou událostí. Události se rozesílají do komunikačního kanálu, ke kterému se služby připojují dvěma porty podle toho, jestli danou událost posílají nebo přijímají. Tak zvané „publish“ a „subscribe“ porty nejsou spojované. Tím pádem je systém udržovatelný a snadnější k rozvoji. [30]



Obrázek č. 12: Událostmi řízená architektura. [32]

- Mikroslužbová architektura – v posledních letech na vzestupu popularity. Jednotlivé služby jsou samostatně běžící procesy, kde každý má zvlášť svojí vlastní databázi. Dále se hojně využívá ve spojení s virtualizačními a kontejnerizačními technologiemi. Tato architektura sice řeší problémy spojené s distribuovaným sdílením, nicméně její největší úskalí je ve výkonu. [25]



Obrázek č. 13: Mikroslužbová architektura. [33]

	Více vrstvá	Pipeline	Mikrojádrová	Servisně orientovaná	Událostmi řízená	Mikroslužbová
Nasaditelnost	1	2	3	1	3	4
Pružnost	1	1	1	3	3	5
Evolučnost	1	3	3	1	5	5
Odolnost vůči chybám	1	1	1	3	5	4
Modularita	1	3	3	3	4	5
Lacinost nákladů	5	5	5	1	3	1
Výkonnost	2	2	3	2	5	2
Spolehlivost	3	3	3	2	3	4
Škálovatelnost	1	1	1	4	5	5
Jednoduchost	5	5	4	1	1	1
Testovatelnost	2	3	3	1	2	4

Tabulka č. 2: Porovnání vlastností architektonických stylu dle bodového hodnocení (vlastní zpracování dle zdroje) [25]

3.4.4 Vývoj

Po analýze problému a vyhotovení návrhu přichází fáze realizace. Ze specifikací od návrhářů systému začnou programátoři s konstrukcí informačního systému. Pokud byly předcházející části životního cyklu vykonány dobře bude konstrukce produktivním a efektivním obdobím. Obdobím, během kterého vývojářský tým postupně píše zdrojový kód a tím budují systém od úplných základů až po plně funkcionální systém. [34]

Už od začátku konstrukce je důležité stanovit a následovat kódovací techniky. Jelikož způsob, jakým je informační systém již zpočátku kódován má velký vliv na celkové náklady projektu. Dále se pozdějším aplikováním kódovacích technik vystavujeme neúspěchu projektu. Vzhledem k tomu, že je téměř nemožné nebo až nadměrně nákladné aplikovat zpětně do každého z tisíce řádků již napsaného kódu. [34]

Disciplinované dodržování kódovacích technik umožňuje vytvořit tak zvaný „čistý kód“. Primární vlastnost čistého kódu je, že je jednoduchý. Jednoduchý v aspektu pochopitelnosti. Stejně jako jazyky, kterými mluvíme čistý kód by měl být pochopitelný napříč veškerou demografií, která rozumí stejnému kódu (jazyku). Porozumění kódu po celém vývojářském týmu přináší řadu benefitů. Jako čitelnost, modifikovatelnost, rozšiřitelnost či udržovatelnost. [35]

„Ve zkratce programátor, který píše čistý kód je umělec, který může vzít čisté plátno přes několik řad transformací, dokud z něj není elegantně nakódovaný systém“ [35, str. 7]

Jeden ze znaků kvalitního softwaru je mít kódovací standard. Kódovací standard zajišťuje jednotný styl, jenž umožní pochopení kódu i v případě odchodu původních vývojářů. V případě, že se používá v rámci organizace stejný programovací jazyk měl by být zachován stejný kódovací standard od projektu k projektu. [34]

Během vývoje jsou s programátory pravidelně vedeny revize kódu. Hlavním účelem je detekce chyb. Dále je kladen důraz na to, že je kód v souladu s kódovacími standardy. Přesto je zbytečné standardizovat kód zcela z každého hlediska. Příliš mnoho standardů způsobí, že vývojáři nebudou mít kapacitu si je všechny pamatovat a dodržovat. [34]

Dle McConnella se tzv. kódovací standard zabývá těmito oblastmi:

- „Uspořádání tříd, modulů, rutin a kódu v rutinách
- Komentování tříd, modulů, rutin a kódu v rutinách
- Názvy proměnných
- Názvy funkcí, včetně názvu častých operací jako je získávání a nastavování hodnot ve třídách a modulech
- Maximální délka rutiny v řádcích zdrojového kódu
- Maximální počet rutin v rámci třídy
- Povolený stupeň složitosti, včetně omezení použití příkazu Goto, logických testů, vložených cyklických příkazů a tak dále
- Kódové uplatňování architektonických standardů pro správu paměti, zpracování chyb, ukládání řetězců a tak dále
- Verze používaných nástrojů a knihoven
- Konvence pro používání nástrojů a knihoven
- Jmenné konvence pro soubory se zdrojovým kódem
- Adresářová struktura zdrojového kódu pro vývojářské počítače, kompilátorské počítače a nástrojů pro správu
- Obsah zdrojových kódů (například jedna C++ třída v jednou souboru)

- Způsobů, jak označit nedodělaný kód (například pomocí „TBD“ komentářů)“ [34, str. 201]

Jména proměnných, funkcí a tříd by měla být výstižná. To znamená, že po přečtení by měl programátor odhalit jejich účel. Jména jako „trida1“ nebo „vysledek“ nemají téměř žádný význam v kontextu systému jako celku. Také je zbytečné nazývat proměnné podle jejich datového typu – například „cisloInt“. [35]

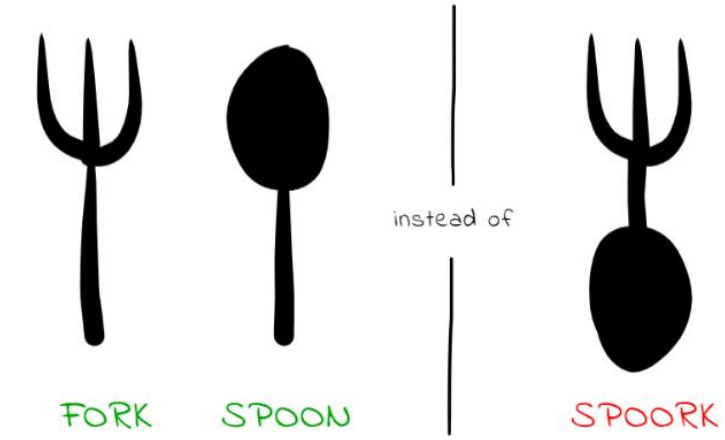
Pro jména tříd a proměnných se používají podstatná jména. Zatímco metody a funkce by měli obsahovat slovesa. Mimo jiné je u metod a funkcí doporučována velbloudí notace. Jako třeba „zapisDoSouboru()“. Je doporučováno při pojmenovávání elementů v kódu využívat termíny související s programováním – názvy algoritmů, programovacích vzorů a tak podobně. Případně lze použít jména, která se týkají vlastnosti nebo podniku. Například třída „Vzorek“ a proměná „cisloVzorku“. [35]

Funkce by měly být relativně malé a dělat pouze jednu věc. Je stěžejní, aby všechny funkce kvůli snadnějšímu pochopení využívaly pouze jednu úroveň abstrakce. Jakmile funkce operuje na nižší i vyšší úrovni stává se těžší k pochopení. Znakem zbytečně zvyšující se komplexnosti bývá rozdělení funkce do dalších podčástí. Pokud funkce skutečně dělá jednu věc, nedává smysl jí rozdělovat na více částí. [35]

Programovací bloky jako if, else a while uvnitř funkcí je účelné rozložit na pouze jeden textový řádek. Do jednotlivých bloků je vhodné přidávat další volání procedur. Přidává to na přehlednosti, když má volaná procedura výstižné jméno. Přehlednost kódu mohou také ovlivnit vnořené struktury, kterým je těžké se vyhnout. Nicméně pro snadné čtení a pochopení funkcí je doporučováno se držet maximálně druhé úrovně odsazení. [35]

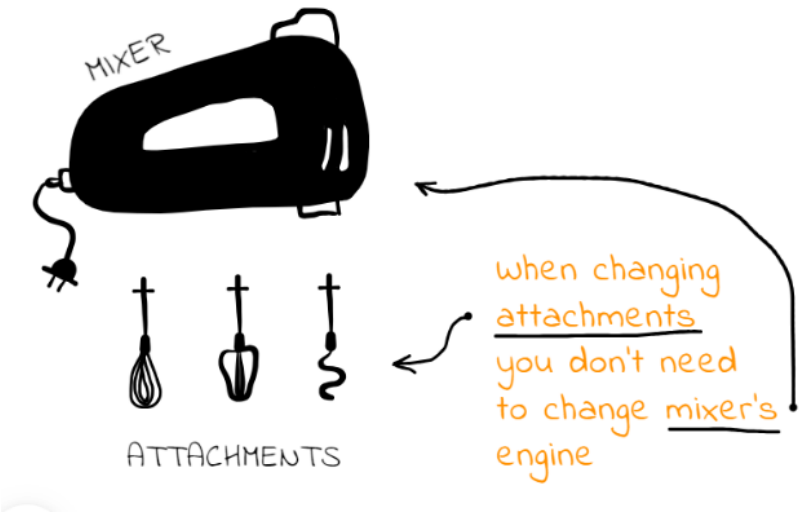
Jakým způsobem organizovat data a funkce do tříd a jejich způsob komunikace nám říkají tak zvané SOLID principy:

- S (Single-Responsibility Principle) – třída by měla mít pouze jeden účel



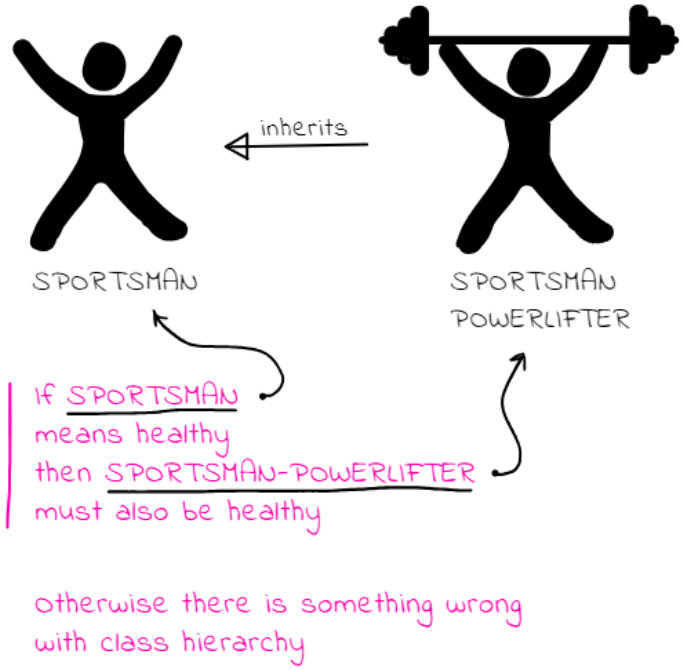
Obrázek č. 14: Single-Responsibility Principle příklad [36]

- O (Open-closed Principle) – entity by měly být modifikovatelné a zároveň neumožňovat změnu samotné třídy



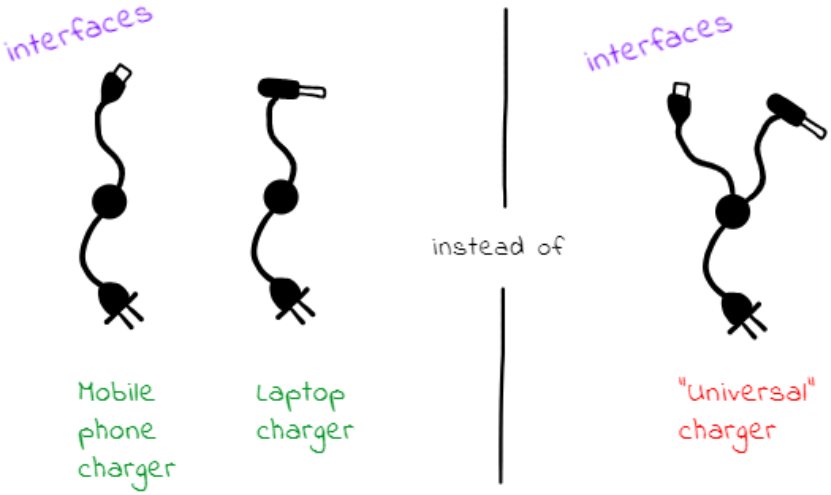
Obrázek č. 15: Open-closed Principle příklad [36]

- L (Liskov Substitution Principle) – každá podtřída nebo odvozená třída je nahraditelná základní nebo rodičovskou třídou



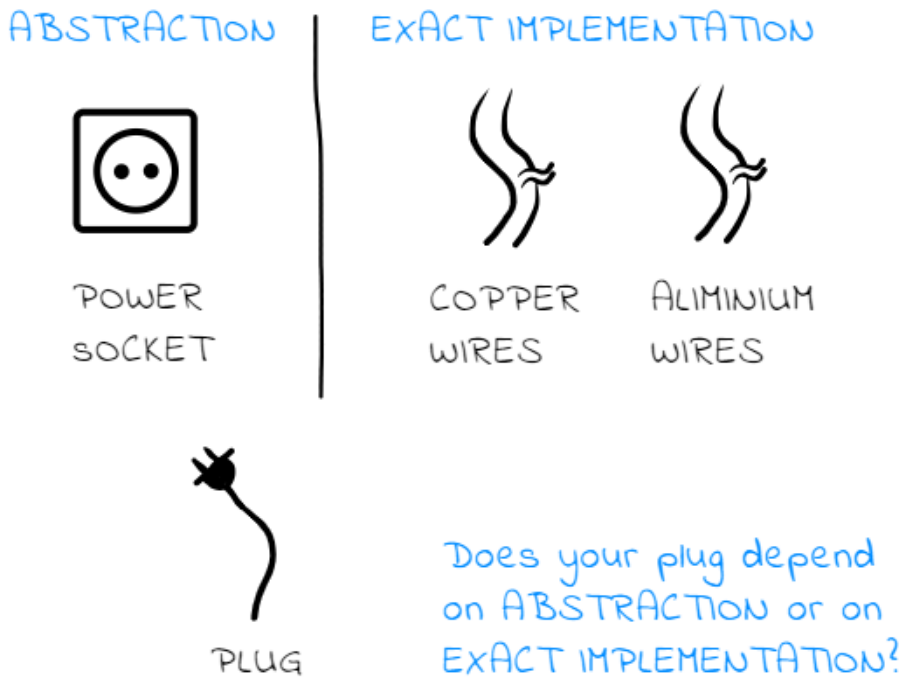
Obrázek č. 16: Liskov Substitution Principle příklad [36]

- I (Interface Segregation Principle) – klient by neměl implementovat rozhraní či metody, které nevyužívá.



Obrázek č. 17: Interface Segregation Principle příklad [36]

- D (Dependency Inversion Principle) – vysokoúrovňový modul by neměl záviset na nízkoúrovňovém, k tomu jsou abstrakce. Entity by neměly záviset konkrétních implementacích. [24]



Obrázek č. 18: Dependency Inversion Principle příklad [36]

3.4.5 Testování

Pátá fáze životního cyklu informačních systémů se zabývá testováním. Testování může začít hned jakmile vývojáři v předchozí fázi doprogramují systém a jeho komponenty. Nicméně správně by měla běžet souběžně již během konstrukce. Účelem fáze testování je ujistit se, že systém splňuje definované uživatelské požadavky na přijatelné úrovni kvality. Dále, že neobsahuje žádné chyby. [34]

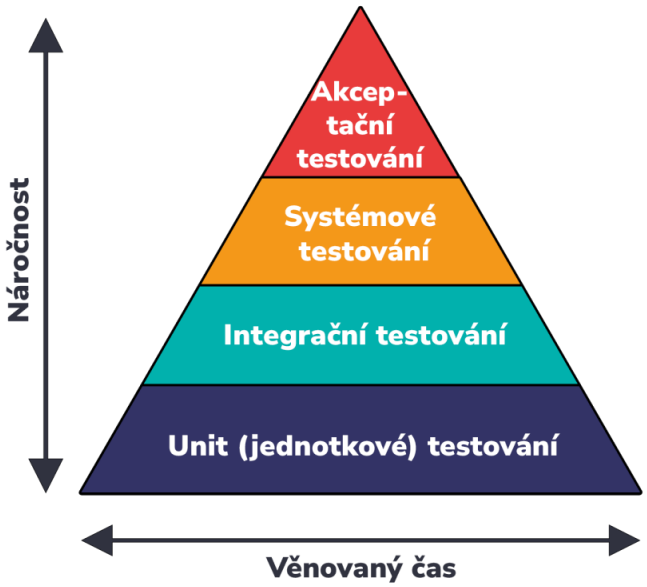
Testování je součástí důležitých aspektů jako je ekonomie a lidské psychologie. I když by se ideálně měly testovat všechna možná chování programu, toto je prakticky nemožné. Důvod je prostý. I sebemenší program může obsahovat stovky až tisíce různých kombinací vstupů a výstupů. Tudíž testování komplexního softwaru by bylo příliš časově nákladné a vyžadovalo nemálo lidských zdrojů, aby se ekonomicky vyplatilo. [38]

Pro konkrétní testování se sepisují tak zvané testovací případy. Testovací případ, kromě popisování předpokládaných vstupů a výstupů, také popisuje chování samotného programu. Program by měl být předvídatelný a konzistentní. Jeho chování by mělo být přesně v souladu s tím, jak byl navržen a neměl by dělat něco nečekaného. Přesto by testovací případy v případě objektových jazyku měly zohledňovat jeho problematické části jako je instancování objektů a správa paměti. [38]

Testovací případy se vytvářejí na základě dvou technik. Tyto techniky se nazývají černá a bílá skřínka. Stav černé skřínky nastává, když všechny informace o systému jsou neznámé. Na systém se v tomto stavu dá hledět pouze z vnějšího uživatelského pohledu. Na druhou stranu bílá skřínka je situace samotného vývojáře, který má dostupný veškerý zdrojový kód a zná vnitřní fungování systému. Technika černé skřínky většinou testuje ze strany uživatele. Zatímco bílá skřínka testuje spíše zdrojový kód. [38]

Testování se provádí na více úrovních:

- Akceptační – jsou prováděny, aby se ověřilo, zda software jako celek vyhovuje požadavkům ze strany zákazníka či odběratele.
- Systémové – testování systému jako celku, včetně databází, front end aplikací a dalších komponentů.
- Integrační – ověřuje správné chování mezi komponenty.
- Jednotkové – neboli v aj. tak zvaný unit test je validování prvků systému na nejnižší možné úrovni. [39]



Obrázek č. 19: Pyramida testovacích vrstev [40]

Kritérium	Jednotkové	Integrační	Systémové	Akceptační
Účel	Správné fungování jednotky/modulu	Správné fungování integračních jednotek	Správné chování celého systému po integraci	Splnění požadavků zákazníka
Zaměření	Nejmenší testovatelná část	Rozhraní a interakce mezi moduly	Interakce a souhrné fungování všech modulů	Software splňující stanovenou specifikaci
Pravidelnost	Jakmile je nový kód napsán	Jakmile je přidán nový komponent	Jakmile je software dokončen	Jakmile je software připraven k provozu
Zodpovědná osoba	Vývojář	Vývojářský tým	Testovací tým	Vývojářský tým a koncový uživatelé
Testovací techniky	Většinou bílá a šedá skřínka	Bílá a černá skřínka	Většinou černá skřínka a šedá skřínka	Černá skřínka
Automatizační nástroj	JUnit, PHPUnit, TestNG atd.	SoapUI, Rest klient atd.	WebDriver	Cucumber
Složitost	Komplexní (požaduje ovladače nebo jeho části)	Komplexní (může požadovat ovladače nebo jeho části)	Žádné ovladače nebo jeho části vyžadovány	Žádné ovladače nebo jeho části vyžadovány

Tabulka č. 3: Vlastnosti testovacích úrovních [41]

3.4.6 Implementace

Smyslem fáze implementace je fyzické nasazení softwaru do produkčního prostředí. Typicky se jedná o aktivity jako příprava infrastruktury, inicializace databáze a následné otestování fungování za ostrého provozu. Před zahájením implementace by se v rámci týmu měli pro tyto akce stanovit předem stanovené postupy. [42]

Instalaci a otestování softwaru předchází příprava infrastruktury. U každého typu infrastruktury se odvíjejí kroky dle různých parametrů. Například u serverů to je druh a verze operačního systému a typ prostředí. Dalším příkladem jsou třeba velikost disků a typ u databází. Co se týče síťové infrastruktury tak zařízení firewall, které může pro správnou funkčnost systému vyžadovat dodatečné povolení používaných portů. [42]

Není-li pro implementaci používána externí infrastruktura třetí strany je důležité, aby projektový tým dbal zřetel na správnost infrastruktury. Projektový tým by se měl o tomto pravidelně ujišťovat a případně dodatečně navýšit hardwarové zdroje. Navyšování v některých organizacích může být zdlouhavý papírový proces, proto je doporučováno zpracovávat v dostatečném časovém předstihu. [42]

Bezprostředně po implementaci a před tím, než bude konfigurován by se měl software otestovat. To i v případě implementace softwaru třetí strany. Některé projekty s krátkou časovou rezervou můžou testování z důvodu časové úspory vynechat. Nicméně testování je rozhodující pro ověření správné funkčnosti softwaru a stanovení výchozích hodnot pro projektový tým. [42]

3.4.7 Údržba

Údržba je poslední a zároveň nejdelší období v životním cyklu informačních systémů. Systém je udržován od implementace první verze až po vyřazení z provozu. Toto období může být pro některé kritické systémy několik desetiletí. Během tohoto období prochází systém řadou změn. Změny jako hlášení a řešení nových závad, přidávání nových funkcí a podpora nových technologií. [43]

Významově se údržba jakožto proces a fáze životního cyklu často mezi sebou zaměňují. Nicméně jedná se o dvě rozdílné věci. Fáze vymezuje období. Zatímco proces aktivity, které se v tomto období dělají. V každém případě sdílí stejný cíl. Jímž je změna softwaru bez narušení jeho konzistence a konceptuální integrity. [43]

Je relevantní znát rozdíl mezi softwarovou a hardwarovou údržbou. Například, u údržby auta si představíme jako náhradu rozbitých či opotřebovaných částí. Tyto části můžou být různé od startéru po výměnu náplně do ostřikovačů. Stejně tak to je i u hardwarové údržby, kde lze měnit výpočetní zařízení a jejich zdroje dle aktuální potřeby software. V porovnání s hardwarovou údržbou je softwarová varianta více nepředvídatelná. [44]

Aktivity softwarové údržby můžeme kategorizovat následovně:

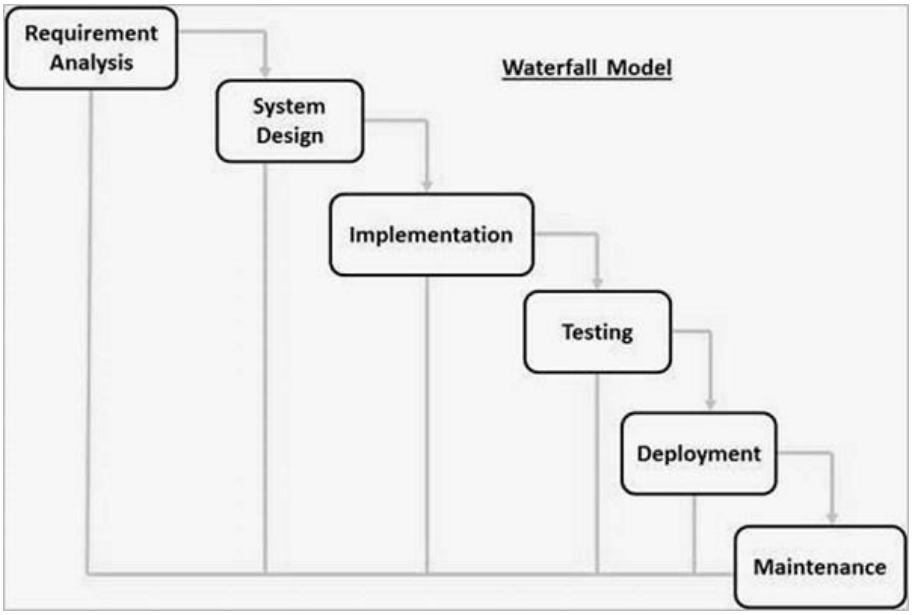
- Korekce – srovnání nesrovnalostí mezi očekávaným a reálným chováním systému
- Vylepšení – transformace systému, která mění jeho chování či implementaci
    - Modifikace stávajících požadavků
    - Vytváření nových požadavků
    - Modifikace implementace (bez změny požadavků) [44]

3.5 Vývojové procesní modely

Ačkoliv se tak může zdát, fáze životního cyklu informačních systémů, které byly popsány v minulé kapitole nemusí jít nutně za sebou. Přestože to tak může být v tradičním pojetí softwarového vývoje. Nicméně způsob, jakým jsou definované jednotlivé fáze a jejich pořadí definují tak zvané vývojové modely životního cyklu IS. [45]

3.5.1 Vodopád

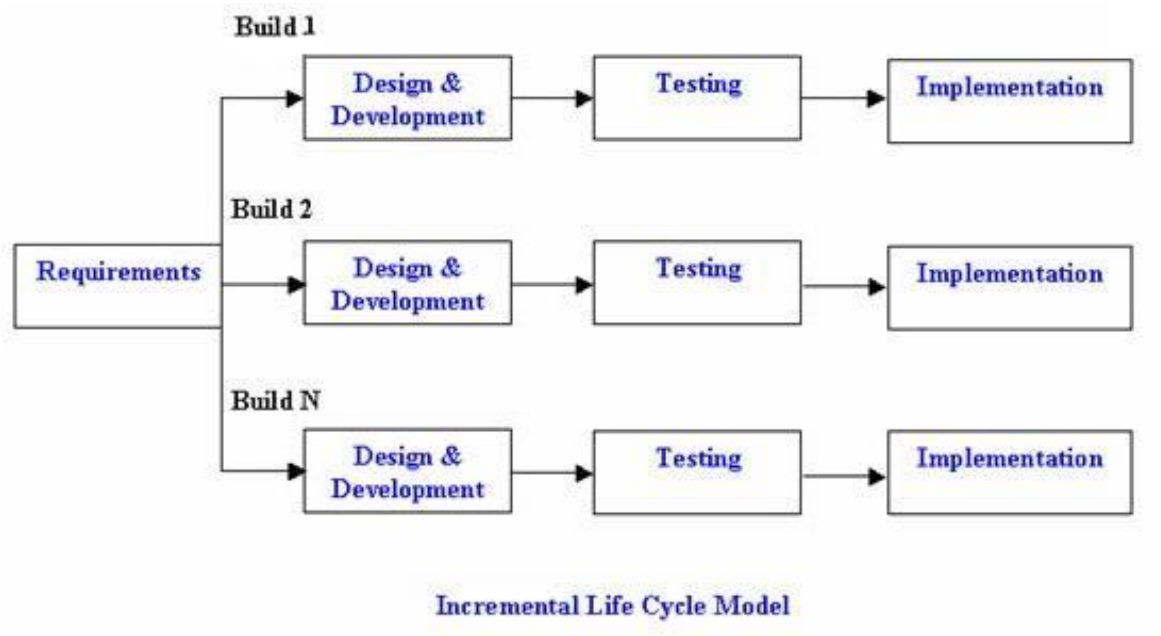
Vodopád je tradiční model softwarového vývoje, ve kterém jdou jednotlivé fáze sekvenčně za sebou. I když je tento model považován za teoretický základ není v praxi tak často využíván. Nutno podotknout, že tak to bylo i před příchodem agilních metodik. Příčinou může být vlastnost, dle které jakmile je ukončena jedna fáze a zahájena druhá není možné se vrátit. Vývoj je variabilní, a tak je často nutné se vrátit o jednu nebo dvě etapy, aby byly opraveny chyby či vyřešit změnu požadavků. [45]



Obrázek č. 20: Vodopádový vývojový model [46]

3.5.2 Inkrementální model

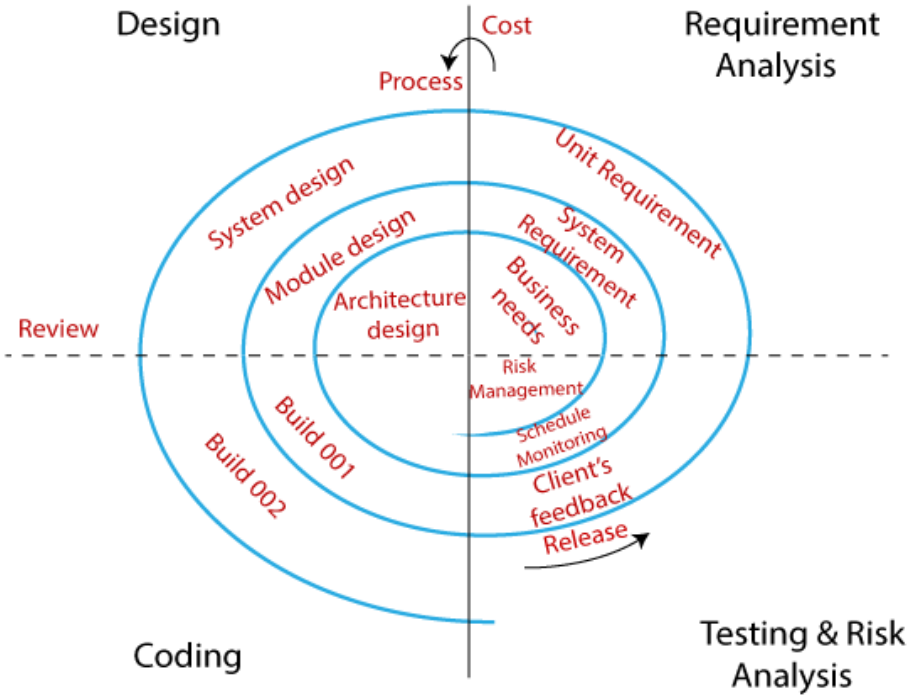
Na rozdíl od vodopádového modelu je více přizpůsobitelný změnám. Jelikož jsou cykly rozděleny do menších dílčích podčástí. V každé pod části vzniká nová verze (prototyp) systému a na základě zpětných vazeb od klientů se zjišťují nové požadavky, které jsou zohledněny v dalších fázích. Prototyp se postupně transformuje na plně funkcionální systém. Tento přístup je vhodný, když je potřeba znát dříve, zda systém splňuje požadavky na výkon a chování jeho kritických prvků. Dále model snižuje riziko implementace, protože je rozdělen do delších časových úseků. [47]



Obrázek č. 21: Inkrementální vývojový model [48]

3.5.3 Spirálový model

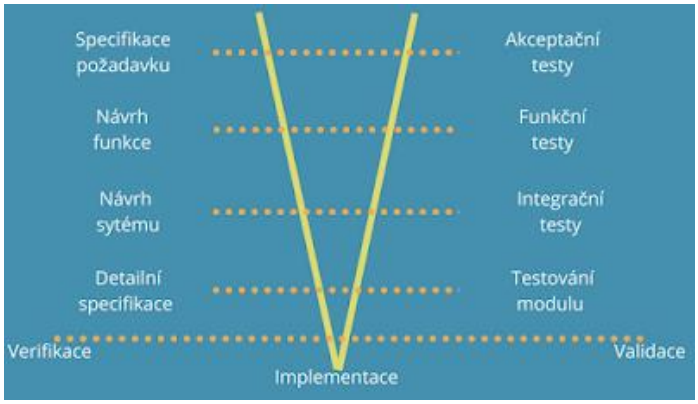
Vznikl jako alternativa vodopádového modelu, která má řešit jeho úskalí. Každý cyklus prochází jedním ze čtyř kvadrantů, kde osa x reprezentuje pokrok a osa y náklady. Podstata modelu je minimalizovat risk. Risk je minimalizován díky fázi analýzy rizik, která probíhá před každým cyklem. Zjištěné riziko je následně možné zmírnit. Například prototypováním požadavků, pokud z analýzy rizik vyplyne, že nejsou pochopeny. [47]



Obrázek č. 22: Spirálový vývojový model [49]

3.5.4 V-model

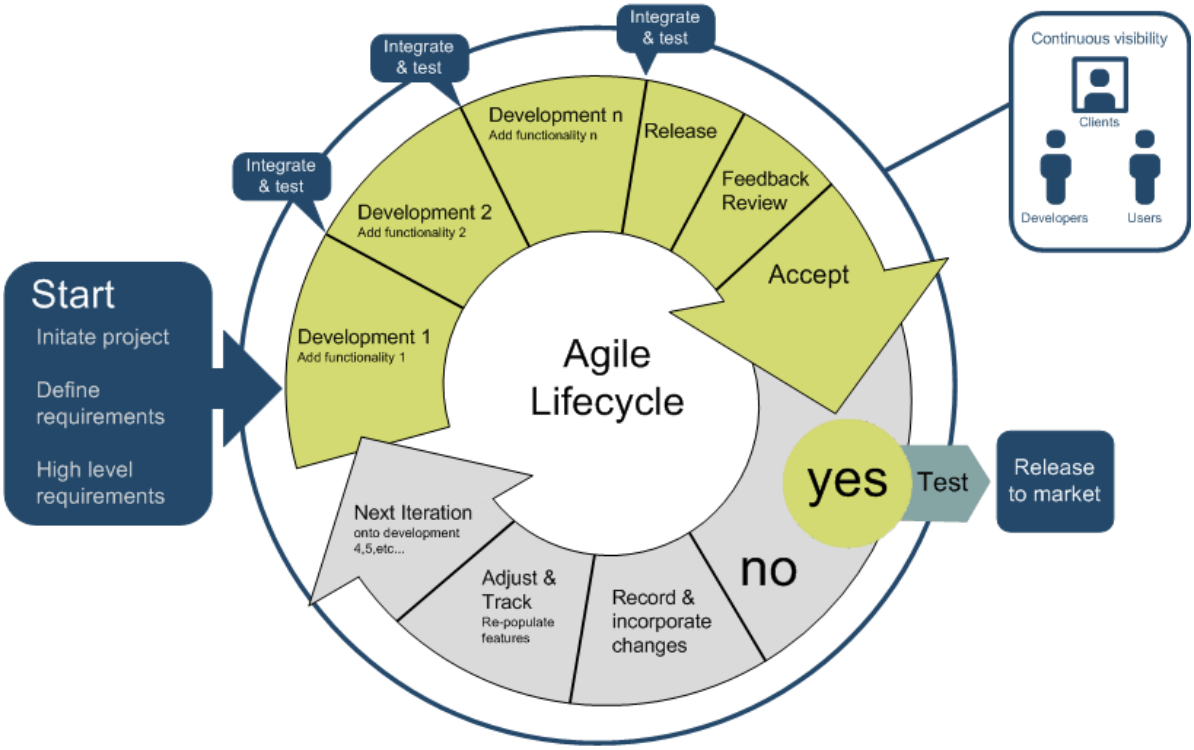
Rozšíření vodopádového modelu, které klade větší důraz na ověřování za pomocí testování. Je používán převážně při vývoji technických bezpečnostních systémů. Model je rozdělen na dvě částí: verifikační a validační. Každá verifikační část je následována tou validační. Ve verifikační části probíhá vývoj, zatímco validační je testování. Ačkoliv je hlavním cílem tohoto modelu ověřování a validace, jeho hlavní kritikou je absence testování v raných fázích. [45]



Obrázek č. 23: V vývojový model [50]

3.5.5 Agilní modely

Agilní modely využívají kombinací tradičního iterativního a inkrementálního způsobu vývoje. Agilní přístup vznikl jako reakce na klasické metody popsané výše. Mezi širokou veřejnost se zpopularizoval v roce 2001, kdy vyšlo tak zvané „The Agile Manifesto“. Základní podstata je rychlá reakce na změny, jenž je docílena prostřednictvím silné sebeorganizace týmů. [45]

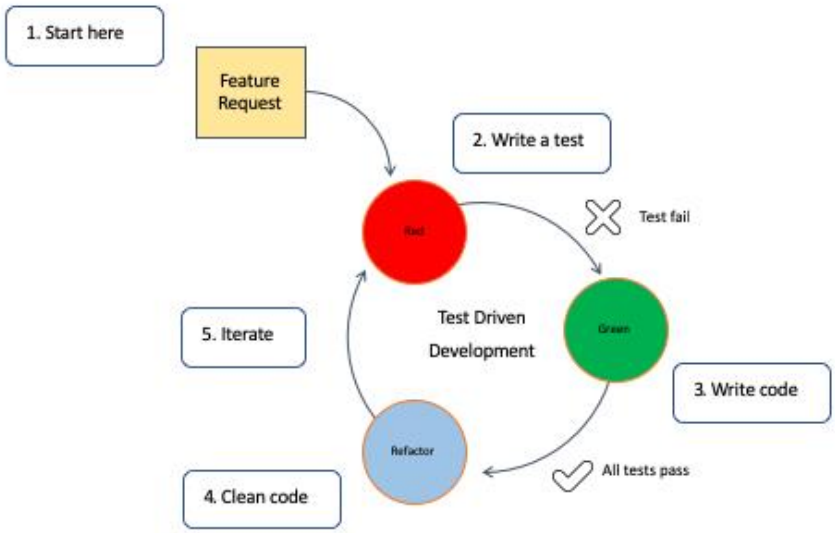


Obrázek č. 24: Agilní životní cyklus [51]

V praxi se můžeme setkat s těmito agilními modely:

- Scrum – nejpoužívanější agilní metodika. Rozděluje vývoj napříč třemi rolemi: scrum master, vývojářský tým a vlastník produktu. Práce je prováděna v tak zvaných sprintech, které trvají dva až čtyři týdny. Jedna z hlavních charakteristik je vydávání produkčních verzí produktu na konci každého sprintu. [52]
- Extrémní programování (XP) – metodika, jejichž cíl je vyvíjet kvalitnější software. Toho dosahuje pomocí dodržování různých (extrémních) praktik. Jako třeba častý refaktoring a revize kódu. Dále je prosazována praktika testově řízeného vývoje. [52]
    - Testově řízený vývoj (TDD) – postup vývoje založeném na opakovaném testování. Jednotlivé iterace jsou rozděleny dle potřebných funkcionalit. Na

rozdíl od klasického vývoje je nejprve napsán test. Tento test by měl logicky neuspět. Následně se napíše kód splňující test. Po refaktoringu tohoto kódu přichází další iterace. [45]



Obrázek č. 25: Testově řízený vývoj [53]

- Kanban – v porovnáním s extrémním programováním se tento model orientuje spíše k projektovému managementu než softwarovému vývoji. Vývoj je jednoduše vizualizován kartami na tabuli dle sloupců: zásobník práce, rozpracované úkoly a dokončené úkoly. Počet karet ve zpracování je ekvivalentní maximální kapacitě týmu. Další úkoly nejsou přijímány, dokud se kapacita neuvolní. [52]

4 Vlastní práce

4.1 Myšlenka

Představme si výzkumnou instituci, která se zaměřuje na výzkum myší. V teoretickém, ale v dnešní době už ne příliš reálném případě tato nejmenovaná instituce nadále používá pro svou správu papírovou variantu, a ještě se neadaptovala době informačních technologií. Převedení procesů z papírové do digitální podoby by mohlo signifikantně optimalizovat workflow a tím snížit vynaložené náklady pro svůj běh.

MyšičkyJedou je webové řešení vhodné pro laboratorní zařízení, které hledají komplexní řešení v oblasti správy, chovu a výzkumu myší.

4.2 Analýza

Před zahájením fáze analýzy je nejprve nutné si zvolit metodu. Tu jsem zvolil objektově-orientovanou analýzu. Důvod je, že se z ní v porovnání s ostatními metody bude nejlépe přecházet na v metodice zvolený programovací jazyk Python, protože je také objektově orientovaný.

4.2.1 Požadavky

V systému bude více rolí uživatelů, každý se svými vlastními požadavky. Tuto skutečnost zachytí diagram případu užití. Nicméně, nehledě na role, zdá se být příliš komplexní popsat celé fungování laboratoře v požadavcích. Proto bych tyto požadavky spíše nazýval procesy, kterými systém musí disponovat, aby mohl činnosti laboratoře alespoň do jisté míry podporovat.
Byly stanoveny následující činnosti systému:

- Chov a výzkum myší
    - Sledování a úprava stavu
    - Přidání a přiřazení klece
    - Aplikování metod výzkumu
    - Definování metod výzkumu
- Správa místností
    - Přidávání/Odstranění

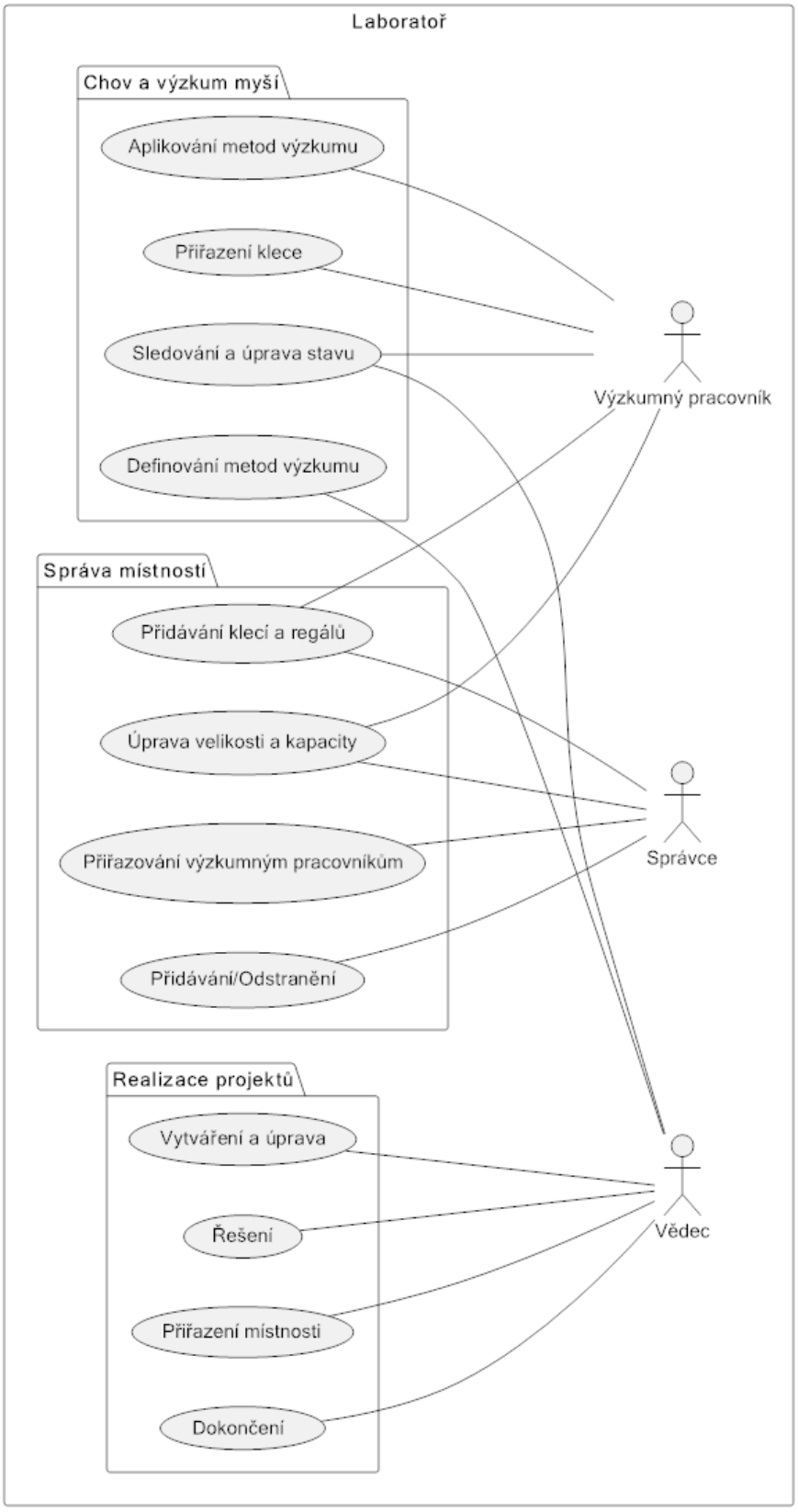
    - Úprava velikosti a kapacity
    - Přiřazování výzkumným pracovníkům
- Realizace projektů
    - Vytvoření a úprava
    - Dokončení
    - Přiřazení místností
    - Řešení

4.2.2 Diagramy

Pro lepší pochopení systému a jeho požadavků je třeba ho namodelovat pomocí grafických nástrojů. K tomuto použiji standard UML (Unified Modeling Language) vzhledem k tomu, že jsme zvolili objektově-orientovanou analýzu. Na trhu existuje mnoho nástrojů pro grafické generování diagramů, které používají notaci UML. Nicméně pro účely své diplomové práce použiji jazyk PlantUML. Generování grafických diagramů v PlantUML probíhá z pouhého prostého textu k čemuž používá programovací jazyk Java. Mezi výhody oproti nástrojům, které se ovládají čistě pomocí grafického rozhraní, patří stabilita, jednodušší zavádění změň či automatické rozložení prvků. [54]

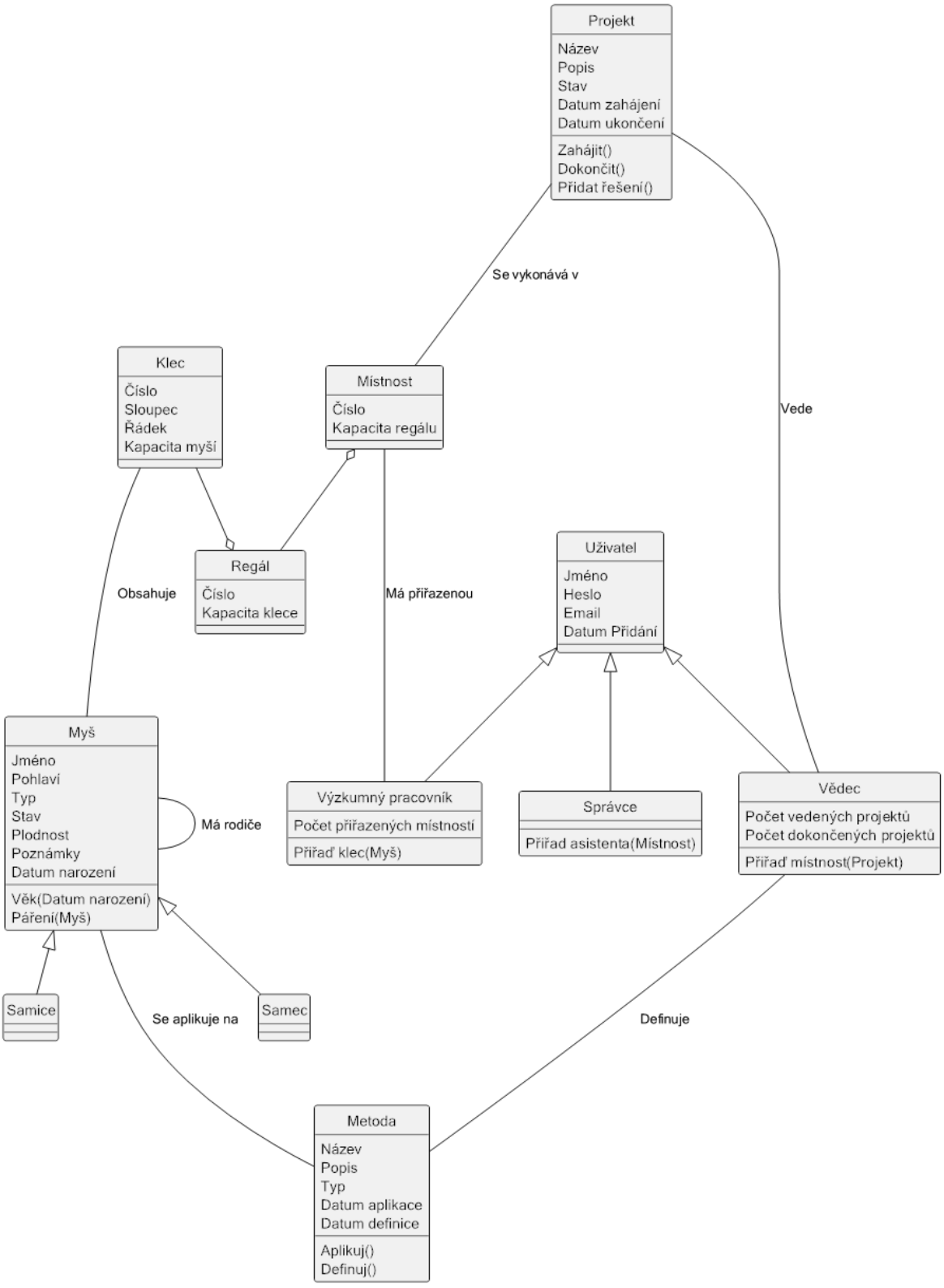
Před modelováním diagramu tříd a interakcí bude nejprve vypracován diagram případu užití. Tímto modelem je potřeba graficky reprezentovat požadavky. Nejprve je, ale nutné k požadavkům určit jednotlivé aktéry v tomto systému:

- Vědec – stará se o vedení projektu a je nadřízený výzkumných pracovníků
- Výzkumný pracovník – je řešitel projektu dle přiřazení místnosti, ve kterých jsou zodpovědní za chov a výzkum myší
- Správci – řeší administrativní činnosti, jako zakládání účtu, přiřazování práv a správu místností.



Obrázek č. 26: Use-case diagram [autor]

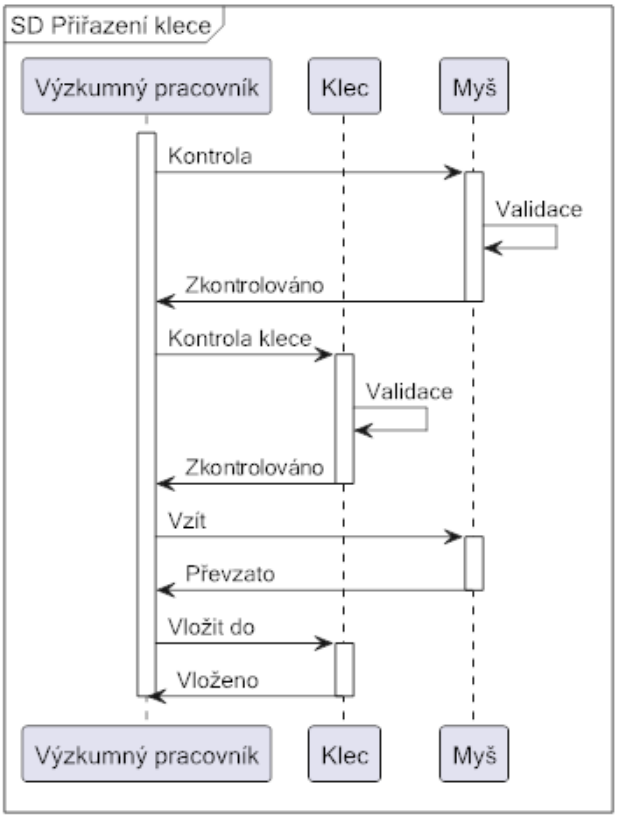
```
@startuml
left to right direction
rectangle "Laboratoř" {
      actor "Vědec" as v
      actor "Správce" as spravce
      actor "Výzkumný pracovník" as vp
      package "Chov a výzkum myší" {
            usecase "Sledování a úprava stavu" as UC1
            UC1 --- v
            UC1 --- vp
            usecase "Přiřazení klece" as UC2
            UC2 --- vp
            usecase "Aplikování metod výzkumu" as UC3
            UC3 --- vp
            usecase "Definování metod výzkumu" as UC4
            UC4 --- v
      }
      package "Správa místností" {
            usecase "Přidávání/Odstranění" as UC5
            UC5 --- spravce
            usecase "Přidávání klecí a regálů" as UC6
            UC6 --- spravce
            UC6 --- vp
            usecase "Úprava velikosti a kapacity" as UC7
            UC7 --- spravce
            UC7 --- vp
            usecase "Přiřazování výzkumným pracovníkům" as UC8
            UC8 --- spravce
      }
      package "Realizace projektů" {
            usecase "Vytváření a úprava" as UC9
            UC9 --- v
            usecase "Dokončení" as UC10
            UC10 --- v
            usecase "Přiřazení místnosti" as UC11
            UC11 --- v
            usecase "Řešení" as UC12
            UC12 --- v

      }
}
@enduml
```



Obrázek č. 27: Class diagram [autor]

```
@startuml
hide circle
class Uživatel {
  Jméno
  Heslo
  Email
  Datum Přidání
}
class Vědec{
  Počet vedených projektů
  Počet dokončených projektů
  Přiřaď místnost(Projekt)
}
class "Výzkumný pracovník"{
  Počet přiřazených místností
  Přiřaď klec(Myš)
}
class Místnost{
      Číslo
      Kapacita regálu
}
class Regál{
      Číslo
      Kapacita klece
}
class Klec{
      Číslo
      Sloupec
      Řádek
      Kapacita myší
}
class Projekt {
  Název
  Popis
  Stav
  Datum zahájení
  Datum ukončení
  Zahájit()
  Dokončit()
  Přidat řešení()
}
class Myš {
  Jméno
  Pohlaví
  Typ
  Stav
  Plodnost
  Poznámky
  Datum narození
  Věk(Datum narození)
  Páření(Myš)
}
class Samec {
```

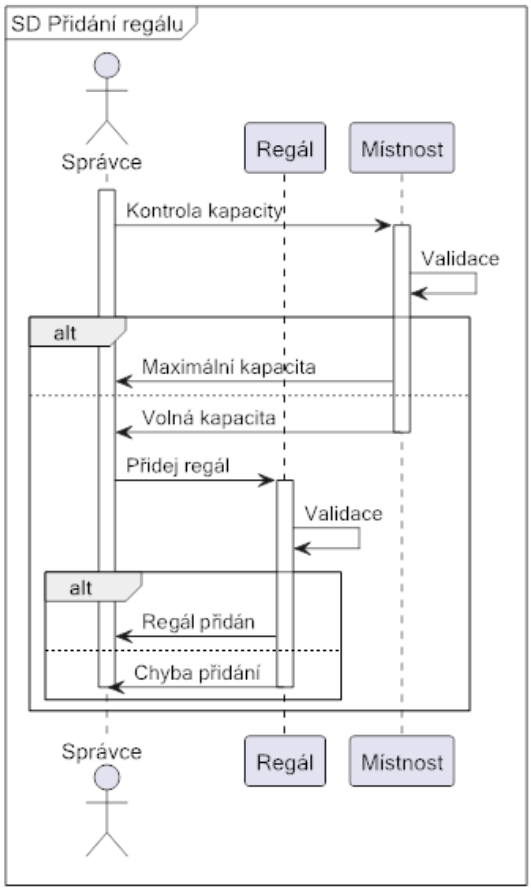
```
}
class Samice {
}
class Metoda{
  Název
  Popis
  Typ
  Datum aplikace
  Datum definice
  Aplikuj()
  Definuj()
}
class Správce{
  Přiřad asistenta(Místnost)
}

Uživatel <|-- Vědec
Uživatel <|-- "Výzkumný pracovník"
Uživatel <|-- Správce
Projekt --- Vědec : Vede
Místnost --- "Výzkumný pracovník" : Má přiřazenou
Místnost o-- Regál
Klec --o Regál
Klec --- Myš : Obsahuje
Myš <|-- Samec
Myš <|-- Samice
Myš --- Myš : Má rodiče
Projekt --- Místnost : Se vykonává v
Myš --- Metoda : Se aplikuje na
Vědec --- Metoda : Definuje
@enduml
```



Obrázek č. 28: Sekvenční diagram přiřazení klece [autor]

```
@startuml
mainframe SD Přiřazení klece
participant "Výzkumný pracovník" as P1
participant Klec as P2
participant Myš as P3

activate P1
P1 -> P3: Kontrola
activate P3
P3 -> P3: Validace
P3 -> P1: Zkontrolováno
deactivate P3
P1 -> P2: Kontrola klece
activate P2
P2 -> P2: Validace
P2 -> P1: Zkontrolováno
deactivate P2
P1 -> P3: Vzít
activate P3
P3 -> P1: Převzato
deactivate P3
P1 -> P2: Vložit do
activate P2
P2 -> P1: Vloženo
deactivate P1
```

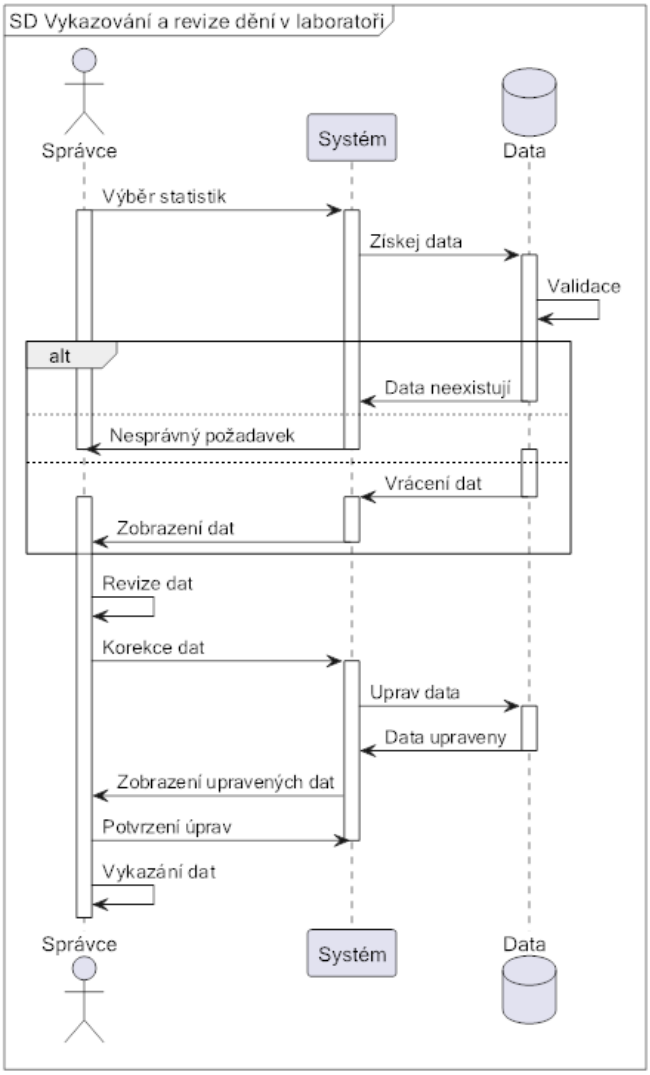
```
deactivate P2

@enduml
```



Obrázek č. 29 Sekvenční diagram přidání regálu [autor]

```
@startuml
mainframe SD Přidání regálu
actor "Správce" as P1
participant Regál as P2
participant Místnost as P3
activate P1
P1 -> P3: Kontrola kapacity
activate P3
P3 -> P3: Validace
alt
P3 -> P1 : Maximální kapacita
else
P3 -> P1: Volná kapacita
deactivate P3
P1 -> P2: Přidej regál
activate P2
P2 -> P2: Validace
```

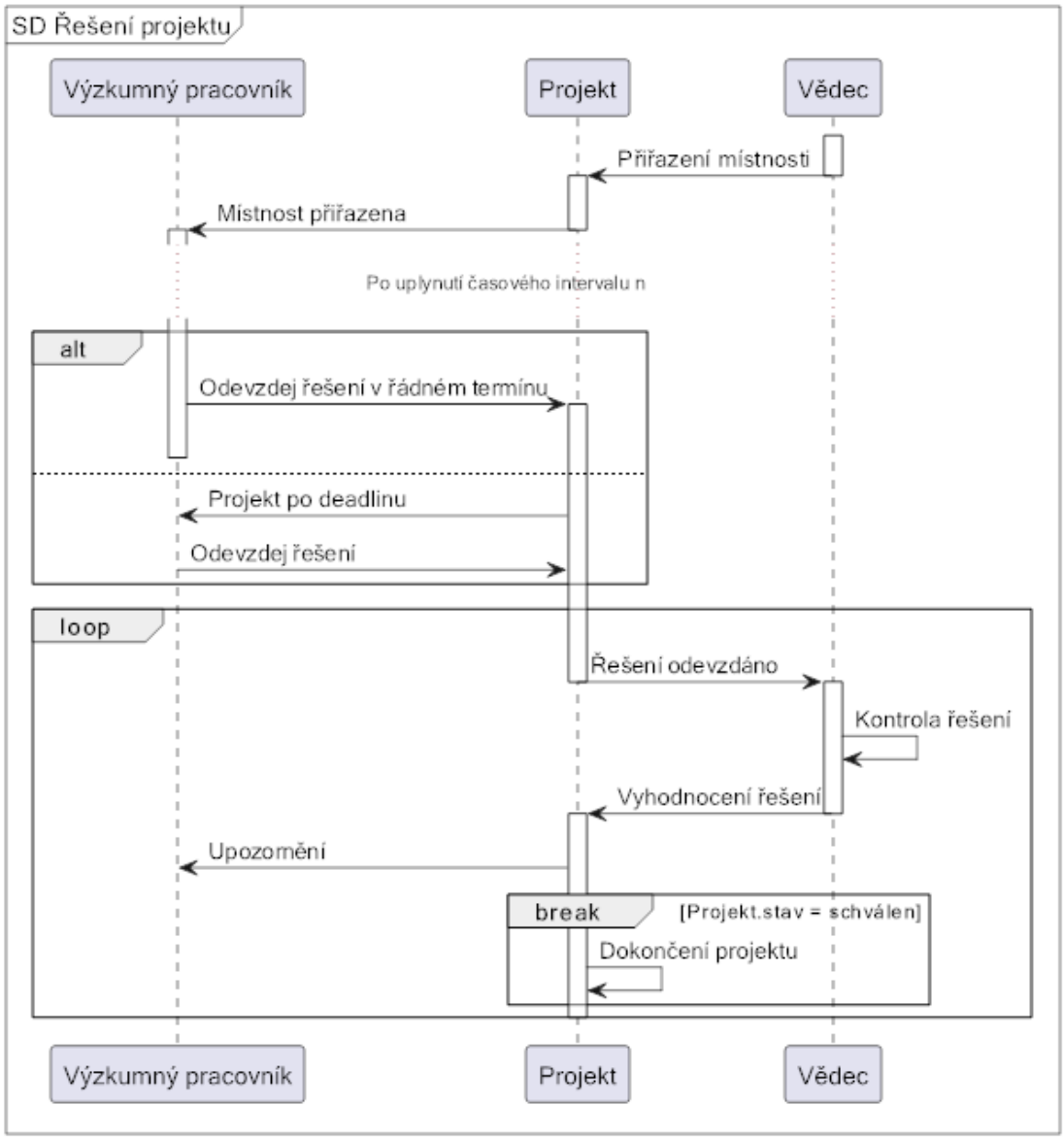
```
alt
P2 -> P1: Regál přidán
else
P2 -> P1: Chyba přidání
deactivate P2
deactivate P1
end
end
@enduml
```



Obrázek č. 30: Sekvenční diagram vykazování a revize dění v laboratoři [autor]

```
@startuml
mainframe SD Řešení projektu
actor Správce
participant Systém
database Data


Správce -> Systém : Výběr statistik
activate Správce
activate Systém
```

```
Systém -> Data: Získej data
activate Data
Data -> Data: Validace

alt
Data -> Systém: Data neexistují
deactivate Data

else
Systém -> Správce : Nesprávný požadavek
deactivate Správce

else
deactivate Systém
activate Data
Data -> Systém: Vrácení dat
deactivate Data
activate Správce
activate Systém
Systém -> Správce: Zobrazení dat
deactivate Systém
end

Správce -> Správce: Revize dat
Správce -> Systém: Korekce dat
activate Systém
Systém -> Data: Uprav data
activate Data
Data -> Systém: Data upraveny
deactivate Data
Systém -> Správce: Zobrazení upravených dat
Správce -> Systém: Potvrzení úprav
deactivate Systém
Správce -> Správce: Vykazání dat
@enduml
```



Obrázek č. 31: Sekvenční diagram řešení projektu [autor]

4.3 Návrh

Při návrhu se bude vycházet z požadavků a diagramů stanovených v části analýzy. Nicméně nejprve bude specifikován výběr technologií a vybrán vhodný systém řízení báze dat (DBMS). V poslední řadě bude vytvořen jednoduchý prototyp uživatelského rozhraní.

4.3.1 Použité technologie

Informační systém bude pro své fungování používat webové technologie. Na straně klienta toto zajistí staticky HTML, CSS a Javascript. Na straně serveru bude použit Python webový framework nesoucí název Django, který je zdarma a jeho kód je otevřený. [55]

Pro usnadnění práce s implementací bude využit kontejnerizační nástroj podman. Jedná se o alternativu populárnějšímu nástroji Docker, který nabízí lepší podporu. Na druhou stranu podman je lepší, pokud jde o zabezpečení, protože nemusí používat pro spuštění root (administrátorského) uživatele. Jsou identické, co se týče uživatelského rozhraní. [56]

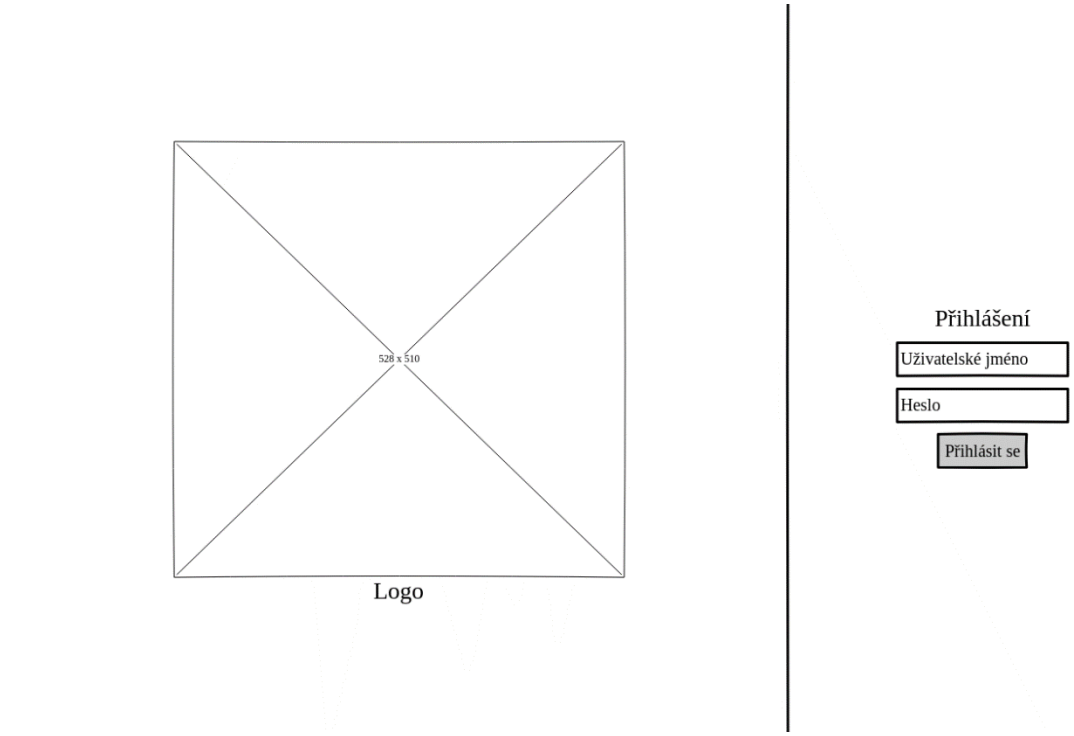
„Django oficiálně podporuje tyto databáze:

- PostgreSQL
- MariaDB
- MySQL
- Oracle
- SQLite“ [55]

Django framework využívá ve výchozím prostředí SQLite databázi. Tato databáze bohatě stačí na testovací účely během vývoje. Nicméně pro produkční prostředí je nedostatečná. Vzhledem k tomu, že informační systém bude uchovávat a zpracovávat velké objemy dat.

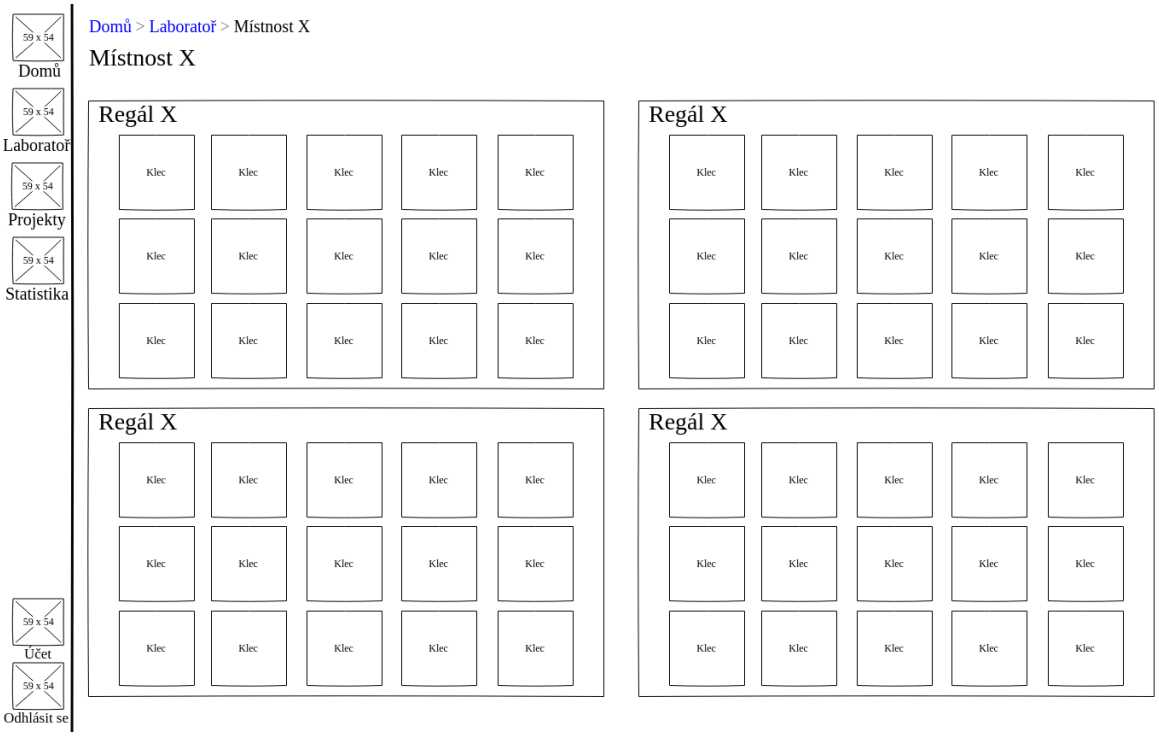
Je důležité podotknout, že díky Objektově relačnímu mapování (ORM), které Django nabízí v základu, nebude nutné s databází v aplikačním kódu interagovat napřímo. To ani během importu databázového schématu. Z tohoto důvodu byla zvolena PostgreSQL, protože pro ni Django nabízí vynikající podporu. Dále má otevřený zdrojový kód, přičemž disponuje pokročilými funkcemi, které lze nalézt v databázích podnikové úrovně jako je Oracle či MySQL. [55]

4.3.2 Wireframy

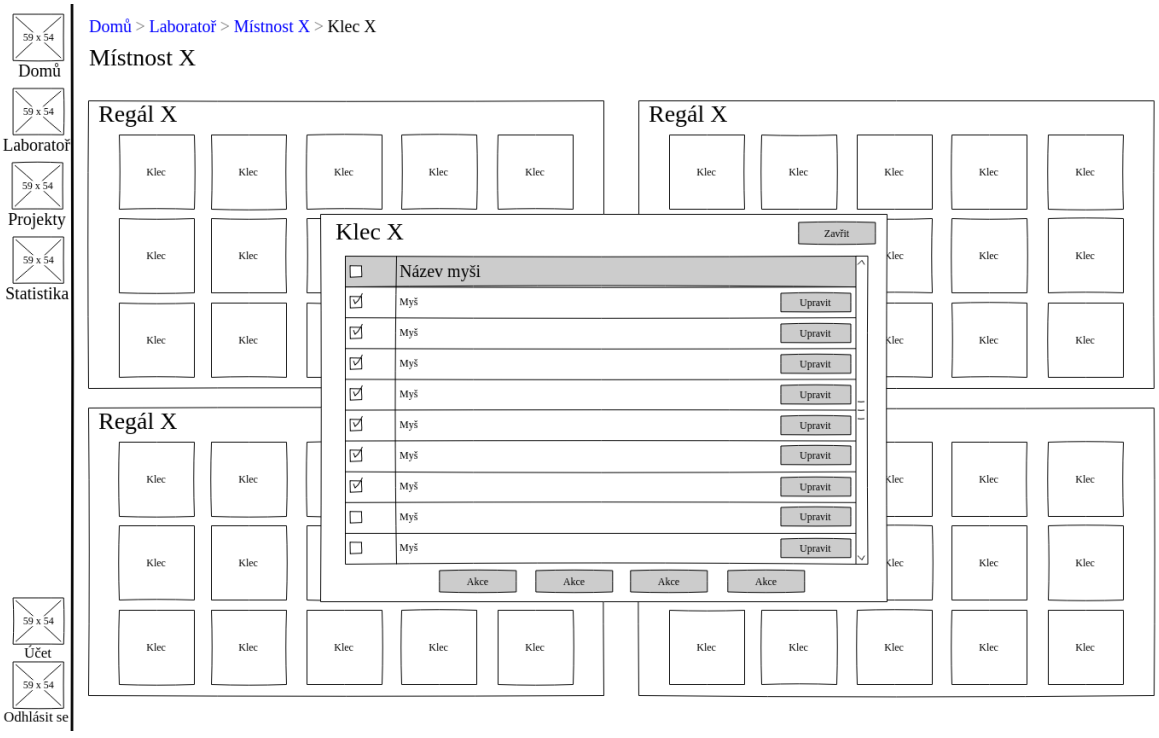


Obrázek č. 32: Wireframe přihlášení [autor]

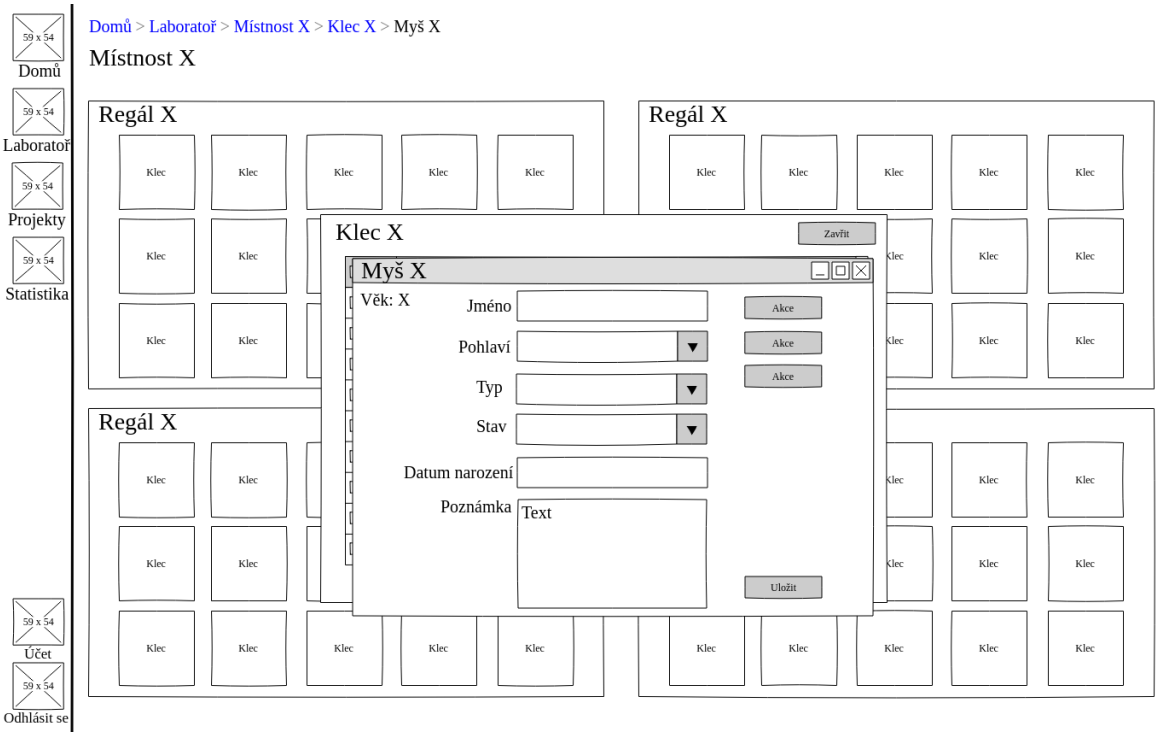
Obrázek č. 33: Wireframe domů [autor]



Obrázek č. 34: Wireframe místnost [autor]

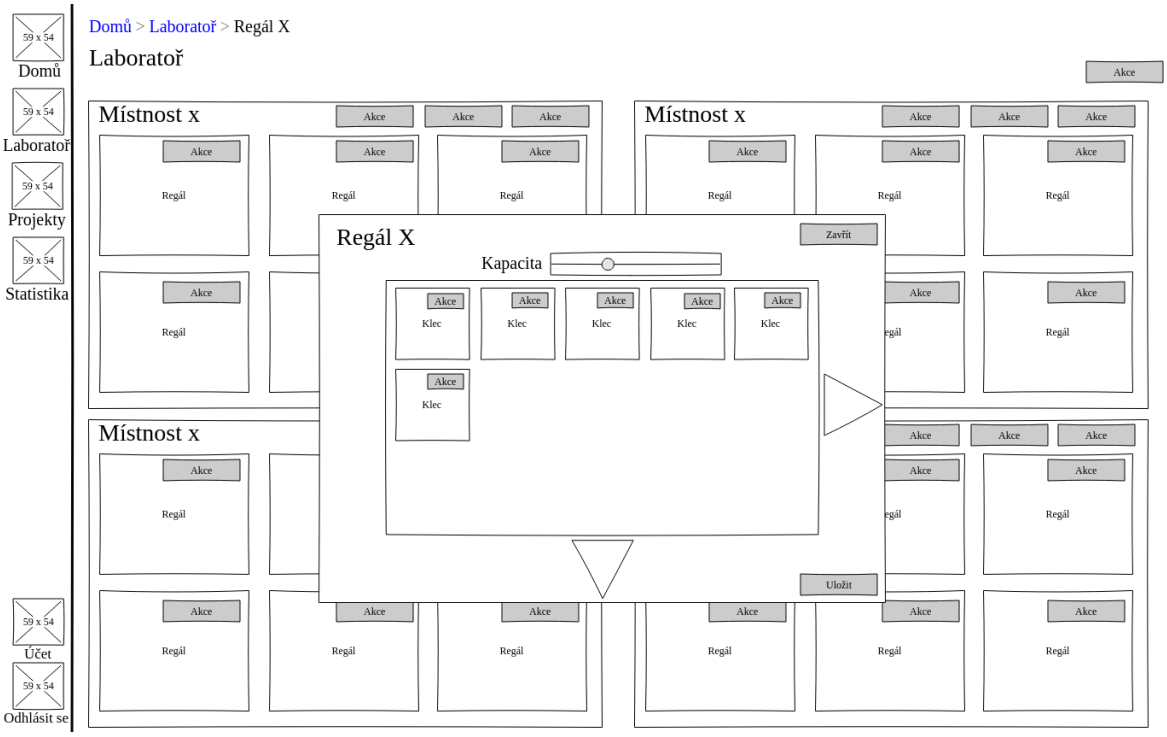


Obrázek č. 35: Wireframe klec [autor]

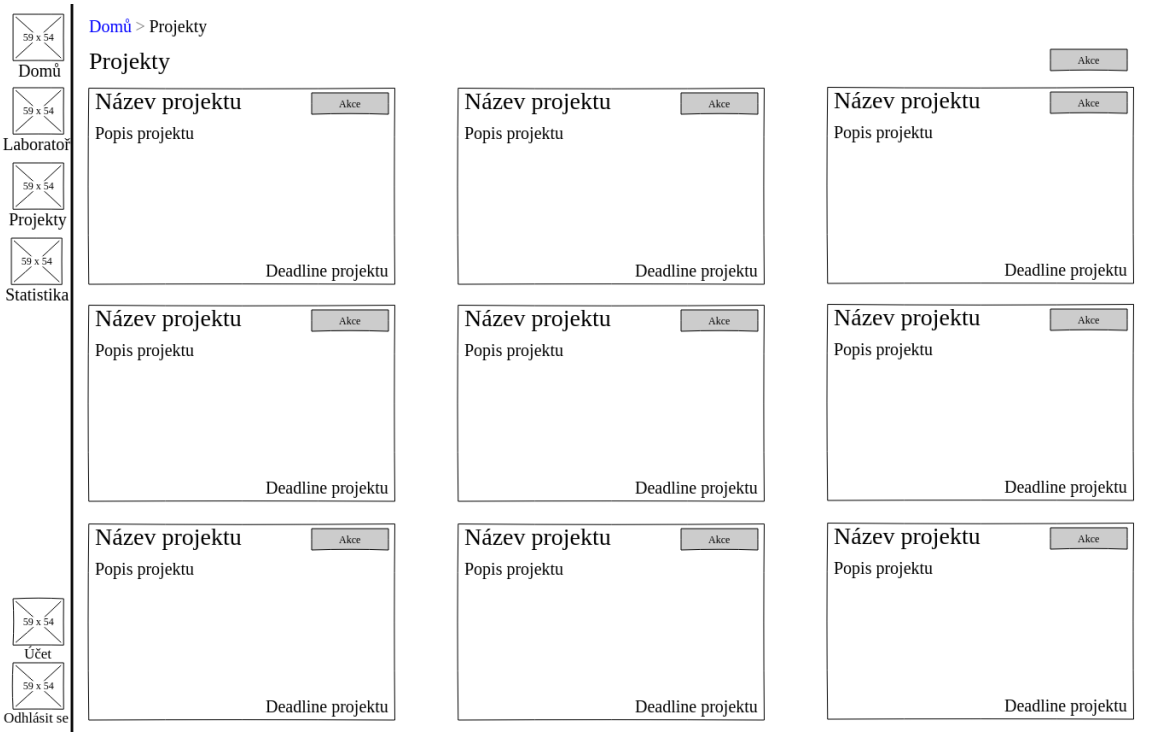


Obrázek č. 36: Wireframe myš [autor]

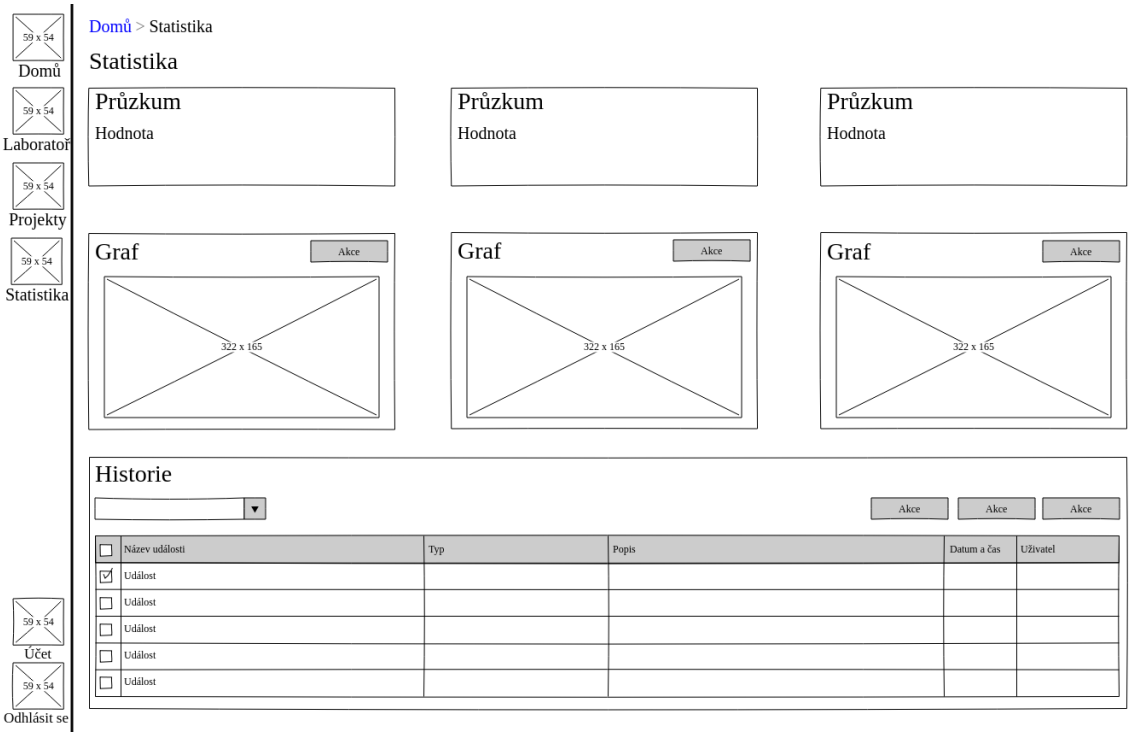
Obrázek č. 37: Wireframe laboratoř [autor]



Obrázek č. 38: Wireframe regál [autor]

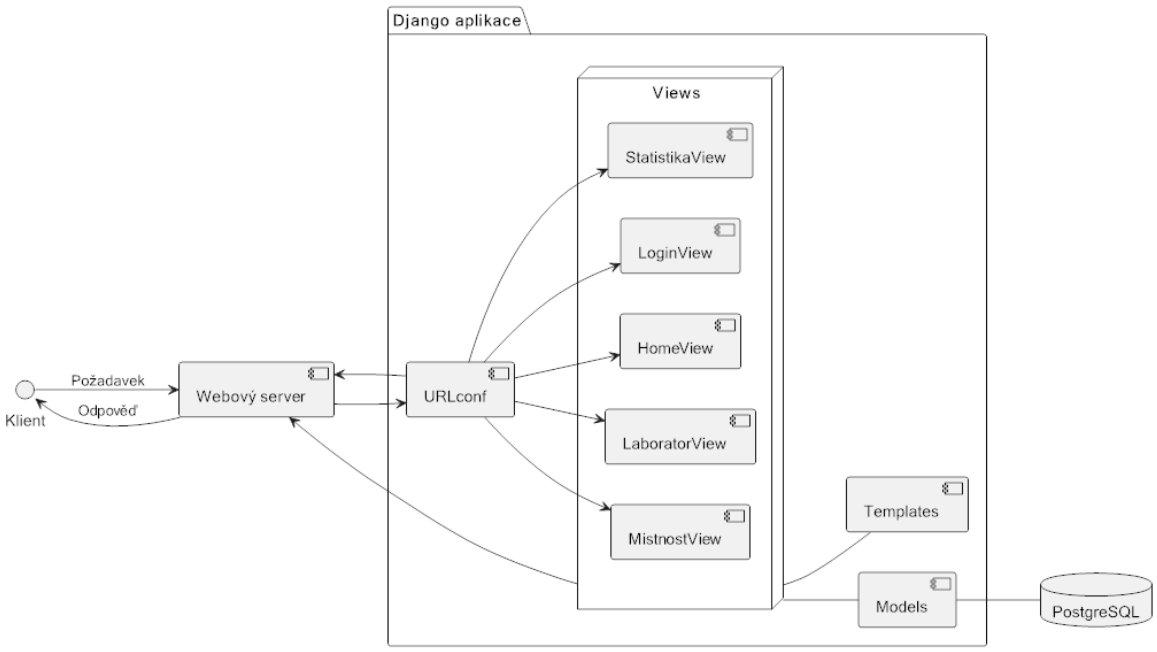


Obrázek č. 39: Wireframe projekty [autor]



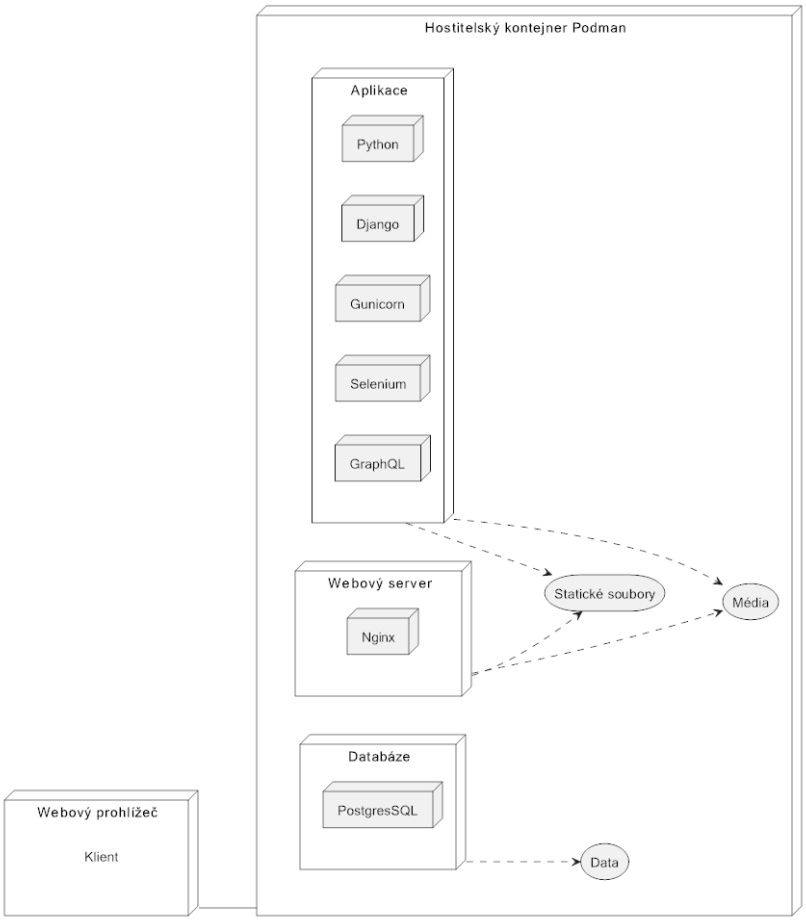
Obrázek č. 40: Wireframe statistika [autor]

4.3.3 Diagramy



Obrázek č. 41: Diagram komponent [autor]

```
@startuml
left to right direction
Klient --> [Webový server] : Požadavek
Klient <-- [Webový server] : Odpověď
package "Django aplikace" {
  [Webový server] --> [URLconf]
  [Webový server] <-- [URLconf]
  node "Views" {
        [URLconf] --> [MistnostView]
        [URLconf] --> [LaboratorView]
        [URLconf] --> [HomeView]
        [URLconf] --> [LoginView]
        [URLconf] --> [StatistikaView]
  }
  Views -- [Models]
  Views -- [Templates]
  Views --> "Webový server"
}
[Models] -- PostgreSQL
database "PostgreSQL" {
  }
}
@enduml
```



Obrázek č. 42 Diagram nasazení [autor]

```
@startuml
left to right direction
node "Webový prohlížeč" {
label Klient
}
node "Hostitelský kontejner Podman" {
      node Aplikace {
            node Python{
            }
            node Django{
            }
            node Gunicorn{
            }
            node Selenium{
            }
            node GraphQL{
            }
      }
      node "Webový server"{
            node Nginx{
            }
      }
      node Databáze{
            node PostgresSQL{
            }
      }
      storage Data
      storage Média
      storage "Statické soubory"
}
Aplikace ..> Média
Aplikace ..> "Statické soubory"
Databáze ..> Data
"Webový server" ...> Média
"Webový server" ..> "Statické soubory"
"Webový prohlížeč" --- "Hostitelský kontejner Podman"
@enduml
```

4.4 Vývoj

Pro každou navrženou funkcionalitu systému bude nejprve napsán funkcionální a jednotkový test. Funkcionální test potvrdí funkčnost z pohledu uživatele. Zatímco jednotkový otestuje část aplikačního kódu dané funkcionality. Na funkcionální testy bude použit nástroj Selenium. Co se týče jednotkových testů bude použita již zabudovaná testovací funkcionalita v Django frameworku. Po ověření toho, že jsou testy neúspěšné bude napsán minimální kód, aby je splnil. Následně bude kód refaktorován do lepší podoby. Tento

proces se bude opakovat dle jednotlivých funkcionalit. Kvůli lepšímu přehledu o změnách bude také použit verzovací systém Git.

4.4.1 Přihlašení

```python
from selenium.webdriver.firefox.webdriver import WebDriver
from django.test import LiveServerTestCase
from django.contrib.auth.models import User
from selenium.webdriver.common.by import By

class LoginTest(LiveServerTestCase):
    def setUp(self):
        self.user = User.objects.create_user(username='test.uživatel', password='Heslo123', email='test@test.com')
        self.user.save()
        self.browser = WebDriver()

    def tearDown(self):
        self.user.delete()
        self.browser.quit()

    def testlogin(self):
        # Uživatel jde na stránku přihlášení
        self.browser.get(('%s%s' % (self.live_server_url, '/prihlaseni')))
        assert "MyšičkyJedou | Přihlášení" in self.browser.title

        # Uživatel vidí dvě vyplňovací pole a tlačítko přihlášení
        user_input = self.browser.find_element(By.ID,'id_username')
        password_input = self.browser.find_element(By.ID,'id_password')
        login_button = self.browser.find_element(By.ID,'login_btn')
        self.assertEqual(login_button.text, 'Přihlásit se')

        # Uživatel vyplní své přihlašovací údaje a kliknutím na tlačítko se přihlásí
        user_input.send_keys('test.uživatel')
        password_input.send_keys('Heslo123')
        login_button.click()

        # Uživatel je nyní na domovské stránce a vidí, že je přihlášen
        assert "MyšičkyJedou | Domů" in self.browser.title
        logged_user = self.browser.find_element(By.ID,'logged_user')
        self.assertEqual(logged_user.text, 'test.uživatel')
```

Obrázek č. 43: Funkcionální test přihlášení [autor]

Na obrázku můžeme vidět funkcionální test na přihlášení uživatele. Dalším krokem je přidání jednotkového testu:

```python
from django.test import TestCase
from django.contrib.auth.models import User
from django.contrib.auth import authenticate

class LoginTestCase(TestCase):
    def setUp(self):
        self.user = User.objects.create_user(username='test.uživatel', password='Heslo123', email='test@test.com')
        self.user.save()
    def tearDown(self):
        self.user.delete()

    def test_spravne(self):
        user = authenticate(username='test.uživatel', password='Heslo123')
        self.assertTrue((user is not None) and user.is_authenticated)

    def test_spatne_jmeno(self):
        user = authenticate(username='spatne', password='Heslo123')
        self.assertFalse(user is not None and user.is_authenticated)

    def test_spatne_heslo(self):
        user = authenticate(username='test.uživatel', password='spatne')
        self.assertFalse(user is not None and user.is_authenticated)
```

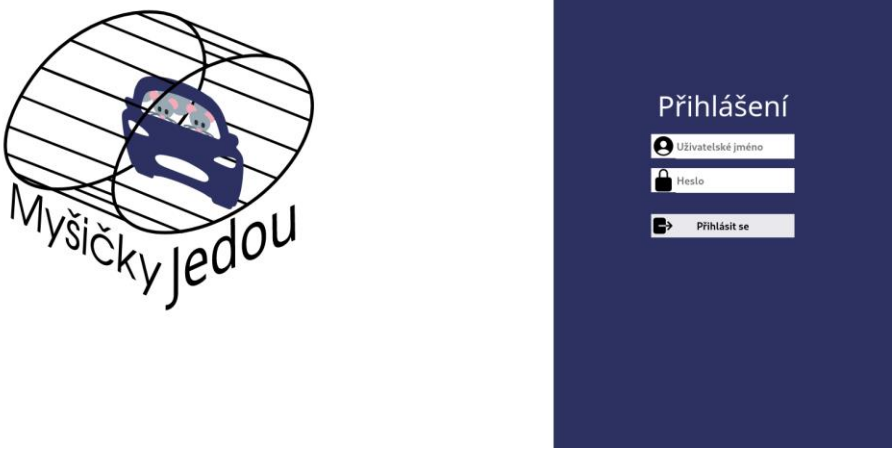
Obrázek č. 44: Jednotkový test přihlášení [autor]

Nyní jsou oba testy neúspěšné, protože zatím nebyl napsán žádný kód. Dále vytvoříme daný view pro přihlášení a domovskou stránku na přesměrování, které nejprve namapujeme na adresu url.

```python
from django.shortcuts import render, redirect
from django.http import HttpResponse
from django.contrib.auth import authenticate, login, logout
from .forms import LoginForm

def prihlaseni(request):
    if request.method == 'POST':
        form = LoginForm(request.POST)
        if form.is_valid():
            username = form.cleaned_data['username']
            password = form.cleaned_data['password']
        user = authenticate(request, username=username, password=password)
        if user:
            login(request, user)
            return redirect('home')
    else:
        form = LoginForm()
    return render(request, 'website/login.html', {'form': form})

def home(request):
    return render(request, 'website/home.html')
```

Obrázek č. 45: View přihlášení [autor]

Oba testy jsou nyní splněné. Nicméně je potřeba provést refaktoring. Vzhled stránek je pouze základní a kódu chybí úprava.



Obrázek č. 46: Stránka přihlášení [autor]

4.4.2 Domovská stránka

V předchozí iteraci, kde byla programováno přihlašování existovala domovská stránka pouze v základní verzi, aby se mělo kam po přihlášení přesměrovat a otestovat, zda přihlášení proběhlo úspěšně. Nyní bude z tohoto vycházeno a přidáno otestování odhlášení a dalších funkcionalit domovské stránky.

```python
class HomeTest(LiveServerTestCase):
    def setUp(self):
        self.user = User.objects.create_user(username='test.uživatel', password='Heslo123', email='test@test.com')
        self.user.save()
        self.browser = WebDriver()
        self.client.login(username='test.uživatel', password='Heslo123')

    def tearDown(self):
        self.user.delete()
        self.browser.quit()

    def testHome(self):
        ## Přihlášený uživatel je odkázán na domovskou stránku
        self.browser.get(self.live_server_url)
        assert "MyšičkyJedou | Domů" in self.browser.title
        ## Uživatel vidí navbar
        nav = self.browser.find_element(By.CLASS,'navbar')
        ## Uživatel vidí drobečkovou navigaci
        breadcrumb = self.browser.find_element(By.CLASS,'breadcrumbnav')
        ## Uživatel vidí novinky
        news_header = self.browser.find_element(By.CLASS,'header_news')
        ## Uživatel vidí odhlašovací tlačítko a klikne na něj, nyní je odkázán zpět na přihlašovací obrazovku
        logout_button = self.browser.find_element(By.ID,'logout_btn')
        logout_button.click()
        assert "MyšičkyJedou | Přihlášení" in self.browser.title
```

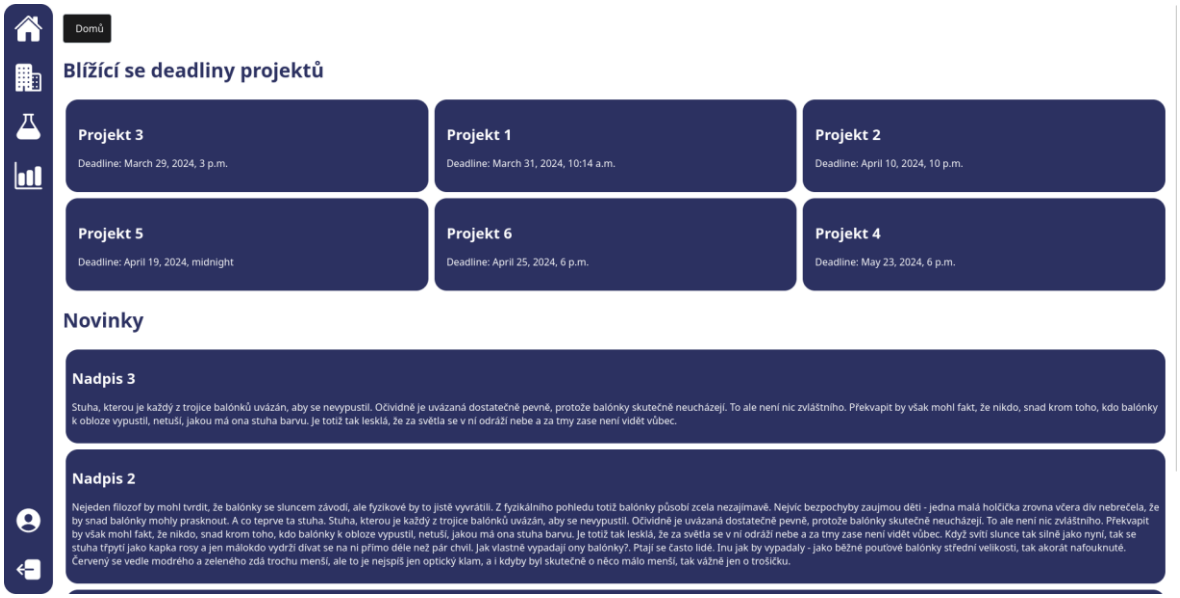
Obrázek č. 47: Funkcionální test domovské stránky [autor]

```python
from django.test import TestCase
from django.contrib.auth.models import User
from django.contrib.auth import authenticate

class HomeTestCase(TestCase):
    def setUp(self):
        self.user = User.objects.create_user(username='test.uživatel', password='Heslo123', email='test@test.com')
        self.user.save()
        self.client.login(username='test.uživatel', password='Heslo123')

    def tearDown(self):
        self.user.delete()

    def test_logout(self):
        self.client.logout()
        response = self.client.get('/')
        self.assertFalse(self.user.is_authenticated)

    def test_template(self):
        response = self.client.get('/')
        self.assertTemplateUsed(response, 'website/home.html')
```

Obrázek č. 49: Jednotkový test domovské obrazovky [autor]

```python
def home(request):
    if request.user.is_authenticated:
        articles_list = Article.objects.order_by('-created')
        projects_list = Project.objects.order_by('end_date')
        return render(request, 'website/home.html', {'articles': articles_list, 'projects':projects_list})
    else:
        return redirect('prihlaseni')
```

Obrázek č. 50: View domovské obrazovky [autor]



Obrázek č. 51: Domovská obrazovka [autor]

4.4.3 Laboratoř

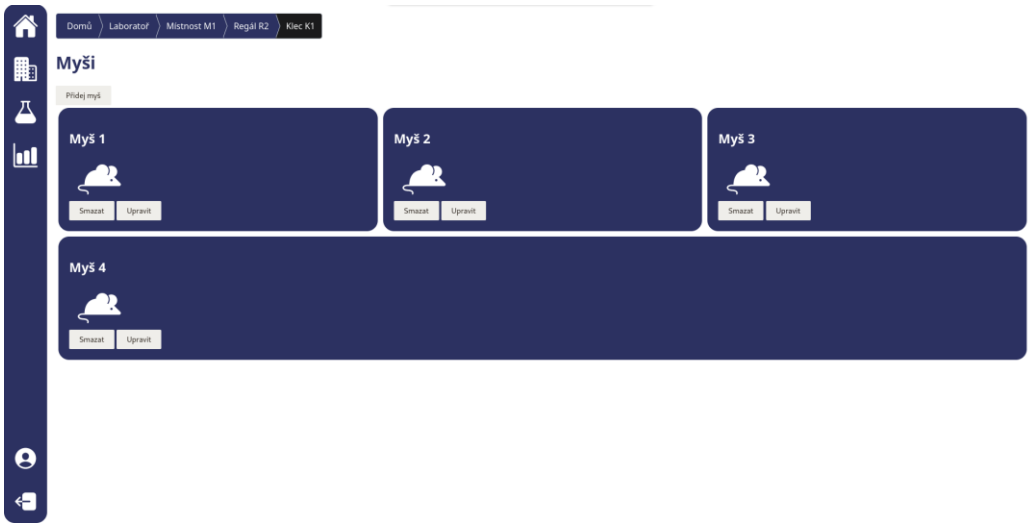
```python
from selenium.webdriver.firefox.webdriver import WebDriver
from django.test import LiveServerTestCase
from django.contrib.auth.models import User
from selenium.webdriver.common.by import By

class LaboratorTest(LiveServerTestCase):
    def setUp(self):
        self.user = User.objects.create_user(username='test.uživatel', password='Heslo123', email='test@test.com')
        self.user.save()
        self.browser = WebDriver()
        self.client.login(username='test.uživatel', password='Heslo123')

    def tearDown(self):
        self.user.delete()
        self.browser.quit()

    def testLab(self):
        ## Přihlášený uživatel jde na stránku laboratoře
        self.browser.get(('%s%s' % (self.live_server_url, '/laborator')))
        ## V drobečkové navigaci a nadpisu uživatel vidí aktivní laboratoř
        breadcrumb = self.browser.find_element(By.CLASS,"breadcrumbnav").find_element(By.CLASS,"is-active")
        self.assertEqual(breadcrumb.text, 'Laboratoř')
        assert "MyšičkyJedou | Laboratoř" in self.browser.title
        ## Uživatel vidí tlačítko na vytvoření, odstranění, přiřazení místnosti a úpravu regálu
        add_room_btn = self.browser.find_element(By.ID,'add_room_btn')
        self.assertEqual(add_room_btn.get_attribute("value"), 'Přidat místnost')
        delete_room_btn = self.browser.find_element(By.ID,'delete_room_btn')
        self.assertEqual(delete_room_btn.get_attribute("value"), 'Odstranit místnost')
        assign_room_btn = self.browser.find_element(By.ID,'assign_room_btn')
        self.assertEqual(assign_room_btn.get_attribute("value"), 'Přiřadit místnost')
        edit_rack_btn = self.browser.find_element(By.ID,'edit_rack_btn')
        self.assertEqual(edit_rack_btn.get_attribute("value"), 'Editovat regál')
```

Obrázek č. 52: Funkcionální test laboratoře [autor]

```python
from django.test import TestCase
from django.contrib.auth.models import User
from django.contrib.auth import authenticate
from .models import Room, Rack, Cage, Mouse

class LaboratorTestCase(TestCase):
    def setUp(self):
        room1 = Room.objects.create(name="Testovaci mistnost", rack_capacity=3)
        rack1 = Rack.objects.create(name="Testovaci regál", cage_capacity=5)
        cage1 = Cage.objects.create(name="Testovaci klec", mouse_capacity=1)
        mouse1 = Mouse.objects.create(name="Testovaci myš")

    def tearDown(self):
        room1.delete()

    def test_template(self):
        response = self.client.get('/laborator')
        self.assertTemplateUsed(response, 'website/laborator.html')

    def test_assign(self):
        rack1.room=room1
        rack1.save()
        cage1.rack=rack1
        cage1.save()
        mouse1.cage=cage1
        cage.save()
```

Obrázek č. 53: Jednotkový test laboratoře [autor]

V rámci vývoje modulu laboratoře jsem zjistil, že je možné využít znovupoužitelnost kódu, který Django nabízí a vzhledem k tomu, že místnosti, regály a klece sdílí téměř stejné vlastnosti mohou používat stejný template k vytváření, mazání a editaci. Tento přístup mi značně urychlil čas potřebný pro vývoj.



Obrázek č. 54: Stránka laboratoře [autor]

4.4.4 Projekty

```python
class ProjektyTest(LiveServerTestCase):
    def setUp(self):
        self.user = User.objects.create_user(username='test.uživatel', password='Heslo123', email='test@test.com')
        self.user.save()
        self.room = Room(name="R1", rack_capacity=10)
        self.room.save()
        self.project = Project(name="P1", desc="Text", room=self.room, start_date=datetime.date.today(), end_date="31. března 2024")
        self.project.save()
        self.browser = WebDriver()
        self.client.login(username='test.uživatel', password='Heslo123')

        def tearDown(self):
            self.user.delete()
            self.browser.quit()

    def testProjects(self):
        ## Přihlášený uživatel jde na stránku projektů
        self.browser.get(('%s%s' % (self.live_server_url, '/projekty')))
        assert "MyšičkyJedou | Projekty" in self.browser.title
        ## Uživatel vidí projekt a jeho status
        header_project = self.browser.find_element(By.CLASS,'header_project')
        projects = self.browser.find_element(By.CLASS,'projects_wrapper')
        projectStatus = self.browser.find_element(By.CLASS,'projectStatus')
        stavy = set(["Nový", "Probíhá", "Čeká na schválení řešení", "Opožděno", "Dokončeno"])
        assertIn (rojectStatus.get_attribute("value"), stavy)
```

Obrázek č. 55: Funkcionální test projektů [autor]

```python
class ProjectTestCase(TestCase):
    def setUp(self):
        self.room = Room(name="Testovací místnost", rack_capacity=3)
        room.save()
        self.project = Project(name="P1", desc="Text", room=self.room, start_date=datetime.date.today(), end_date="21. září 2020")
        project.save()

    def tearDown(self):
        room1.delete()

    def test_template(self):
        response = self.client.get('/projekty')
        self.assertTemplateUsed(response, 'website/projekty.html')

    def test_status_change(self):
        self.project.end_date="31. března 2024"
        self.project.save()
        assertEqual(self.project.isProjectLate(), True)

    def test_reassign(self):
        self.room2 = Room(name="Testovací místnost2", rack_capacity=5)
        self.room2.save()
        self.project.room =self.room2
        self.project.save()
        assertEqual(self.room2, self.project.room)
```

Obrázek č. 56: Jednotkový test projektů [autor]

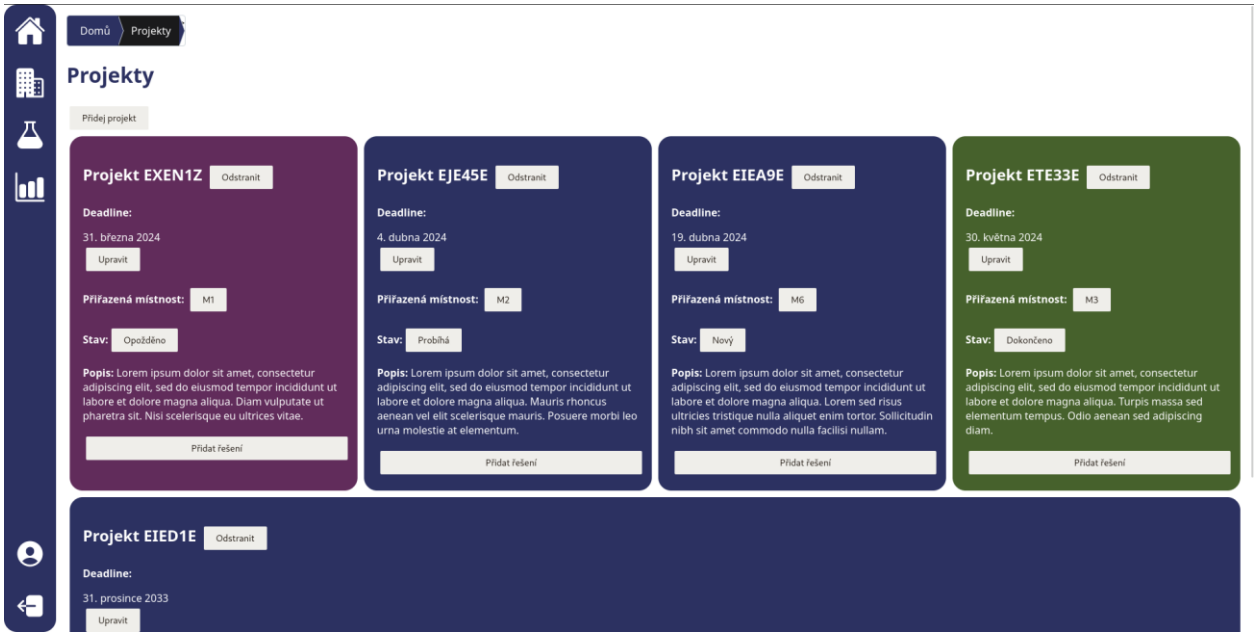
Změna a možné stavy projekty byly naprogramovány na backendové části. Kdežto na front endové části jsem při refactoringu využil dynamického chování javascriptu. K šabloně byl přidán malý modul, který mění barvu pomocí třídy kaskádových stylů na základě stavu projektu.

```python
class Project(models.Model):
    class StatusChoice(models.TextChoices):
        NEW = u'Nový', 'N'
        IN_PROGRESS = u'Probíhá', 'P'
        WAITING = u'Čeká na schválení řešení', 'W'
        LATE = u'Opožděno', 'W'
        FINISHED = u'Dokončeno', 'F'
    status = models.CharField(max_length=24,null=True,choices=StatusChoice.choices,default=StatusChoice.NEW)
    name = models.CharField(max_length=250)
    desc = models.TextField()
    room = models.ForeignKey(Room, on_delete=models.SET_NULL, null=True)
    start_date = models.DateTimeField(auto_now_add=True)
    end_date = models.DateField()

    def __str__(self):
        return self.name

    def isProjectLate(self):
        if self.end_date < timezone.now().date() or self.end_date == timezone.now().date():
            return True
        else:
            return False
```

Obrázek č. 57: ORM Model projektu [autor]

```
{% extends "website/base.html" %}
{% load static %}
{% block title %}MyšičkyJedou | Projekty{% endblock %}
{% block js %}
function paintProjects() {
    document.querySelectorAll("#projectStatus").forEach((Deadline) => {
        if (Deadline.innerHTML=="Opožděno"){
            Deadline.closest(".project").classList.add("late_project");

        }
        if (Deadline.innerHTML=="Dokončeno"){
            Deadline.closest(".project").classList.add("finished_project");

        }
    });
}
window.onload = paintProjects;
{% endblock %}
{% block main %}
    <nav class="breadcrumbnav">
        <a href="/" class="breadcrumbnav_item">Domů</a>
        <a href="/projekty" class="breadcrumbnav_item is-active">Projekty</a>>
    </nav>
    <div class="header_projects">
        <h1>Projekty</h1>
        <a href="/pridejprojekt"><button>Přidej projekt</button></a>
        <div class="projects_wrapper">
            {% if projects %}
                {% for project in projects %}
                    <div class="project">
                        <h2>{{ project.name }} <a href="/odstranitprojekt/{{ project.id }}"><button>Odstranit</button></a></h2>
                        <p><strong>Deadline:</strong> <div class="project_deadline">{{ project.end_date}}</div><a href="/upravit/{{ project.id }}"><button>Upravit</button></a></p>
                        <p><strong>Přiřazená místnost:</strong> <a href="/mistnost/{{ project.room.id }}"><button>{{project.room.name}}</button></a></p>
                        <p><strong>Stav:</strong><a><button id="projectStatus">{{ project.status }}</button></a>{{form}}</p>
                        <p><strong>Popis:</strong> {{ project.desc }}</p>
                        <a class=edit_side href="/pridatreseni/{{ project.id }}"><button>Přidat řešení</button></a>
                    </div>
                    {% if forloop.counter|divisibleby:4 %}<div class="break"></div>{% endif %}
                {% endfor %}
            {% else %}
                <div class="project">
                    <h2>Žádné projekty nenalezeny</h2>
                </div>
            {% endif %}
        </div>
    </div>
{% endblock %}
```

Obrázek č. 58: Django HTML/Javascript šablona[autor]



Obrázek č. 59: Stránka projektů [autor]

4.4.6 Role

Role se dají implementovat vícero způsoby. Například vytvořením abstraktního modelu třídy uživatele. Nicméně jak vyšlo najevo praktičtější je využit již zabudovanou Django funkcionalitu autentizačního systému. Dále byly redukovány jednotlivé view přidáním dekorátorů.

```
class RoleTest(LiveServerTestCase):
    def setUp(self):
        self.user = User.objects.create_user(username='test.uživatel', password='Heslo123', email='test@test.com')
        self.user.role = Role.SCIENTIST
        self.client.login(username='test.uživatel', password='Heslo123')

        def tearDown(self):
            self.user.delete()
            self.browser.quit()

    def testRoom(self):
        ## Přihlášený vědec jde na stránku místností
        self.browser.get(('%s%s' % (self.live_server_url, '/laborator')))
        assert "MyšičkyJedou | Laboratoř" in self.browser.title
        ## Vědec nevidí tlačítko na vytvoření, odstranění, přiřazení místnosti a úpravu regálu
        add_room_btn = self.browser.find_element(By.ID,'add_room_btn')
        self.assertNotEqual(add_room_btn.get_attribute("value"), 'Přidat místnost')
        delete_room_btn = self.browser.find_element(By.ID,'delete_room_btn')
        self.assertNotEqual1(delete_room_btn.get_attribute("value"), 'Odstranit místnost')
        assign_room_btn = self.browser.find_element(By.ID,'assign_room_btn')
        self.assertNotEqual(assign_room_btn.get_attribute("value"), 'Přiřadit místnost')
        edit_rack_btn = self.browser.find_element(By.ID,'edit_rack_btn')
        self.assertNotEqual(edit_rack_btn.get_attribute("value"), 'Editovat regál')
```

Obrázek č. 60: Funkcionální test rolí [autor]

```
class RoleTestCase(TestCase):
    def setUp(self):
        self.user = User.objects.create_user(username='test.uživatel', password='Heslo123', email='test@test.com')

    def tearDown(self):
        self.user.delete()

    def test_role_change(self):
        self.user.role=Role.SCIENTIST
        self.user.save()
        assertEqual(Role.SCIENTIST,  self.user.role)
```

Obrázek č. 61: Jednotkový test rolí [autor]

```
from django.contrib.auth.decorators import login_required
from django.shortcuts import render
from django.urls import reverse
import datetime


@login_required
def mistnost(request, id):
    room = Room.objects.get(id=id)
    try:
        rack_list = Rack.objects.filter(room=room)
    except Rack.DoesNotExist:
        rack_list= None
        return render(request, 'website/mistnost.html', {'room': room, 'racks': rack_list})

def regal(request, id):
    if request.user.is_authenticated:
        rack = Rack.objects.get(id=id)
        try:
            cage_list = Cage.objects.filter(rack=rack)
        except Cage.DoesNotExist:
            cage_list= None
        return render(request, 'website/regal.html', {'room': rack.room, 'rack': rack, 'cages': cage_list})
    else:
        return redirect('prihlaseni')
```

Obrázek č. 62: Porovnání refaktoringu modelů [autor]

4.5 Implementace

Pro implementaci byl použit kontejnerizační nástroj podman. Tento nástroj pracuje s dvěma typy soubory. První je tak zvaný „container file“. Tento soubor bude používat čistě pro kontejner aplikace ke spuštění. Druhý se nazývá compose a obsahuje nastavení jednotlivých kontejnerů, které budou běžet. Konkrétně kromě aplikace to je PostgresSQL databáze a webový server nginx. Po nastavení konfiguračních souborů stačí spustit příkaz: „podman-compose up“ ve složce s compose souborem a systém je připraven k používání.

```
FROM python:3.11.3-alpine

WORKDIR /usr/src/app

ENV PYTHONDONTWRITEBYTECODE 1
ENV PYTHONUNBUFFERED 1

RUN pip install --upgrade pip
COPY requirements.txt .



RUN pip install -r requirements.txt
COPY . .

EXPOSE 8000

CMD ["python", "manage.py", "runserver", "0.0.0.0:8000"]
```

Obrázek č. 63: Containerfile [autor]

```
version: '3.9'

services:
  db:
    image: postgres:latest
    volumes:
    - postgres_data:/var/lib/postgresql/data/
  environment:
    MYSQL_ROOT_PASSWORD: ${MYSQL_ROOT_PASSWORD}
    MYSQL_DATABASE: ${MYSQL_DATABASE}
    MYSQL_USER: ${MYSQL_USER}
    MYSQL_PASSWORD: ${MYSQL_PASSWORD}
  web:
    build: ./app
    volumes:
    - static_volume:/home/app/web/staticfiles
    - media_volume:/home/app/web/mediafiles
    env_file:
    - .dev.env
    expose:
        - "8000"
    depends_on:
            - db
  nginx:
    image: nginx:latest
    volumes:
        - static_volume:/home/app/web/staticfiles
        - media_volume:/home/app/web/mediafiles
    ports:
      - 1337:80
    depends_on:
      - web
volumes:
  postgres_data:
  static_volume:
  media_volume:
```

Obrázek č. 64: Composefile [autor]

5 Závěr

S použitým webovým frameworkem Django se dobře pracovalo. Mnoho funkcionalit má již zabudované, a tak se teoretická část obešla bez většího počtu závislostí na softwarové balíčky třetích stran. Django nabízí řadu užitečných funckionalit. Jednou z nich jsou bezpochyby HTML šablony, které umožňují vnořit i javascriptové moduly. Další skvělou funkcionalitou je podpora ORM, která převádí python třídy na databázové schéma. Dále je třeba vyzdvyhnout i uživatelsky přívětivé správcovské rozhraní.

Během vývoje systému jsem se setkal i s řadou výzev. Tyto výzvy vznikly především podceněním komplexnosti mého návrhu daného systému. Z důvodu těchto výzev byly během vývoje zahozeny některé požadavky. Jedním z nich byl statistický modul, ve kterém by správci laboratoře mohli sledovat na grafech dění v laboratoři. Dále vykazovat a následně exportovat tyto data do dokumentových formátů.

Vývoj značně ovlivnila vybraná vývojová metoda. Touto metodou je testově orientovaný vývoj. Tato metodika sice přináší výhody v podobě produkce kvalitnějšího kódu. Nicméně nutnost psát funkční i jednotkové testy před samotným kódováním podstatně zvýšilo časovou náročnost, tak i komplexitu vývoje.

Navzdory těmto výzvám lze považovat cíle práce za splněné. Literární rešerše byla vypracována na základě odborných literárních zdrojů. Výstup teoretické části práce přispívá k rozvoji základních znalostí o informačních systémech a etapách jeho vývoje. Praktická část ke splnění cíle práce využívá teoretické podklady z literární rešerše ve spojení s dokumentacemi vybraných technologií. Výsledek vývoje poskytuje solidní základ laboratorního systému, který může být na základě zpětné vazby dále zdokonalován tak, aby podporoval další činnosti laboratoře.

6 Seznam použitých zdrojů

1. ŠARMANOVÁ, Jana. Informační systémy a datové sklady. Ostrava: Vysoká škola báňská – Technická univerzita, 2008. ISBN 978-80-248-1500-8.
2. ČERNÝ, Michal. Informační systémy ve vzdělávání: od matrik k sémantickým technologiím a dialogovým systémům pro učení. Brno: Masarykova univerzita, 2016. ISBN 978-80-210-8326-4.
3. Information systems in context – Woo. [online]
   Dostupné z: https://sites.google.com/a/jamesruse.nsw.edu.au/woo/hsc-ipt/2008-09/syllabus/cc/prelim/8-1/information-systems-in-context
4. KLEPPMANN, Martin. Designing data-intensive applications: the big ideas behind reliable, scalable, and maintainable systems. Beijing: O'Reilly, 2017. ISBN 978-1-449-37332-0.
5. SOMMERVILLE, Ian. Software Engineering. 9. vyd. Boston: Pearson, 2010. ISBN 978-0-13-703515-1.
6. OPPENHEIMER, David, Archana GANAPATHI a David PATTERSON. Why Do Internet Services Fail, and What Can Be Done About It? Seattle, 2003. Kalifornská univerzita.
   Dostupné z: http://roc.cs.berkeley.edu/papers/usits03.pdf
7. ABBA, Haruna, Nordin ZAKARIA a Nazleeni HARON. Grid Resource Allocation: A Review. Research Journal of Information Technology. 2012, 4(2), 38-55. ISSN 2041-3114.
8. KUMAR, Rajesh. Key Characteristics of Distributed System : System Design. [online]. Dostupné z: https://medium.com/rtkal/key-characteristics-of-distributed-system-system-design-f3a64d878814
9. MCCONNELL, Steve. Code Complete: A Practical Handbook of Software Construction, Second Edition. Redmond: Microsoft Press, 2004. ISBN 978-0735619678.
10. HUGHES, Bob, ed. Project Management for IT-Related Projects. Second Edition. Swindon: BCS, The Chartered Institute for IT, 2012. ISBN 978-1-78017-119-7.
11. SCHWALBE, Kathy. Information Technology Project Management. Vyd. 9. Boston: Cengage Learning, 2018. ISBN 978-1-337-10135-6.
12. NIETO-RODRIGUEZ, Antonio. Harvard Business Review Project Management Handbook. Boston: Harvard Business Review Press, 2021. ISBN 978-1-64782-125-8.
13. Projektové řízení. [online]
   Dostupné z: https://www.vovcr.cz/odz/ekon/416/page01.html
14. A guide to the project management body of knowledge (PMBOK guide). 6. vyd. Newtown Square, Pa.: Project Management Institute, 2017. ISBN 978-1-62825-184-5.
15. SDLC (Software Development Life Cycle)Phases, Process. What is SDLC. [online] Dostupné z: https://www.numpyninja.com/post/sdlc-software-development-life-cycle-phases-process-what-is-sdlc
16. LANGER, Arthur. Guide to Software Development: Designing and managing the life cycle. Vyd. 2. Londýn: Springer, 2016. ISBN 978-1-4471-6797-6
17. SHELLY, Gary a ROSENBLATT, Harry. Systems Analysis and Design. 9. vyd. Boston: Cengage Learning, 2012 ISBN 978-0-538-48161-8.

18. SATZINGER, John; JACKSON, Robert a BURD, Stephen. Systems Analysis and Design in a Changing World. Vyd. 6. Boston: Cengage Learning,. ISBN 978-1-111-53415-8.
19. O'DOCHERTY, Mike. Object-Oriented Analysis and Design: Understanding System Development with UML 2.0. Chichester: John Wiley & Sons, 2005. ISBN 978-0-470-09240-8
20. WHITTEN, Jeffrey a BENTLEY, Lonnie. Systems Analysis and Design Methods. Vyd. 7.New York: McGraw-Hill/Irwin, 2007. ISBN 978-0-07-305233-5.
21. Diagram datových toků – Informační systém lékárny. Václav Adamec. [online] Dostupné z: https://islekarny.weebly.com/diagram-datovyacutech-tok367.html
22. Sally / PostreSQL – ERD. Petr Bílek. [online] Dostupné z: https://www.sallyx.org/sally/psql/erd.php Diagram datových toků –
23. Lekce 5 – UML – Class diagram. ITnetwork. David Hartinger. [online] Dostupné z: https://www.itnetwork.cz/navrh/uml/uml-class-diagram-tridni-model
24. MARTIN, R. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Londýn: Prentice Hall, 2018. ISBN: 978-0-13-449416-6.
25. RICHARDS, Mark a FORD, Neal. Fundamentals of software architecture: an engineering approach. Beijing: O'Reilly, 2020. ISBN 978-1-492-04345-4.
26. FAIRBANKS, George. Just Enough Software Architecture: A Risk-Driven Approach. Boulder: Marshall & Brainerd, 2010. ISBN 978-0-9846181-0-1
27. Třívrstvá architektura (Three-tier architecture) - ManagementMania.com. [online] Dostupné z: https://managementmania.com/cs/trivrstva-architektura-three-tier-architecture
28. When to use the Pipeline Architecture Style - Airspeed Consulting. [online] Dostupné z: https://airspeed.ca/when-to-use-the-pipeline-architecture-style/
29. What Are The Different Types of Enterprise Software Architectures? AppInventiv. [online] Dostupné z: https://appinventiv.com/blog/choose-best-enterprise-architecture/
30. M. van STEEN a A.S. TANENBAUM. Distributed Systems, 3rd ed., distributed-systems.net, 2017. ISBN 978-15-430573-8-6
31. Primer: Understanding Software and System Architecture - The New Stack. [online] Dostupné z: https://thenewstack.io/primer-understanding-software-and-system-architecture/
32. Advantages of the event-driven architecture pattern - IBM Developer. Grace Jansen a Johanna Saladas. [online] Dostupné z: https://developer.ibm.com/articles/advantages-of-an-event-driven-architecture/
33. Machine Learning-based Orchestration of Containers: A Taxonomy and Future Directions - Scientific Figure on ResearchGate. [online] Dostupné z: https://www.researchgate.net/figure/An-Example-of-the-Microservice-Architecture_fig4_353068829
34. MCCONNELL, Steve. Software Project Survival Guide. Redmond: Microsoft Press, 1998. ISBN 978-1572316218.
35. MARTIN, Robert C. Clean code: a handbook of agile software craftmanship. Robert C. Martin series. Upper Saddle River, NJ: Prentice Hall, 2009. ISBN 978-0-13-235088-4.
36. Ok! So... Showcase | S.O.L.I.D Principles. [online]. Dostupné z: https://okso.app/showcasolid

37. MARTIN, Robert C. Clean code: a handbook of agile software craftmanship. Robert C. Martin series. Upper Saddle River, NJ: Prentice Hall, 2009. ISBN 978-0-13-235088-4.
38. MYERS, Glenford J.; BADGETT, Tom a SANDLER, Corey. The art of software testing. 3rd ed. Hoboken, New Jersey: Wiley, 2012. ISBN 978-1-118-03196-4
39. MAURÍCIO, Aniche. Effective Software Testing: A developer's guide. New York: Manning Publications, 2022. ISBN 9781633439931
40. Co je To Test Automatizace, můžeme testovat pouze frontend? – Tredgate. [online] Dostupné z: https://tredgate.cz/2023/08/15/co-je-to-test-automatizace-muzeme-testovat-pouze-frontend/
41. Comprehensive study of software testing: Categories, levels, techniques, and types - Scientific Figure on ResearchGate. Dostupné z: https://www.researchgate.net/figure/The-Software-Testing-Levels-compared-12_tbl1_337331361
42. TAYNTOR, Christine B. Successful packaged software implementation. Boca Raton: Auerbach Publications, 2006. ISBN 978-0-8493-3410-8
43. VARGA, Ervin. Unraveling Software Maintenance and Evolution. Londýn: Springer, 2017. ISBN 978-3-319-71302-1
44. TRIPATHY, Priyadarshi a NAIK, Kshirasagar. Software evolution and maintenance: A Practitioner's Approach. Hoboken, New Jersey: Wiley, 2015. ISBN 978-1-118-03196-4
45. KNEUPER, Ralf. Software Processes and Life Cycle Models: An Introduction to Modelling, Using and Managing Agile, Plan-Driven and Hybrid Processes. Cham: Springer, 2018. ISBN 978-3-319-98844-3
46. SDLC - Waterfall Model. TutorialsPoint. [online] Dostupné z: https://www.tutorialspoint.com/sdlc/sdlc_waterfall_model.htm
47. GOMAA, Hassan. Software modeling and design: UML, use cases, patterns, and software architectures. New York: Cambridge University Press, 2011. ISBN 978-0-521-76414-8.
48. What is Incremental model- advantages, disadvantages and when to use it?. Tryqa. Dostupné z: https://tryqa.com/what-is-incremental-model-advantages-disadvantages-and-when-to-use-it/
49. Spiral Model – javatpoint. [online] Dostupné z: https://www.javatpoint.com/spiral-model
50. Metody vývoje aplikací. Waterfall, V-model, Inkrementální model. Iquest. [online] Dostupné z: https://blog.iquest.cz/2017/07/metody-vyvoje-aplikaci-waterfall-v.html
51. Agile Development Process in Software Development. Saigon Technology. [online] Dostupné z: https://saigontechnology.com/blog/agile-development-process-in-software-outsourcing
52. STELLMAN, Andrew a GREENE, Jennifer. Learning agile. Sebastopol, CA: O´Reilly, 2015. ISBN 978-1-449-33192-4.
53. 5 steps of test-driven development – IBM Developer. IBM Developer. [online] Dostupné z: https://developer.ibm.com/articles/5-steps-of-test-driven-development/
54. Frequently Asked Questions. PlantUML. [online]. Dostupné z: https://plantuml.com/faq
55. The web framework for perfectionists with deadlines | Django. [online]. Dostupné z: https://www.djangoproject.com/

56. Docker vs Podman: V čem se liší – Ondřej Šika. [online]. Dostupné z: https://ondrej-sika.cz/blog/docker-vs-podman-v-cem-se-lisi/

7 Seznam obrázků a tabulek

7.1 Seznam obrázků

7.2 Seznam tabulek