Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Diplomová práce

Informační systém pro správu garančních prohlídek

Ing. Stanislav Šimeček

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ DIPLOMOVÉ PRÁCE

Ing. Stanislav Šimeček

Systémové inženýrství a informatika
Informatika

Název práce

Informační systém pro správu garančních prohlídek

Název anglicky

Information system for warranty checks management

Cíle práce

Cílem práce je navržení a implementace informačního systému moto prodejny se servisem. Informační systém bude sloužit pro správu garančních kontrol a náhradních dílů pro motocykly a zahraní techniku. Informační systém bude podporovat tříúrovňový přístup: administrátor, vedení a zaměstnanec. Dílčími cíly diplomové práce je analýza požadavků a následný návrh informačního systému.

Metodika

Diplomová práce bude složena ze dvou částí. V první části budou nastudovány a shromážděny informace z vhodných informačních zdrojů. Nasbírané informace budou po té použity jako zdroj informací pro praktickou část diplomové práce.

V praktické části budou zpracovány požadavky na informační systém. Na základě těchto požadavků bude navrhnut a implementován informační systém. Pro implementaci informačního systému bude použit Java framework Spring Boot, ORM Hibernate a ORDBMS PostgreSQL.

Doporučený rozsah práce

50-60 stran

Klíčová slova

Informační systém, Java, Spring Boot, Hibernate, PostgreSQL

Doporučené zdroje informací

BRUCKNER, T. *Tvorba informačních systémů : principy, metodiky, architektury.* Praha: Grada, 2012. ISBN 978-80-247-4153-6.

HALBICH, Č. – BRECHLEROVÁ, D. – ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE. KATEDRA INFORMAČNÍCH TECHNOLOGIÍ. *Bezpečnost informačních systémů : vybrané kapitoly.* V Praze: Česká zemědělská univerzita, Provozně ekonomická fakulta ve vydavatelství Credit, 2003. ISBN 80-213-1090-1.

MERUNKA, V. – PERGL, R. – PÍCKA, M. – ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE. KATEDRA INFORMAČNÍHO INŽENÝRSTVÍ. *Objektově orientovaný přístup v projektování informačních systémů.* V Praze: Česká zemědělská univerzita, Provozně ekonomická fakulta, 2005. ISBN 80-213-1352-8.

MOLNÁR, Z. – ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE. STROJNÍ FAKULTA. *Podnikové informační systémy.* V Praze: České vysoké učení technické, 2009. ISBN 978-80-01-04380-6.

Předběžný termín obhajoby

2020/21 LS – PEF

Vedoucí práce

Ing. Marek Pícka, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 19. 11. 2020

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 19. 11. 2020

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 29. 12. 2020

Čestné prohlášení

Prohlašuji, že svou diplomovou práci "Informační systém pro správu garančních prohlídek" jsem vypracoval(a) 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 29.3.2021 _____

Poděkování

Rád bych touto cestou poděkoval Ing. Marku Píckovi, Ph.D. za odbornou pomoc a vedení diplomové práce.

Informační systém pro správu garančních prohlídek

Abstrakt

Diplomová práce popisuje návrh a implementaci informačního systému pro správu garančních prohlídek. V teoretické části se práce zabývá teoretickými základy, kterých je poté využito v praktické části diplomové práce. Teoretická část popisuje základy objektové analýzy, základy statického a dynamického modelování pomocí jazyka UML a základy vývoje Java aplikací pomocí frameworku Spring Boot.

V praktické části jsou shromážděny požadavky na analyzovaný systém. Na základě těchto požadavků je poté systém analyzován pomocí jazyka UML. V rámci analýzy jsou vytvořeny diagramy tříd, diagramy aktivit a digram případů užití. Na základě analýzy je systém implementován pomocí zvolených technologií. V praktické části jsou představeny vybrané části kódu a jejich vysvětlení. Dále jsou v praktické části představeny vybrané části grafického uživatelského rozhraní aplikace. Zdrojový kód implementovaného informačního systému je přiložen k diplomové práci.

Klíčová slova: Informační systém, UML, Java, Spring Boot, Hibernate, PostgreSQL

Information system for warranty control management

Abstract

Diploma thesis describes design and implementation of information system for managing warranty controls. Theoretical part of diploma thesis explains theoretical basics, which are applied for thesis practical part. Theoretical part explains basics of object analysis, basics of static analysis, dynamic analysis with UML language and basics of development Java application with framework Spring Boot.

In practical part of thesis are summarized analyzed information system requirements. Summarized requirements are applied for system analysis with UML language. According to results of analysis is information system implemented with defined technologies. In practical part of thesis are described selected parts of code with explanations. In practical part of thesis are displayed selected parts of system user interface. Information system source code is attached to diploma thesis.

Keywords: Information system, UML, Java, Spring Boot, Hibernate, PostgreSQL

Obsah

Seznam obrázků

Seznam zdrojových kódů

1 Úvod

Na základě velmi rychlého růstu technické úrovně většiny služeb roste i potřeba uchovávat a pracovat s velkým množstvím informací. Většina firem pro zkvalitnění jejich práce a zkvalitnění komunikace se zákazníky potřebuje efektivně pracovat s informacemi, které uchovávají. Proto je v dnešní době pro firmy téměř nutností vlastnit informační systém, který jejich práci zefektivní a tím společnosti může zajistit vyšší zisk. S rostoucí potřebou firem používat informační systémy roste i potřeba tyto systémy vytvářet kvalitně nejen z pohledu uživatele, ale také z pohledu programátora.

Kvalitní software by měl nejen splňovat definované požadavky na funkčnost, ale také by měl být dobře škálovatelný a rozšiřovatelný. Jelikož poptávka po informačních systémech roste, roste i poptávka po doplněních funkcionalit zákazníkům na míru. Proto by informační systém měl umožňovat snadnou rozšiřovatelnost tak, aby nová funkcionalita neovlivnila stávající funkcionality a tím nesnížila výslednou kvalitu informačního systému, ale na druhou stranu také snížila časovou náročnost pro doplnění nových funkcionalit. Této kvality docílíme především kvalitní analýzou systému, kterou je nutné vytvořit před samotnou implementací systému.

Motohobby je menší moto prodejna se servisem, která poskytuje servis pro mnoho zákazníků, kteří u nich stroje zakoupili a potřebují provádět pravidelné garanční prohlídky strojů pro udržení nároku na záruku. Garanční prohlídky je třeba provádět v různých intervalech v závislosti na druhu stroje a frekvenci používání. Podniku by rád zefektivnil sledování nadcházejících garančních prohlídek strojů a vyhledání strojů, jenž pravidelnou garanční kontrolu nemají. Na základě této situace po dohodě s majitelem společnosti bylo dohodnuto, že autor vytvoří systém, jenž firmě pomůže se správou těchto garančních prohlídek.

Pro vývoj tohoto systému autor zvolil Java framework Spring Boot a databázový systém PostgreSQL.

2 Cíl práce a metodika

2.1 Cíl práce

Diplomová práce Informační systém pro správu garančních prohlídek má za cíl navrhnout a implementovat informační systém podle zadaných požadavků pro moto prodejnu se servisem. V dané prodejně bude informační systém sloužit pro správu garančních prohlídek zahradní techniky, motocyklů a náhradních dílů. Informační systém bude podporovat úrovně přístupu pro administrátora, manažera a zaměstnance.

Dílčím cílem diplomové práce je provést analýzu systému pomocí jazyka UML na základě požadavků na systém. Analýza bude sloužit také pro konzultace informačního systému s majitelem moto prodejny.

Dalším dílčím cílem je implementace informačního systému na základě UML analýzy. Informační systém bude implementován pomocí frameworku Spring Boot a relačního databázového systému PostgreSQL.

Posledním dílčím cílem diplomové práce je implementace systému na základě UML modelu. Systém bude implementován pomocí zvolených technologií. Do systému bude umožněn tříúrovňový přístup s rolemi administrátor, manažer a zaměstnanec. Tento požadavek bude implementován pomocí knihovny Spring Security.

2.2 Metodika

Diplomová práce bude složena ze dvou částí. V první části budou nastudovány a shromážděny informace z vhodných informačních zdrojů. Nasbírané informace budou poté použity jako zdroj informací pro praktickou část diplomové práce.

V praktické části budou zpracovány požadavky na informační systém. Na základě těchto požadavků bude navrhnut a implementován informační systém. Pro implementaci informačního systému bude použit Java framework Spring Boot, ORM Hibernate a ORDBMS PostgreSQL.

3 Teoretická východiska

3.1 Objektová analýza

3.1.1 Objektově orientovaný přístup

Objektově orientovaném přistup se liší od klasického procedurálního programování tím, že pracuje pouze s objekty. Objekty v tomto přístupu se podobají reálným objektům tím, že mají své unikátní vlastnosti, operace, stavy a v podstatě vše, co muže mí t i skutečný objekt. Rozdělení systému do objektů usnadňuje chápání systému a tím i ulehčuje jeho vývoj. Objektovou orientaci je možné brát jako chápání struktury softwaru jakožto množinu objektů s přidělenými vlastnostmi a chováním (Pecinovský, 2013).

Mezi přednosti objektově orientovaného programování patří znovupoužitelnost a komponentový přístup a celkově snadnější tvorba softwaru (Merunka, Pergl a Pícka, 2005).

3.1.2 Zobecnění

Zobecnění je proces zjednodušování složitých problémů. Pokud řešíme složité problémy, není nutné se zabývat každým detailem řešení. Namísto toho řešení zjednodušíme tak, že si nastavíme pouze hlavní body procesu. Zobecnění nám dovoluje řešit problémy jednodušeji a také nám dovoluje docílit znovu-použitelnosti tříd tím, že snižuje jejich specializaci (Sintes, 1977).

3.1.3 Zapouzdření

Namísto pohledu na systém jako na jednu velkou monolitickou jednotku, zapouzdření umožnuje program rozdělit do menších nezávislých částí. Každá část vlastní sama sebe a provádí úkoly nezávisle na ostatních částech. Zapouzdření způsobuje nezávislost tím, že skrývá části interních detailů, nebo implementaci před okolními částmi (Sintes, 1997).

3.1.4 Dědičnost

Dědičnost v objektově orientovaném přístupu je možné přirovnat k dědičnosti ve skutečném světe. Stejně jako děti mohou potencionálně zdědit vlastnosti a schopnosti rodičů, stejně tak mohou objekty dědit od jiných objektů. Objekt, který získává atributy a vlastnosti nazýváme potomkem a třída, která atributy a vlastnosti předává se nazývá rodičovská třída. Dědičnost je v objektovém přístupu důležitá, protože umožnuje vytvářet komplexní a znovupoužitelné třídy. Dědičnost nám umožnuje vytvářet třídy na základě jiných tříd (Trapper, Tallbot a Haffner, 2004).

3.1.5 Polymorfismus

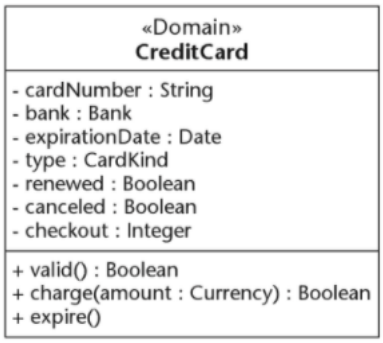
Polymorfismus doslova znamená „mnoho forem“. V objektově orientovaných kruzích polymorfismus poukazuje k mnoha implementacím jedné abstrakce. Abstrakce může být součástí třídy, nebo interface. Je tedy možné docílit polymorfismu tím, že několik podtříd zdědí vlastnosti a parametry od jedné nadtřídy a poté, každá z těchto podtříd vytvoří vlastní implementaci metody poskytnuté nadtřídou. Polymorfismu je také možné docílit tím, že několik tříd implementuje stejný interface, protože každá třída musí implementovat všechny metody deklarované v daném interface (Arrington, 2002).

Polymorfismus přináší dvě velké výhody. První výhodou je neomezená flexibilita v bežícím systému, protože implementace abstrakcí mohou být částečně kombinovány a spojovány a tím je možno docílit zajímavých efektů. Druhou výhodou je dobrá rozšiřovatelnost systému v dlouhodobém horizontu, takže systému může implementovat nové funkcionality bez ovlivnění stávajícího systému (Arrington, 2002).

3.2 Statické modelování

3.2.1 Třída

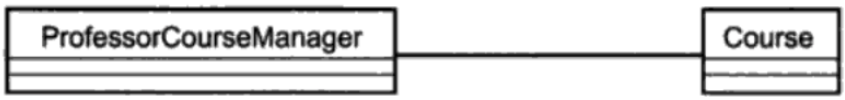
Třída je prototypem, návrhem nebo také modelem, který definuje různé vlastnosti. Těmito vlastnostmi mohou být data nebo operace. Data jsou reprezentována proměnnými instance nebo datovými proměnnými ve třídě. Operace jsou také známy jako chování, či metody, nebo funkce. Jelikož je třída datový typ, tak s ní nemůže být přímo manipulováno. Třída popisuje množinu objektů. Například jablko je ovoce, a proto je jablko příkladem ovoce. Pojem ovoce je typ jídla a jablko instancí ovoce stejně jako třída je datový typ a objekt je instancí třídy (Buyya, 2009).



Obrázek 1: Grafické znázornění třídy kreditní karta (Chonoles a Schardt, 2011, str. 281)

3.2.2 Asociace

Asociace je vazba, která popisuje skupinu spojení se stejným významem. Každá vazba v diagramu tříd odpovídá souboru spojení v diagramu výskytů. Výskytem vazby je spojení, které popisuje vztah mezi dvěma, nebo vícero objekty (Rumbaugh a Blaha, 2005).



Obrázek 2: Grafické znázornění asociace (Quatrani, 2000, str. 89)

3.2.3 Agregace

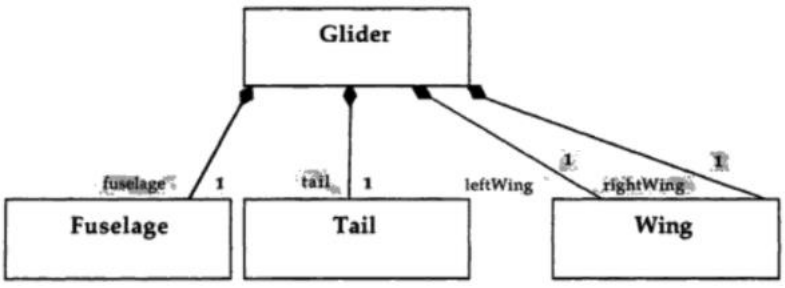
Agregace je zvláštní typ vazby typu “součást-celek”. Agregace popisuje vztah typu “skládá se z”, nebo také vztah typu “je součástí”. Tato vazba přenáší vlastnosti z celku na součásti. Zánikem celku nemusí zanikat i všechny části, jak je tomu u kompozice. Agregace má neomezený počet úrovní a je značena v diagramu jako čára vztahu s prázdným “diamantem” (Rumbaugh a Blaha, 2005).

Obrázek 3: Grafické znázornění agregace (Quatrani, 2000, str. 90)

3.2.4 Kompozice

Kompozice je vazba, vycházející z vazby tipu agregace. V kompozici každý objekt vazby nemůže existovat bez své nadřazené třídy (Chiarelli, 2016).

Kompozice reprezentuje silný typ vztahu mezi třídami. Kompozice je používána pro vyjádření vazby jako celek. Životní cykly tříd zapojených v kompoziční vazbě jsou zcela propojeny. Zanikne-li nadtřída této vazby, potom zaniknou i všechny podtřídy této vazby (Pilone a Pitman, 2009).



Obrázek 4: Kompozice objektu a její komponenty (Conaway, 2000, str. 124)

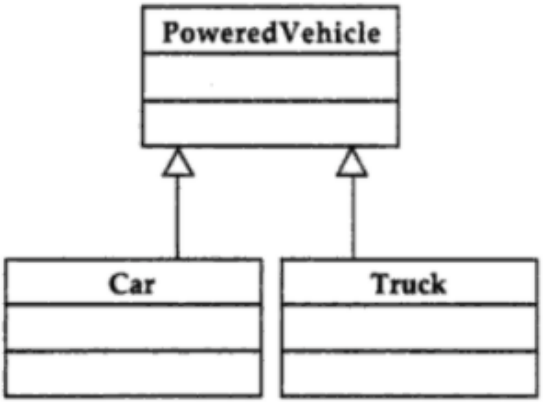
3.2.5 Dědičnost

Dědičnost je vazba, podle které jedna třída dědí vlastnosti a atributy jiné třídy (Matha, 2008).

Dědičnost neboli generalizace je často používána pro předání společných vlastností mezi různými třídami. Například pokud máme třídy Kočka a Pes je možné použít generalizaci pro obě třídy s třídou Zvíře. (Pilone a Pitman, 2005).

Dobrým příkladem dědičnosti mohou být objetky individuálního a firemního zákazníka. Zmínění zákazníci mají mezi sebou určité rozdíly, ale mají toho také mnoho společného. Podobnosti mezi nimi tedy mohou být umístěny do obecné třídy Zákazník, tzv. nadtypu (supertype), ze kterých bude dědit několik podtypů (subtypes) a to třídy IndividuálníZákazník a FiremníZákazník (Fowler, 2009).

Velmi důležitým principem správného použití dědičnosti je nahraditelnost, která tvrdí, že jakýkoliv objekt nadtypové třídy by měl být nahraditelný objektem podtypové třídy. V softwarovém světě to znamená, že objekt typu Zákazník může být nahrazen kdekoliv ve zdrojovém kódu objektem typu IndividuálníZákazník, nebo objektem typu FiremníZákazník (Fowler, 2009).

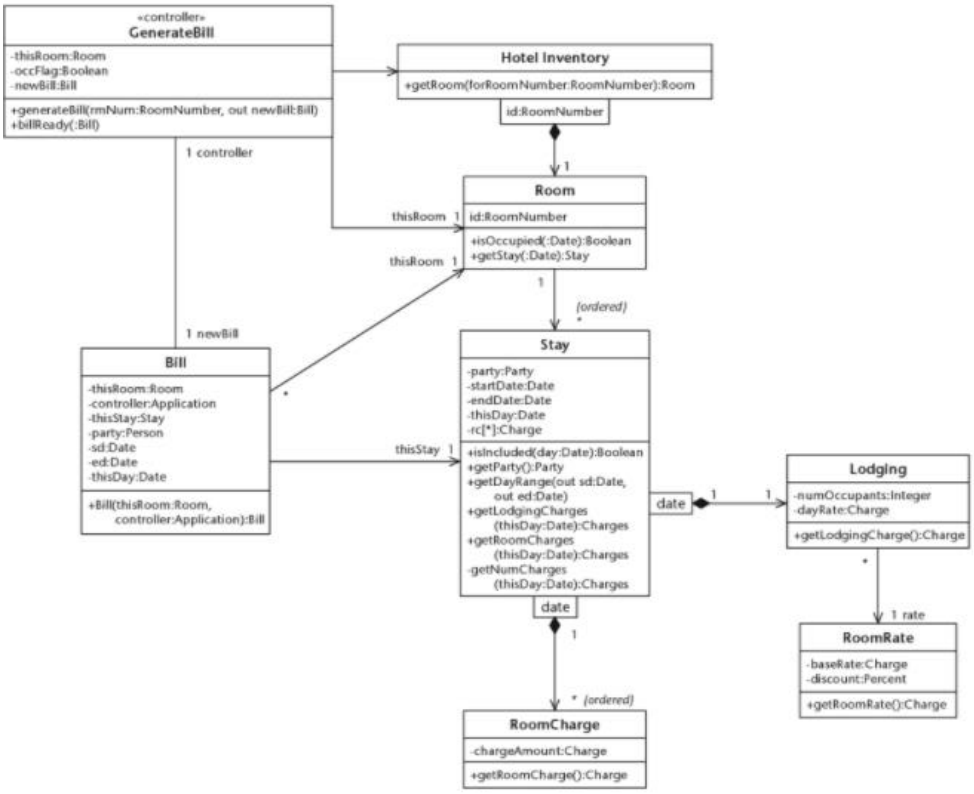


Obrázek 5: Znázornění dědičnosti (Conaway, Page-Jones a Constantine, 2000, str. 108)

3.2.6 Diagram tříd

Diagramy tříd jsou jedny ze základních typů UML diagramu. Diagramy tříd jsou určeny pro vyjádření statických vztahů v analyzovaném software. Jinak řečeno, jak jsou věci propojeny (Pilone a Pitman, 2005).

Třída reprezentuje skupiny objektů, která mají stejný stav a chování. Třídu si můžeme přestavit jako návrh objektů v objektově orientovaném systému. V jazyce UML to znamená, že třída je druh klasifikátoru. Například Volkswagen, Toyota a Ford jsou auta, proto je můžeme reprezentovat jako třídu Auto (Pilone a Pitman, 2005).



Obrázek 6: Příklad diagramu tříd (Chonoles a Schardt, 2011, str. 246)

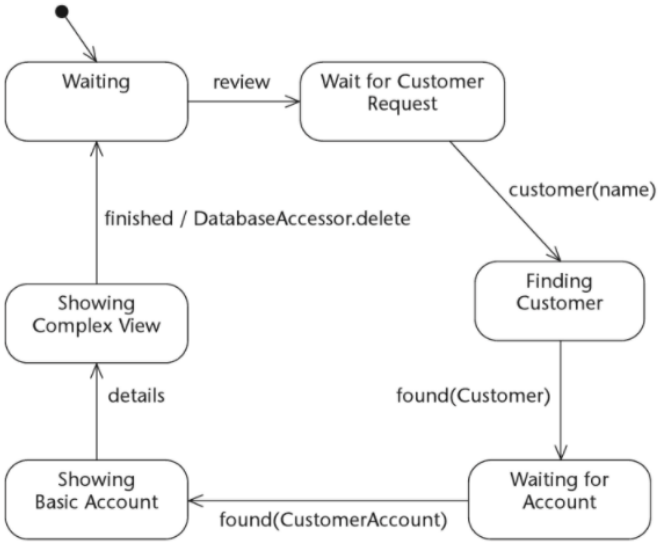
3.3 Dynamické modelování

Dynamický model zobrazuje různé stavy prvků a události, které mohou dané stavy změnit. Pokud by byl dynamický model zastaven v čase, stal by se z dynamického modelu model statický. Dynamická povaha modelu netkví pouze v čase, ale jak se v čase prvky mění. Například model, který popisuje stav dveří měnící se ze zavřených na otevřené a naopak, zachycuje tuto dynamickou povahu modelu. Pokud bychom tento model zastavovali v čase, vznikl by časově založený statický model (Unhelkar, 2005).

3.3.1 Stavový diagram

Stavový diagram popisu stavy objektu dané třídy, které mohou nastat a také změny těchto stavů. Tento diagram je vhodný pro modelování tříd, která obsahují proměnou, do které může být dosazena stav z množiny povolených stavů třídy. Tato proměnná se nazývá stavový atribut (Conaway, Page-Jones a Constantine, 2000).

Každý nevnořený stav ve stavovém diagramu odpovídá jedné možné hodnotě stavového atributu. Stav, ve kterém se třída nachází na počátku stavového diagramu se nazývá výchozí stav a je značen černou tečkou. Poslední stav stavového diagramu se nazývá koncový stav a je značen černou tečkou s kruhem (Conaway, Page-Jones a Constantine, 2000).

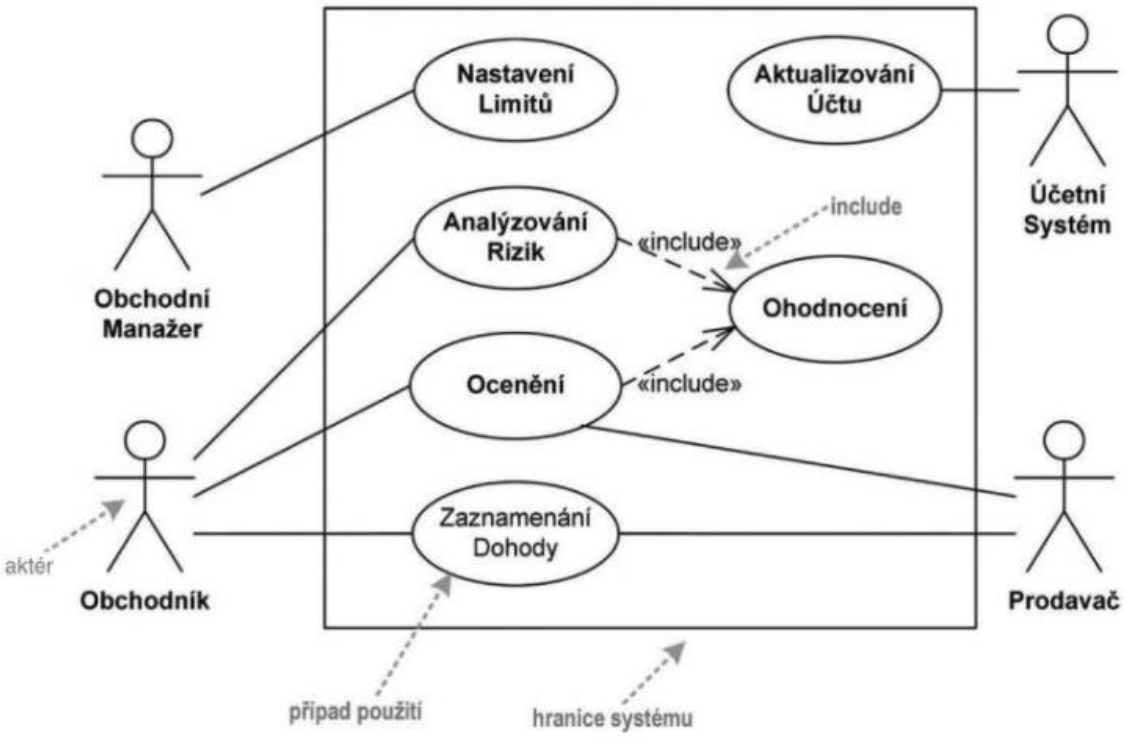


Obrázek 7: Stavový diagram ověření účtu (Chonoles, 2011, str. 276)

3.3.2 Případy užití

Diagramy případů užití slouží pro popis funkčních požadavků na systém. Diagramy případů užití, též use case diagramy, definují užití systému z pohledu uživatele a popisují pohled, v jakém bude systém používán (Fowler, 2009).

UML nemá jasné specifikace pro tvorbu případů užití, pouze definuje formát diagramu pro zobrazení. Samotný diagram případů užití není vždy nutný pro analýzu systému, ale může být užitečný. Nejlépe je možné si diagram užití představit tak, že se jedná o grafické znázornění sady případů užití. Diagram případů užití graficky znázorňuje aktéry, případy, užití a vztahy mezi nimi (Fowler, 2009).

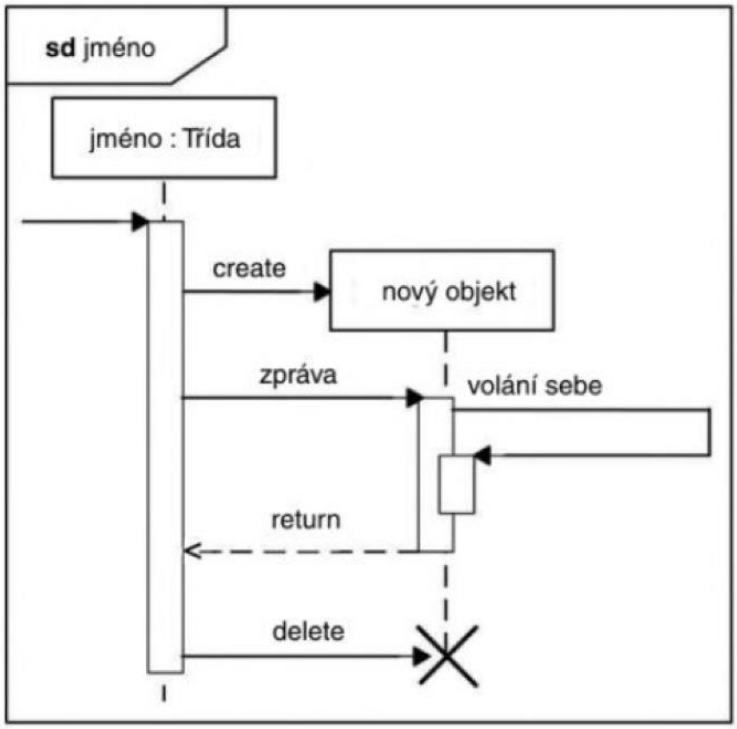


Obrázek 8: Příklad diagramu užití (Fowler, 2009, str. 106)

3.3.3 Sekvenční model

Sekvenční diagramy zobrazují čas, nebo sekvence akcí založených na událostech. Sekvenční digramy vychází z UML a jsou považovány za speciální typ určený pro modelování interakcí. Sekvenční diagramy zobrazují pořadí operací, které jsou použity pro komunikaci mezi zúčastněnými objekty. Sekvenční diagramy také mohou zvýrazňovat a upozorňovat na kritické návrhové chyby, stejné jako na místa, kde chybí řešení chybových stavů aplikace, nebo místa, která nejsou rovnoměrně bezpečnostně pokryta (Tarandach a Coles, 2020).

Sekvenční diagram většinou zobrazuje chování jednoho scénáře. Ukazuje několik vzorových objektů a zpráv, které jsou předávány mezi těmito objekty v rámci daného případu užití (Fowler, 2009).

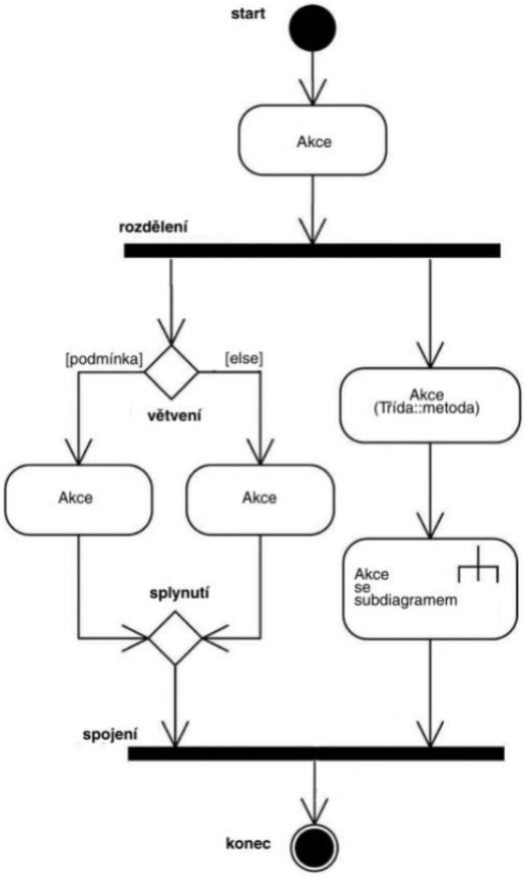


Obrázek 9: Příklad sekvenčního diagramu (Fowler, 2009, str. 167)

3.3.4 Model aktivit

Diagramy aktivit slouží k popisu procedurální logiky obchodních procesů a toků práce. Diagram aktivit a vývojový diagram jsou v mnoha užití velmi podobné a mohou sloužit ke stejným účelům, avšak na rozdíl od vývojových diagramů jsou diagramy aktivit schopné popsat i paralelní chování (Fowler, 2009).

Diagramy aktivit byly ve verzi UML 2 výrazně pozměněny a rozšířeny. V první verzi UML byly diagram aktivit považovány za speciální případy stavových diagramů, což bylo velmi problematické pro modelování toků práce. Tato vazba byla v druhé verzi UML odstraněna (Fowler, 2009).



Obrázek 10: Příklad diagramu aktivit (Fowler, 2009, str. 166)

3.4 Java

Java je silně typový programovací jazyk jehož syntax je velmi podobná programovacímu jazyku C. Java byla vyvinuta firmou Sun Microsystems ve roce 1991. Jako hlavní autory jazyka jsou uváděni James Gosling, Patrick Naughton, Mike Sheridan, Ed Frank a Chris Warth (Schildt, 2014).

Velkou výhodou programovacího jazyka Javy je nezávislost spouštěného programu na operačním systému. Tato výhoda je způsobena tím, že Java programy jsou spouštěné pomocí interpretu Java Virtual Machine (Loy, Niemeyer a Leuck, 2020).

3.4.1 Platformy jazyka

Java je distribuována v několika edicích. Každá edice Javy má své specifické použití podle účelu vytvářeného programu (Masterson, 2017).
Java Standart Edition
Java Enterprise Edition
Java Micro Edition

3.4.1.1 Java Standart Edition

Standardní edice jazyka Javy, které obsahuje základní knihovny pro tvorbu desktopových aplikací, pro přístup do sítě, či k přístupu do databází (Masterson, 2017).

3.4.1.2 Java Enterprise Edition

Edice jazyku Java pro tvorbu podnikových aplikací a informačních systémů. Tato edice je postavena na edici Standart rozšířenou o knihovny Enterprise edice. Hlavním důvodem pro vytvoření Enterpise edice Javy bylo zvýšení bezpečnosti a spolehlivosti jazyka. Enterprise edice je vytvořena tak, aby snížila složitost vývoje aplikaci (Masterson, 2017).

3.4.1.3 Java Micro Edition

Java Micro Edition slouží pro tvorbu aplikací, které budou spouštěny v prostředí s omezenými zdroji (např. mobilní telefony). Java Micro Edition vychází ze standardní edice a poskytuje API pro tvorbu aplikací s omezenými zdroji (Masterson, 2017).

3.4.2 Java Garbage Collector

Garbage collector je proces hledání v paměti programu, kde se identifikují objekty, které jsou programem používaný a ty, které programem používány nejsou. Používaný objekt nebo objekt s referencí jsou objekty na kterých běh programu závisí. Nepoužívané objekty a objekty bez reference jsou ty, na kterých běh programu nezávisí a mohou být z paměti programu odstraněny (Downey a Mayfield, 2020).

3.4.3 Java Virtual Machine

Zdrojový kód je nejprve kompilátorem přeložen do Java bajtkódu. V tomto kroku dojde také ke statické kontrole kódu, která pro další krok musí dopadnout úspěšně. Pokud kompilátor ve zdrojovém kódu najde chybu, tak přeruší svoji činnost. Pokud kontrola proběhne úspěšně, tak přeložený Java bajtkód je následně spuštěn pomocí interpretu Java Virtual Machine (Sierra a Bates, 2005).

Mezi výhody interpretu Java Virtual Machine patří zvýšení bezpečnosti programů, z důvodu sledování potenciálně nebezpečných operací a zabezpečení správy paměti (Roubalová, 2015).

3.4.4 Java Core API

Zkratka API značí Application Programming interface (aplikační programové rozhraní). Java Core API je nazýváno velké množství knihoven, které se musí vyskytovat v každém prostředí, kde se Java vyskytuje. Pokud námi vytvořený program využívá metody z Java Core API, nejsou tyto metody součástí námi vytvořeného programu, ale součástí Java Core API (Herout, 2000).

3.5 Spring framework

Při vývoji klasických průmyslových Java aplikací je zodpovědností vývojáře vytvořit dobře strukturované, rozšiřitelné a jednoduše testovatelné aplikace. Vývojáři proto využívají mnoho různých návrhových vzorů k docílení těchto požadavků, které jsou pro uživatele aplikace skryty. To vede ke snížení produktivity vývojářů a také ke snížení kvality samotné aplikace (Sarin a Sharma, 2017).

Spring framework je open source framework, který ulehčuje vývoj Java průmyslových aplikací. Poskytuje infrastrukturu pro vývoj dobře strukturovaných, rozšiřitelných a jednoduše testovatelných aplikací. Při použití Spring Frameworku se může vývojář soustředit pouze na vývoj aplikační logiky, což vede ke zvýšení produktivity. Spring framework je možné použít pro vývoj webových aplikací, appletů, či jiných typů Java aplikací (Sarin a Sharma, 2017).

První verze Spring Frameworku byly charakteristické především svojí jednoduchostí a svojí POJO orientací. To znamená, že jednotlivé komponenty Frameworku byly od sebe zcela odděleny a velmi málo komponent bylo definováno ve zdrojovém kódu. Ve verzi 2.5 byly poprvé použity anotace, které výrazně zredukovaly XML konfigurace Spring Frameworku díky použití skenování komponentů Frameworku. Skenování komponent ulehčilo konfiguraci aplikace, ale Spring Framework je oproti jiným webových Frameworkům stále dost náročný na konfiguraci (Rajput, 2018).

3.5.1 Spring Boot

Spring Boot poskytuje novou strategie pro vývoj aplikací se Spring Frameworkem s minimální námahou. Poskytuje vývojářům soustředit se pouze na vývoj aplikační logiky, protože veškeré konfigurace Spring Frameworku je již přednastavena. Spring Boot požaduje minimální, nebo nulovou potřebu konfigurace (Rajput, 2018).

Spring Boot změnil způsob vývoje Java aplikací pomocí Spring Frameworku. Spring Boot poskytuje všestranný způsob pro tvorbu Spring aplikací, protože poskytuje všestranný běhové prostředí pro Spring projekty, které mohou být velmi rozdílné. Spring

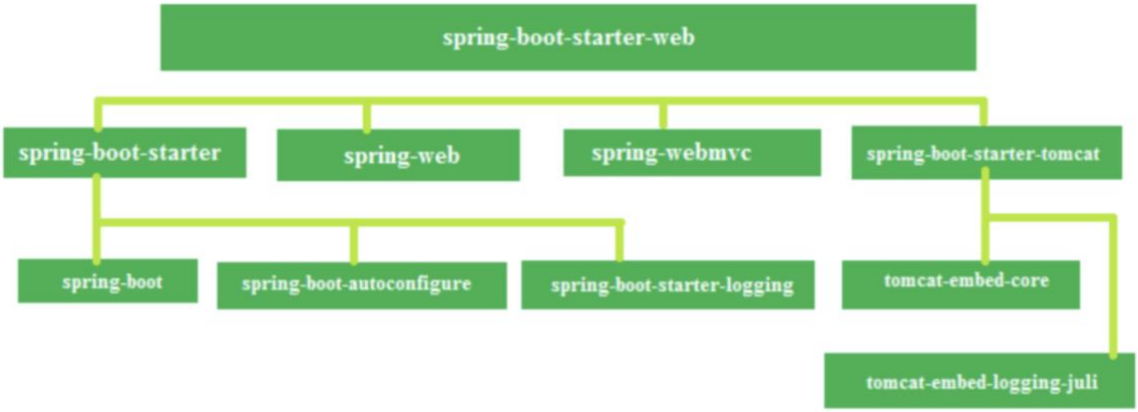
Boot poskytuje například prostředí pro webové aplikace ale i pro batch aplikace (Rajput, 2018).

3.5.2 Spring Boot starters

Starter je malý spring projekt určený pro každý modul jako jsou například MVC, JDBC, ORM apod. Do Spring aplikací je nutné pouze přidat starter pro potřebnou část projektu a Spring Boot poté přidá do aplikaci všechny potřebné knihovny pomocí Mavenu nebo Gradlu. Tato funkcionalita přináší časovou úsporu vývojářům, kterým odpadá povinnost dohledat a zakomponovat všechny potřebné knihovny pro daný modul (Rajput, 2018).

Seznam nejpoužívanějších starter balíčků (Rajput, 2018):

- spring-boot-starter-web-services: slouží pro vývoj aplikací používající SOAP protokol
- spring-boot-starter-web: slouží pro vývoj webových a REST aplikací
- spring-boot-starter-test: slouží pro jednotkové a integrační testy
- spring-boot-stater-jdbc: slouží pro manipulaci s daty v relačních databázích
- spring-boot-starter-heteoas: slouží pro usnadnění vývoje REST aplikací
- spring-boot-starter-security: slouží pro autentizace a autorizaci se Spring Security
- spring-boot-starter-data-jpa: slouží pro práci s ORM Hibernate Frameworkem
- spring-boot-starter-data-cache: poskytuje podporu pro Spring Framework chache
- spring-boot-starter-data-rest: poskytuje základní komponenty pro vývoj REST aplikací



Obrázek 11: Diagram balíčku Spring Boot Startet Web (Rajput, 2018, str. 13)

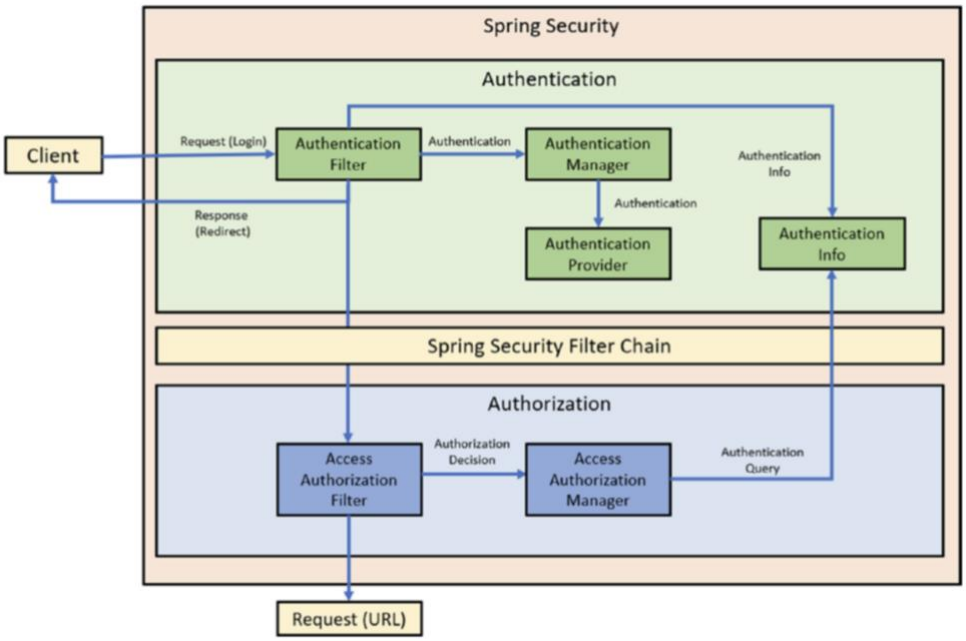
3.5.3 Spring Security

Spring Security je framework poskytující sadu bezpečnostních nástrojů pro Java aplikace vývojářské přívětivým a flexibilním způsobem. Spring Security dodržuje všechny doporučené bezpečnostní opatření použitelné se Spring Frameworkem a také se snaží pokrýt všechny vrstvy zabezpečení aplikací (Scarioni a Nardone, 2019).

Jedna z největších výhod Spring Security je možnost použití mnoha autentizačních modelů. Spring Security nabízí možnost integrovat autentizaci přes LDAP, Active Directory, OpenID, databázovou autentizaci, Jasypt kryptografii a mnohé další způsoby. Různé možnosti autentizace jsou velkou výhodou obzvláště v korporátním prostředí, nebo bankovních prostředí, kde je na bezpečnost prioritní (Scarioni a Nardone, 2019).

Mezi další výhody Spring Security patří možnost zabezpečit aplikaci v několika vrstvách. Můžeme například zabezpečit URL adresy, view templaty, metody anebo datové modely. Popřípadě můžeme tato opatření kombinovat podle potřeby (Scarioni a Nardone, 2019).

Jako další přednost můžeme označit to, že Spring Security je open source projekt, který má velkou komunitu vývojářů. Vývojáři mají tedy možnost nahlédnout do zdrojových kódu Spring Security a lépe porozumět tomu, jak tento framework funguje (Scarioni a Nardone, 2019).

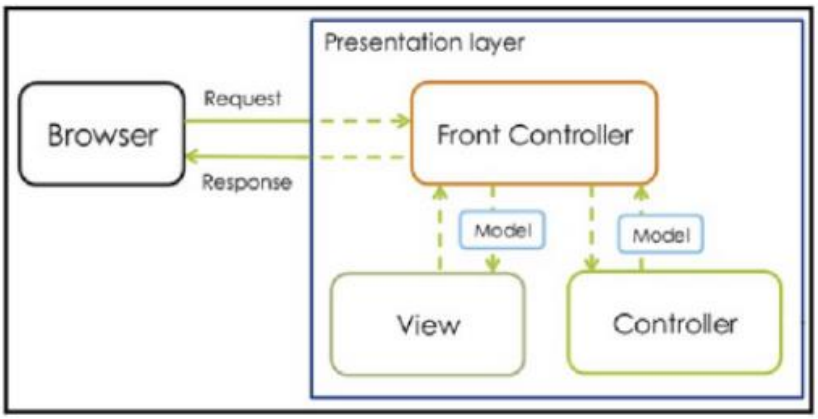


Obrázek 12: Flow diagram autentizace/autorizace (Scarioni a Nardone, 2019, str. 33)

3.5.4 Spring MVC

Hlavní princip MVC architektury spočívá v oddělení dat a prezentace. Například pokud chceme stejná data vykreslit v různých situacích jinak, postačí nám pouze použít jinou prezentační vrstvu. MVC Architektura rozděluje aplikaci na tři části, které spolu komunikují. Jak naznačuje zkratka MVC, aplikace je rozdělena na části Model, View a Controller (Niemeyer a Leuck, 2005).

Model obsahuje aplikační logiku a stará se o ukládání a načítání dat a ve většině případů komunikuje s databází. Vrstva View se stará o prezentaci dat uživateli. Ve většině případů se jedná o temple engine. Controllerová vrstva zpracovává požadavky uživatele a odesílá odpovědi zpět uživateli. Controller získává data od modelové vrstvy, tyto data následně vloží do View vrstvy, čímž vznikne požadovaná odpověď uživateli (Ganeshan, 2016).



Obrázek 13: Diagram Spring MVC (Ganeshan, 2016, str. 51)

3.5.5 Spring Data JPA

Spring Data JPA poskytuje jednoduchý způsob implementace datové vrstvy, která používá Java EE specifikaci JPA. JPA implementace většinou předchází velké množství zbytečného opakujícího se kódu, a to velmi ztěžuje implementaci dalších změn do datové vrstvy. Spring Data JPA se snaží vyřešit tyto problémy a poskytuje jednoduchý způsob implementace datové vrstvy bez nadbytečného a opakujícího kódu. JPA specifikace poskytuje vrstvu abstrakce pro komunikaci s různými databázovými systémy, které byly implementovány. Spring Data JPA vytvoří implementace repositářů a zapouzdří všechny implementační detaily JPA (de Oliveira a kol., 2018).

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

Zdrojový kód 1: Import hibernate knihovny (de Oliveira a kol., 2018, str. 79)

3.6 PostgreSQL

PostgreSQL je objektový a relační databázový management systém (ORDBMS), který je v různých formách vyvíjen od roku 1977. Na počátku se tento projekt nazýval Ingres na Kalifornské univerzitě v Berkeley. Ingres byl poté komerčně vyvíjen společností Relational Technologies/Ingres Corporation (Drake a Worsley, 2002).

V roce 1986 tým pod vedením Michela Stonebrakera z Berkeley navázal s vývojem na Ingres s cílem vytvoření objektového a relačního databázovému systému nazvaný Postgres (Drake a Worsley, 2002).

V roce 1996 kvůli díky novým open source požadavkům a zdokonaleným funkcionalitám softwaru byl Postgres přejmenován na PostgreSQL (Drake a Worsley, 2002).

PostgreSQL je široce považován za jeden z nejpokročilejších open source databázových systému na světě. Poskytuje spoustu vyspělých funkcionalit, které jsou často viděny pouze u průmyslově zaměřených komerčních produktů. PostgeSQL jsou poskytované jako open source volně dostupné verze, ale také jako komerční produkt (Drake a Worsley, 2002).

3.6.1 Výhody

PostgreSQL umožňuje psát procedury a funkce pomocí různých programovacích jazyků, protože architektura poskytuje danou flexibilitu pro použití více jazyků. Například můžeme používat pro psaní procedur jazyk SQL, ale také můžeme používat PL/pgSQL, PL/Perl, PL/Python, PL/Java a také PL/R. Podpora různých jazyků umožňuje řešit daný problém pomocí jazyku, který se pro daný problém hodí nejvíce. Například pokud potřebujeme vyřešit statistický problém, můžeme využít jazyk R, který pro řešení statistkách problému poskytuje mnoho knihoven (Obe a Hsu, 2012).

PostgreSQL funguje spolehlivě na všech podporovaných platformách. Pokud vyvíjíme aplikaci, která podporuje různé typy operačních systémů, je PostgreSQL velmi výhodný, jelikož podporuje Mac OS X, Windows i Linux (Obe a Hsu, 2012).

3.6.2 Nevýhody

PostgreSQL bylo od počátku navrhováno jako serverová databáze. Spousta lidí ale PostgreSQL používá na osobním počítači stejně jako SQL Server Express, nebo Oracle Express. Ale jelikož má PostgreSQL stejně jako zmiňované databáze velmi dobré zabezpečení, není snadné umožnit připojení externí aplikace na PostgreSQL databázi. A proto není vhodné použití této databáze jakožto embedované databáze, oproti databázím jako SQLite, nebo FireBird, které se pro toto použití hodí více (Obe a Hsu, 2012).

Většina sdílených hostingů bohužel nemá předinstalováno PostgreSQL, nebo mají pouze velmi zastaralé verze. Při použití sdílených hostingů je lepší použít MySQL. To se možná v budoucnu změní. Oproti tomu virtuální hostingy a cloudové servery nejsou o tolik dražší a je možno zde nainstalovat jakýkoliv software. Taková možnost je pro použití PostgreSQL výhodnější (Obe a Hsu, 2012).

4 Vlastní práce

Účelem kapitoly vlastní práce je využít informace z teoretické části k navržení a implementaci informačního systému pro monitoring garančních kontrol. Na začátku je nutné shromáždit požadavky na systém, na jejichž základě bude navržen a implementován daný systém.

4.1 Analýza požadavků

4.1.1 Popis společnosti

Firma Motohobby Kyjov je malá rodinná společnost, která se věnuje prodeji a opravě zahradní techniky, skútrů a motocyklů. Firma má celkem pět zaměstnanců, z toho se dva zaměstnanci věnují primárně servisu a zbylí zaměstnanci se věnují prodeji. Firma poskytuje servis především produktům zakoupených v jejich prodejně. S tím souvisí také garanční prohlídky, které jsou nedílnou součásti podmínek záruky produktů. Analyzovaný systém má za úkol zaměstnancům firmy ulehčit evidenci vykonaných garančních prohlídek a případně upozornit na stroje s opožděnou garanční prohlídkou.

4.1.2 Systémové požadavky

Na základě konzultací s majitelem firmy byly na systém kladeny následující požadavky:

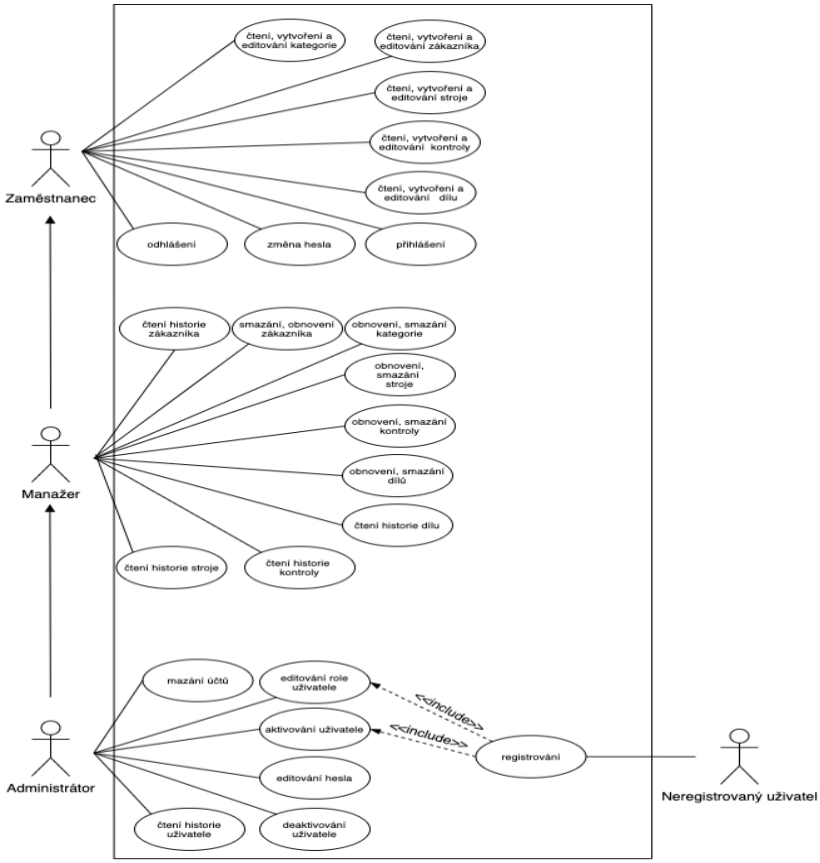
1) Zobrazení záznamů o strojích, zákaznících a vykonaných garančních prohlídkách pod zaměstnaneckým účtem
2) Zobrazení strojů se zpožděnou garančních prohlídkou pod zaměstnaneckým účtem
3) Tříúrovňový přístup pro zaměstnance, manažery a administrátora
4) Možnost spravovat uživatele, uživatelská práva a hesla uživatelů pod administrátorským účtem
5) Editovat a mazat záznamy o strojích, zákaznících a vykonaných garančních prohlídkách pod manažerským účtem

4.1.3 Analýza systému pomocí UML

Před samotnou implementací je nutné analyzovat systém také z technického hlediska. K tomu nám poslouží analýza pomocí UML, díky které můžeme požadavky zapracovat do návrhu systému a připravit si návrh systému pro samotnou implementaci. Pro analýzu pomocí UML je vypracováno několik diagramů. Pro pochopení fungování systému z hlediska uživatele poslouží diagram použití. Dále pro analýzu systému jsou zpracovány diagramy tříd a pro analýzu datové vrstvy je zpracován ER diagram.

Diagram případů užití

Diagram případů užití slouží pro popis vlastností systému z hlediska uživatele. Na diagramu jsou znázorněni aktéři systému a jejich případy užití. Systém umožňuje tříúrovňový přístup, což znázorňuje i tento diagram případů užití.



Obrázek 14: Diagram případů užití (zpracování vlastní)

Scénáře případů užití

Na základě diagramu případů užití byly pro hlubší analýzu vytvořeny scénáře těchto případů. Tyto scénáře popisují jednotlivé kroky, které jsou mezi aktéry třeba provést pro provedení daného případu užití. Pro každý scénář je uveden základní tok a alternativní tok případu užití a také aktéři, kteří jsou pro daný případ užití zapojení. Pro vybrané scénáře jsou vyhotoveny diagramy aktivit.

UC1 Přihlášení do systému

Popis

- Přihlášení zaměstnance do systému

Aktéři

- Zaměstnanec, Systém

Podmínky spuštění

- Zaměstnanci byly administrátorem vytvořeny přihlašovací údaje do systému.

Základní tok

1) Zaměstnanec se připojí do systému
2) Systém odešle přihlašovací formulář
3) Zaměstnanec vyplní přihlašovací údaje a formulář odešle
4) Systém ověří zaslané přihlašovací údaje
5) Systém přihlásí zaměstnance do aplikace

Alternativní tok 1

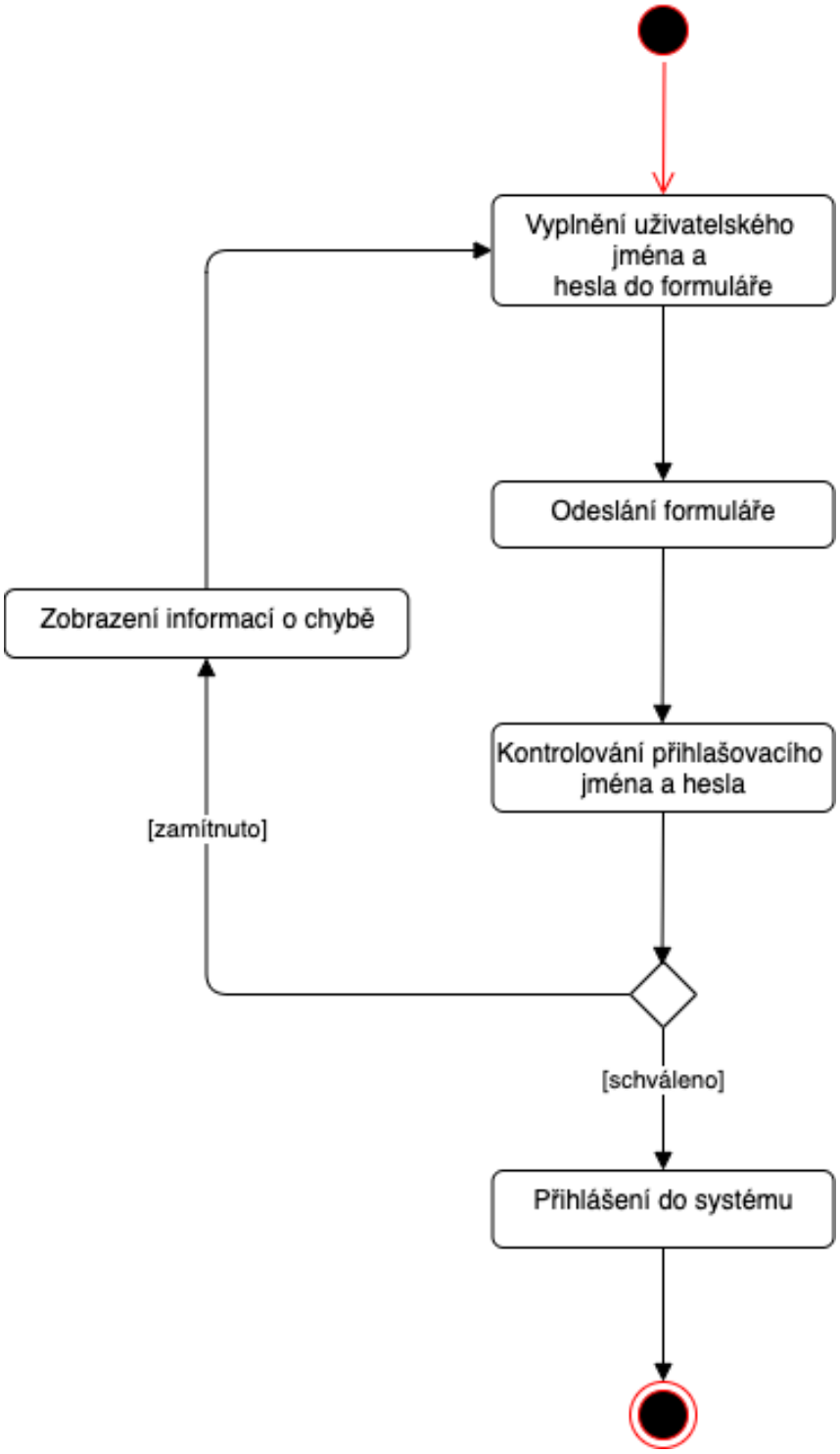
3.1) Zaměstnanec vyplní nesprávné přihlašovací údaje

3.2) Systém odešle znovu přihlašovací formulář s chybovou hláškou o nesprávnosti přihlašovacích údajů

3.3) Zaměstnanec vloží správné přihlašovací údaje a odešle formulář znovu, tok pokračuje na základním toku v bodě 4

Podmínky ukončení

- Zaměstnanec je přihlášen do aplikace



Obrázek 15: Diagram aktivit přihlášení (vlastní tvorba)

UC2 Vytvoření zákazníka

Popis

- Vytvoření záznamu zákazníka

Aktéři

- Zaměstnanec
- Systém

Podmínky spuštění

- Zaměstnanec je přihlášen do systému

Základní tok

1) Zaměstnanec vstoupí na stránku s formulářem pro vytvoření zákazníka
2) Systém odešle formulář pro vytvoření zákazníka
3) Zaměstnanec vyplní údaje zákazníka a odešle formulář
4) Systém provede validaci údajů zákazníka
5) Systém uloží záznam o zákazníkovi do databáze

Alternativní tok 1

3.1) Zaměstnanec zadá nesprávný údaj o zákazníkovi

3.2) Systém odešle formulář znovu s chybovými hláškami o údajích

3.3) Zaměstnanec opraví údaje ve formuláři a odešle formulář znovu, tok pokračuje na základním toku v bodě 4

Podmínky dokončení

- Systém úspěšně uloží záznam o zákazníkovi do databáze. Uživatel je přesměrován na detail zákazníka.

UC3 Přidání záznamu stroje

Popis

- Přidání záznamu stroje

Aktéři

- Systém
- Zaměstnanec

Podmínky spuštění

- Zaměstnanec je přihlášen do systému. Pro daný stroj je již v databázi uložen záznam k majiteli stroje, jakožto zákazníka.

Základní tok

1) Zaměstnanec vstoupí na stránku se seznamem všech uložených zákazníků
2) Zaměstnanec vyhledá záznam zákazníka a u tohoto záznamu klikne na odkaz ke stránce s formulářem pro vytvoření stroje
3) Systém odešle formulář pro vytvoření záznamu stroje
4) Zákazník vyplní údaje o stroji a odešle formulář
5) Systém provede validaci údajů o stroji
6) Systém přiřadí k záznamu o stroji cizí klíč k ID zákazníka a údaj uloží
7) Systému uloží do historie stroje záznam o vytvoření

Alternativní tok 1

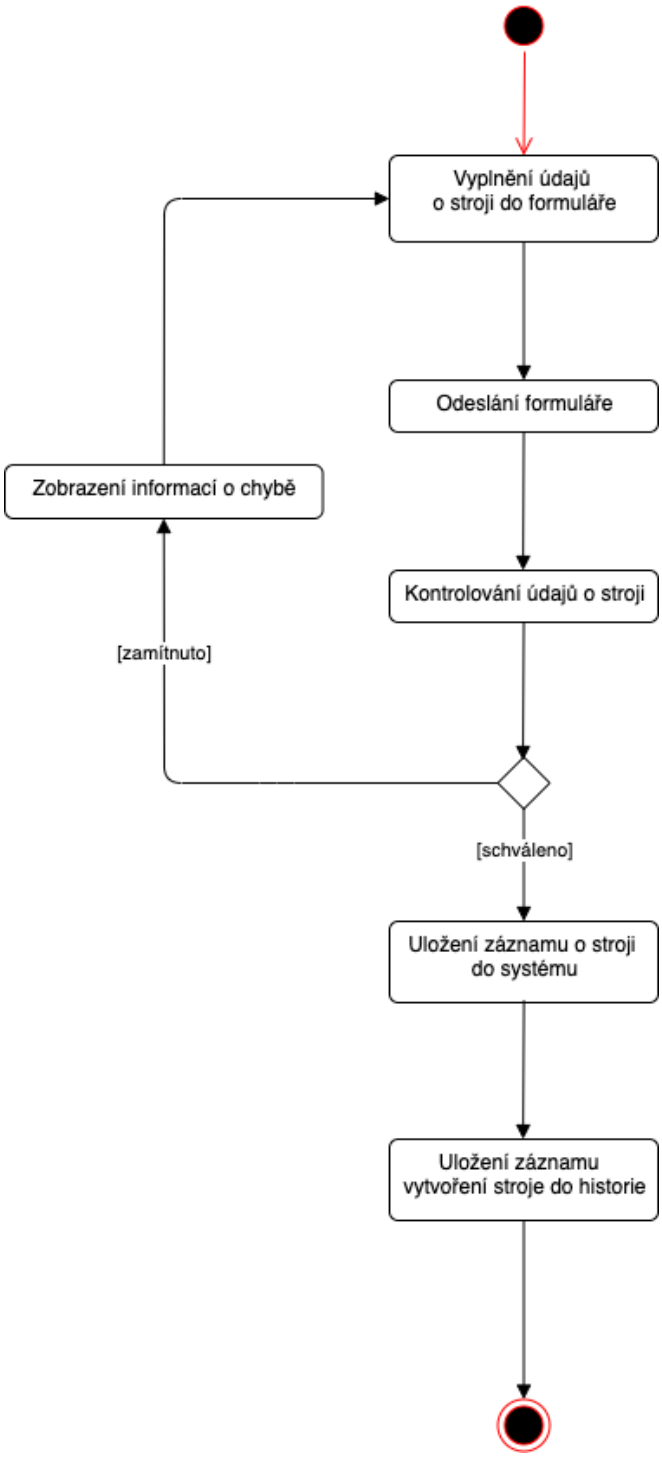
5.1) Zaměstnanec vloží nesprávné údaje o stroji

5.2) Systém odešle formulář znovu s chybovými hláškami o údajích o stroji

5.3) Zaměstnanec opraví údaje a odešle formulář znovu, tok pokračuje v základním toku na kroku 5

Podmínky ukončení

- Systém úspěšně uloží záznam o stroji s referencí na daného zákazníka. Uživatel je přesměrován na detail vytvořeného záznamu o stroji.



Obrázek 16: Diagram aktivit vytvoření stroje (vlastní zpracování)

UC4 Vytvoření garanční prohlídky

Popis

- Vytvoření garanční prohlídky zaměstnancem

Aktéři

- Zaměstnanec
- Systém

Podmínky spuštění

- Zaměstnanec je přihlášen do systému

Základní tok

1) Zaměstnanec vstoupí na stránku se seznamem strojů s opožděnou garanční prohlídkou
2) Systém v databázi vyhledá stroje s opožděnou garanční prohlídkou a odešle data zaměstnanci
3) Zaměstnanec vyhledá v seznamu stroj a u tohoto stroje klikne na odkaz stránky s formulářem k zaznamenání garanční prohlídky
4) Systém vytvoří formulář a odešle jej zaměstnanci
5) Zaměstnanec vyplní formulář a odešle tento formulář do systému
6) Systém provede validaci údajů
7) Systém uloží údaj do databáze s cizím klíčem k ID stroje

Alternativní tok 1

4.1) Zaměstnanec vloží nesprávné údaje o garanční prohlídce a odešle formulář

4.2) Systém provede validaci a odešle formulář zpět s chybovými hláškami

4.3) Zaměstnanec údaje opraví a odešle formulář do systému, tok pokračuje v základním toku v bodě 6

Podmínky dokončení

- Systém uloží údaj o garanční prohlídce a zaměstnanec je přesměrován na detail záznamu stroje.

UC5 Vytvoření účtu zaměstnance

Aktéři

- Systém
- Administrátor
- Neregistrovaný uživatel

Podmínky spuštění

- Administrátor je přihlášen do systému

Základní tok

1) Neregistrovaný uživatel vstoupí na stránku s formulářem registrace
2) Neregistrovaný uživatel vyplní formulář a odešle formulář do systému
3) Systém provede validaci údajů formuláře
4) Systém vytvoří neaktivního uživatele
5) Systém uloží záznam o vytvoření do historie uživatele
6) Administrátor vstoupí na stránku se seznamem blokovaných účtů
7) Administrátor provede validaci údajů uživatele
8) Administrátor aktivuje účet uživatele
9) Systém uloží záznam o aktivaci do historie uživatele

Alternativní tok 1

3.1) Zaměstnanec vloží nesprávné údaje o garanční prohlídce a odešle formulář

3.2) Systém provede validaci a odešle formulář zpět s chybovými hláškami

3.3) Zaměstnanec údaje opraví a odešle formulář do systému, tok pokračuje v základním toku v bodě 5

Alternativní tok 2

6.1) Administrátor upraví roli uživatele

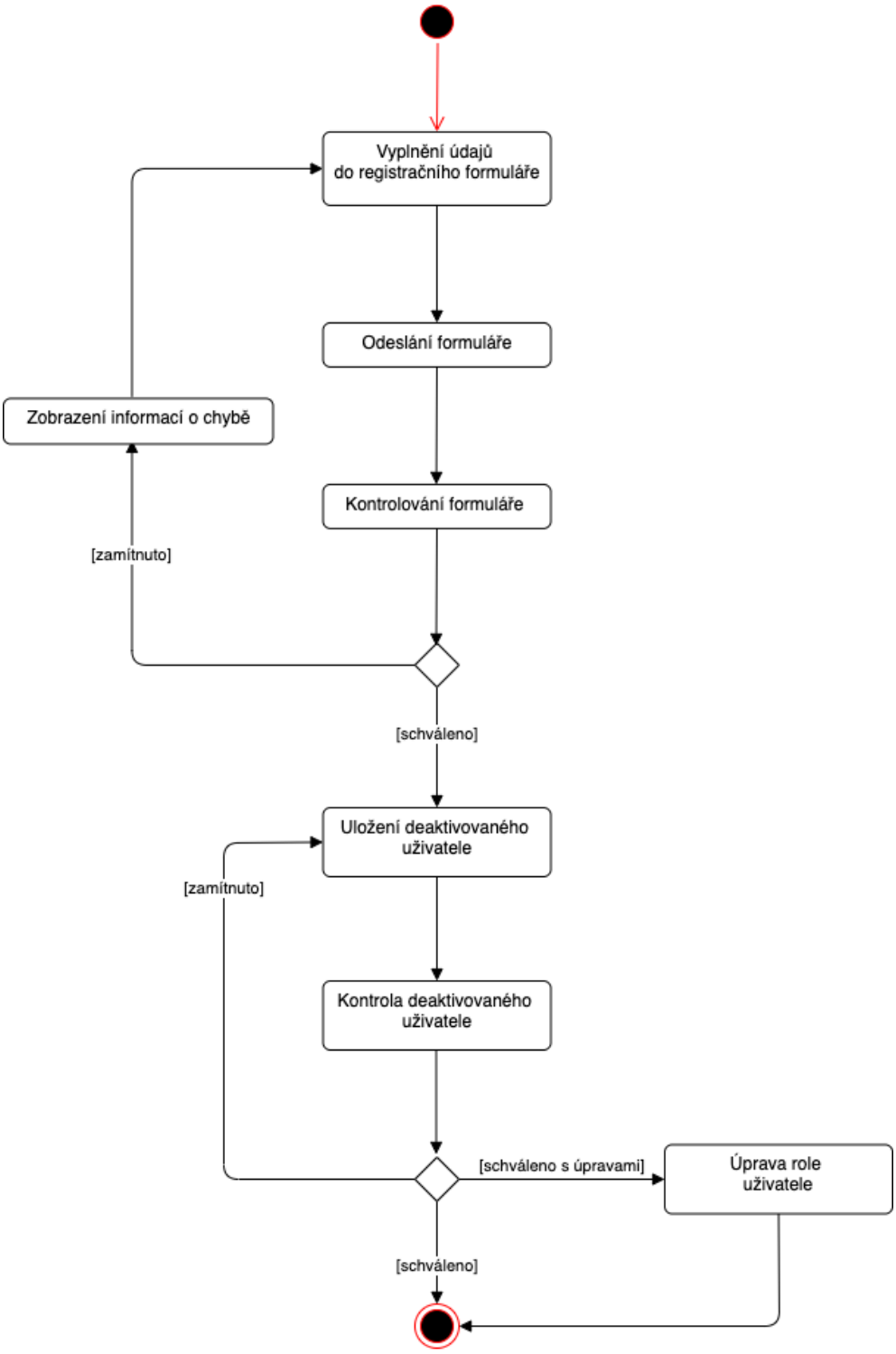
6.2) Systém uloží záznam o změně role do historie uživatele, tok pokračuje v základním toku v bodě 8

Alternativní tok 3

6.2) Administrátor ponechá uživatele neaktivního

Podmínky dokončení

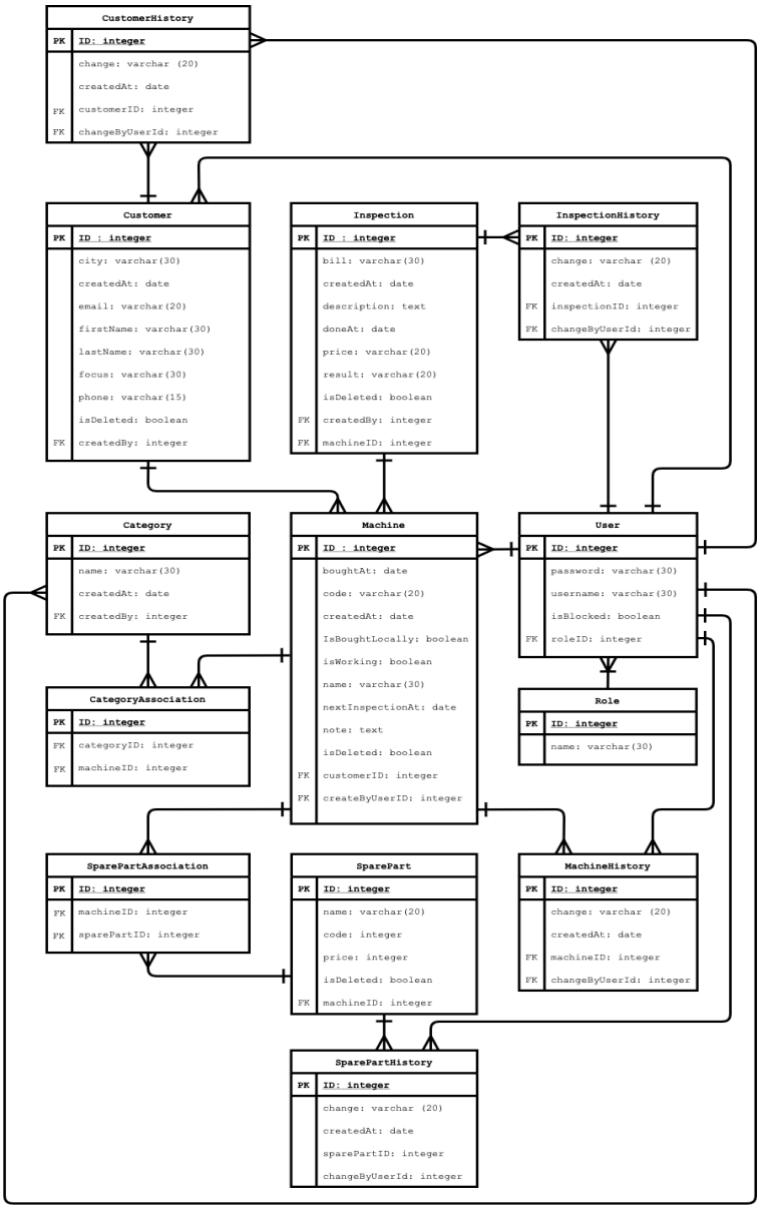
- Systém úspěšně uloží nový uživatelský účet



Obrázek 17: Diagram aktivit vytvoření uživatele (vlastní zpracování)

ER Diagram

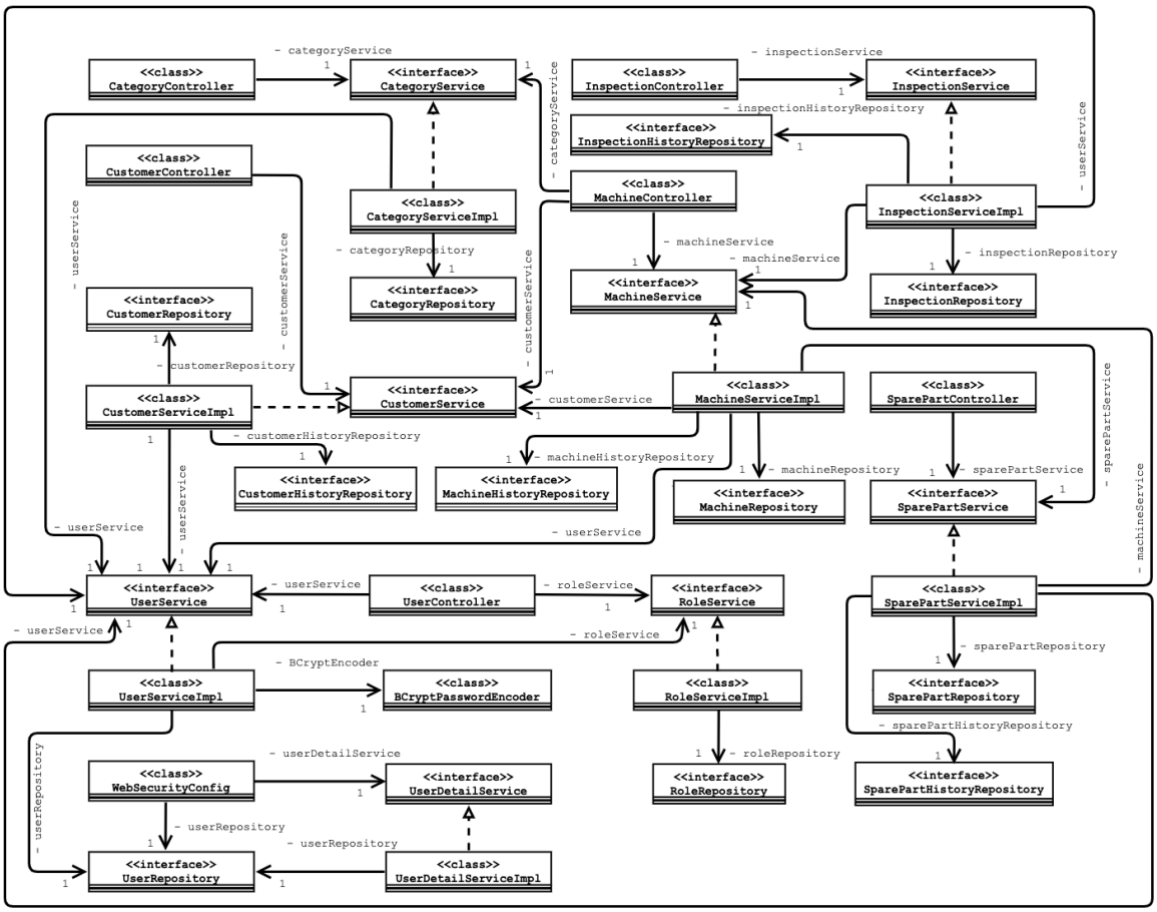
ER diagram nepatří mezi UML diagramy, ale pro potřeby analýzy systému je vyhovující. ER diagram popisuje databázovou strukturu. Na diagramu jsou vyznačeny jednotlivé databázové tabulky a vztahy mezi nimi. Tabulky jsou propojeny pomocí cizích klíčů, které jsou označeny jako FK. Primární klíče jsou unikátní identifikátory v rámci jednotlivých tabulek a jsou označeny PK.



Obrázek 18: ER Diagram (vlastní tvorba)

Základní digram tříd

Struktura diagramu tříd zobrazuje jednotlivé třídy a vazby mezi těmito třídami systému. Všechny zobrazené třídy jsou dále detailně popsány v následujících diagramech tříd. U jednotlivých tříd nejsou zobrazeny metody ani parametry. Metody a parametry tříd jsou zobrazeny v detailních digramech tříd.

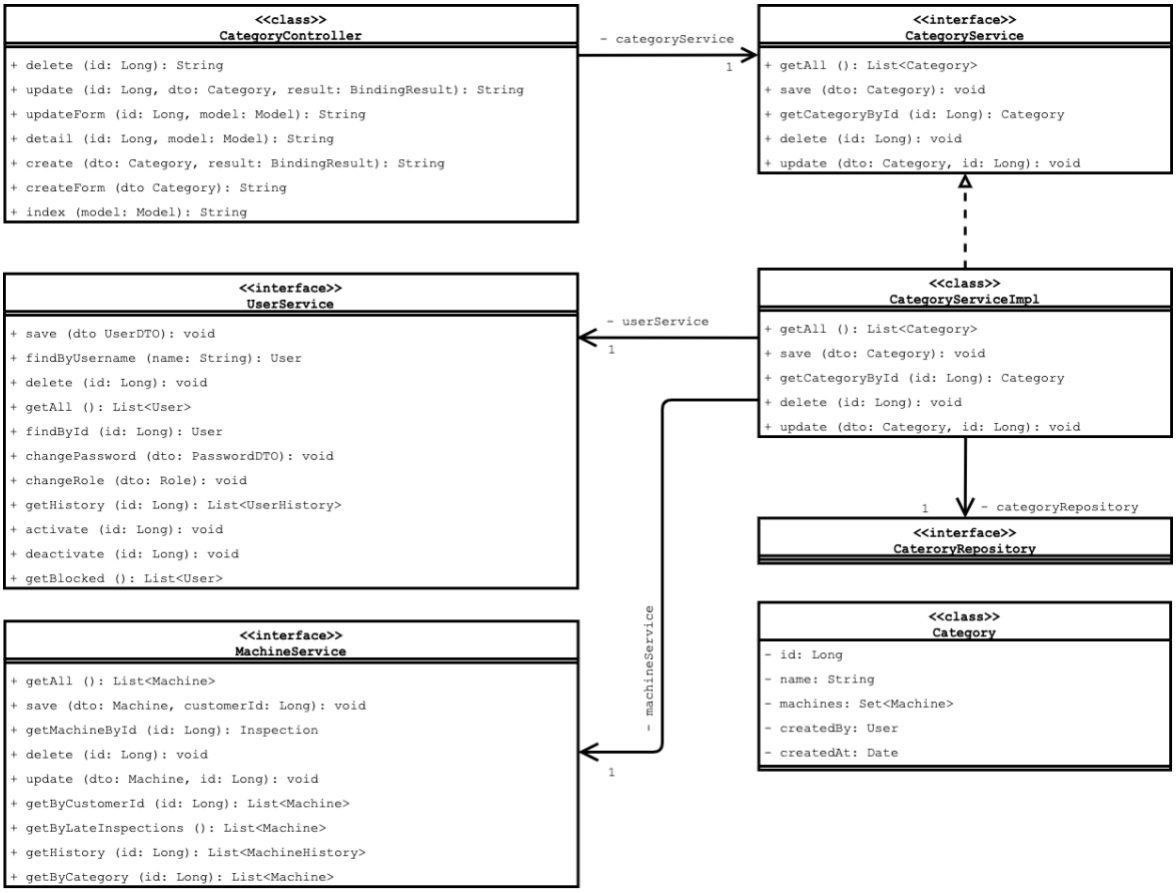


Obrázek 19: Základní přehled diagramu tříd (zpracování vlastní)

Detail diagramu tříd pro část kategorie

Detail diagramu tříd analyzuje část systému, která slouží pro zpracování záznamů o kategoriích strojů. Na základě těchto zadaných kategorií je možné poté stroje filtrovat. Pro každou kategorii je uloženo jméno, zařazené stroje a uživatele, který tuto kategorii vytvořil.

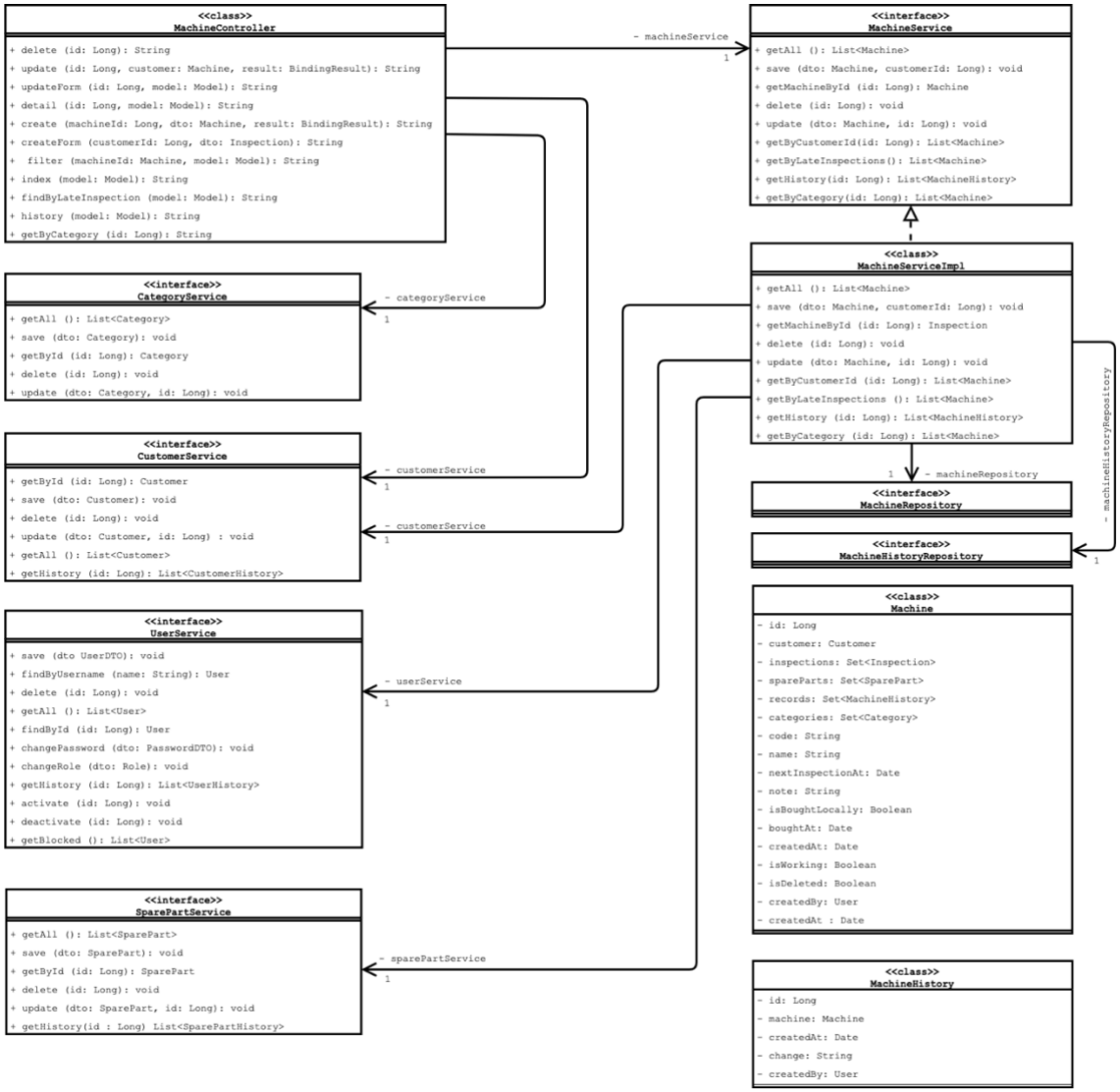
Třída CategoryController slouží pro příjem požadavků od klienta. Tyto požadavky následně zpracuje a odešle odpověď ke klientovi v podobě html souboru. Kontrolerová vrstva využívá metod servisní vrstvy třídy CategoryService. V této třídě je implementována obchodní logika. Servisní vrstva v tomto případě využívá metod třídy CategoryRepository, která slouží pro manipulaci dat v databázi.



Obrázek 20: Detail diagramu tříd pro kategorie (zpracování vlastní)

Detail diagramu tříd pro část stroje

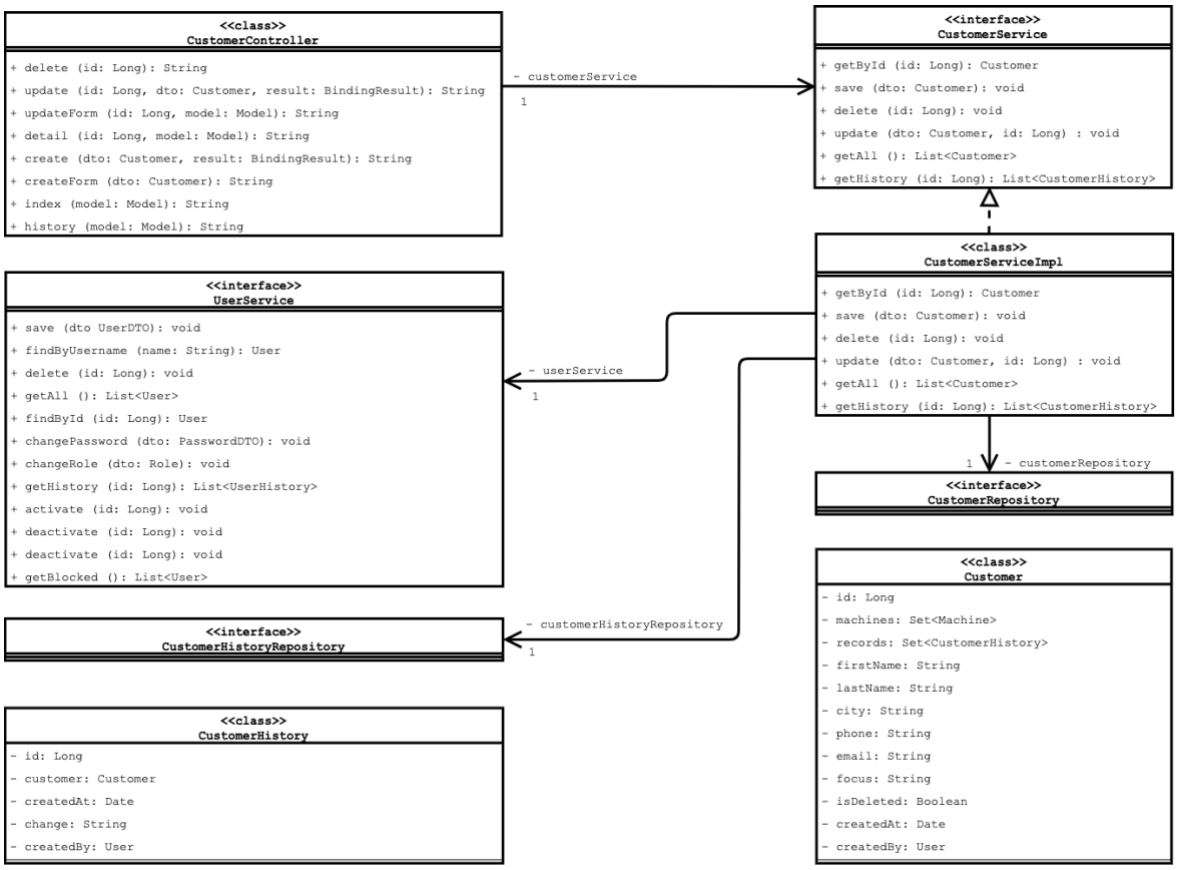
Detail digramu tříd je analýzou systému pro část systému sloužící pro zpracování záznamů o strojích. K záznamům o strojích je možné přiřadit záznamy o provedených kontrolách. Všechny změny provedené na strojích jsou zaznamenávány do historie strojů, kde je možné najít popis této změny, datum provedení a uživatele, který tuto změnu provedl. Ke každému stroji je možné také přiřadit náhradní díly. To zajistí přehled nad náhradními díly, které byly k danému stroji zakoupeny.



Obrázek 21: Detail diagramu tříd pro stroje (zpracování vlastní)

Detail diagramu tříd pro část zákazníci

Detail digramu tříd je analýzou systému pro část systému sloužící pro zpracování záznamů o zákaznících. K záznamům o zákaznících je možné přiřadit záznamy o strojích, které daný zákazník provozuje. Všechny provedené změny na záznamech o zákaznících jsou zaznamenávány do databáze. Pro každou změnu je zaznamenán popis změny, datum a uživatel, který tuto změnu provedl.

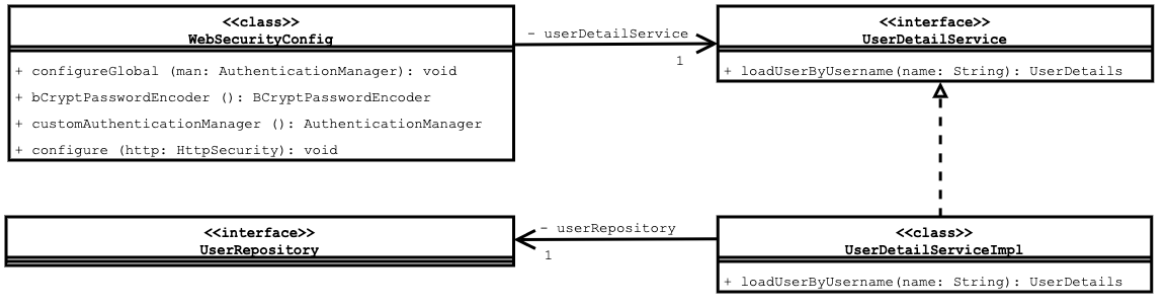


Obrázek 22: Detail diagramu tříd pro zákazníky (zpracování vlastní)

Detail diagramu tříd pro část nastavení zabezpečení

Detail diagramu tříd pro část nastavení zabezpečení slouží pro analýzu systému, která slouží pro zabezpečení přístupu k uloženým datům. Pro přístup k uloženým údajům je nutné, aby uživatel byl autorizován a měl dostatečná práva pro dané záznamy.

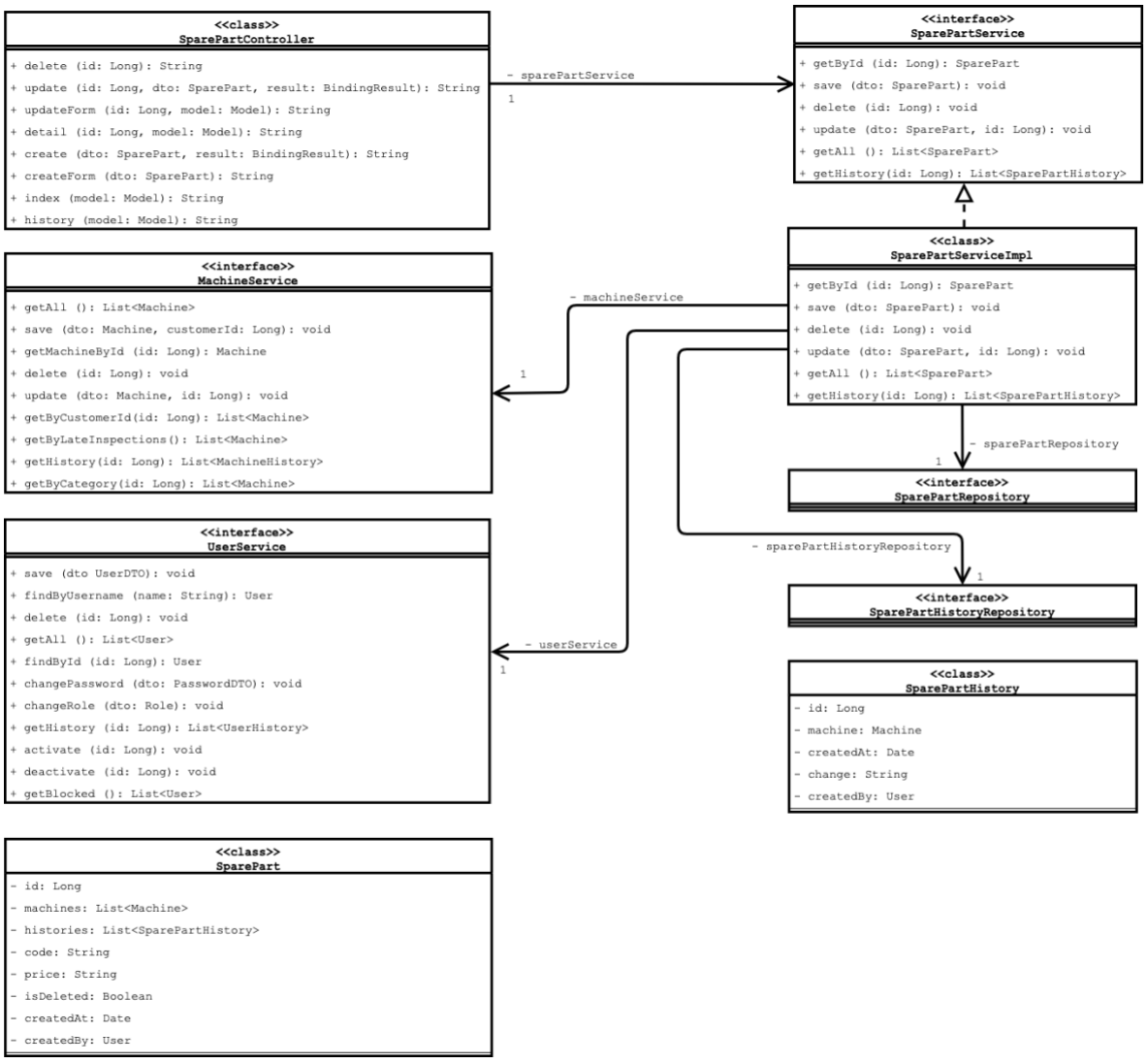
Zabezpečení systému využívá knihovny SpringBoot Security, která dále řeší autentizaci a autorizaci uživatelů.



Obrázek 23: Detail diagramu tříd pro zabezpečení (zpracování vlastní)

Detail diagramu tříd pro náhradní díly

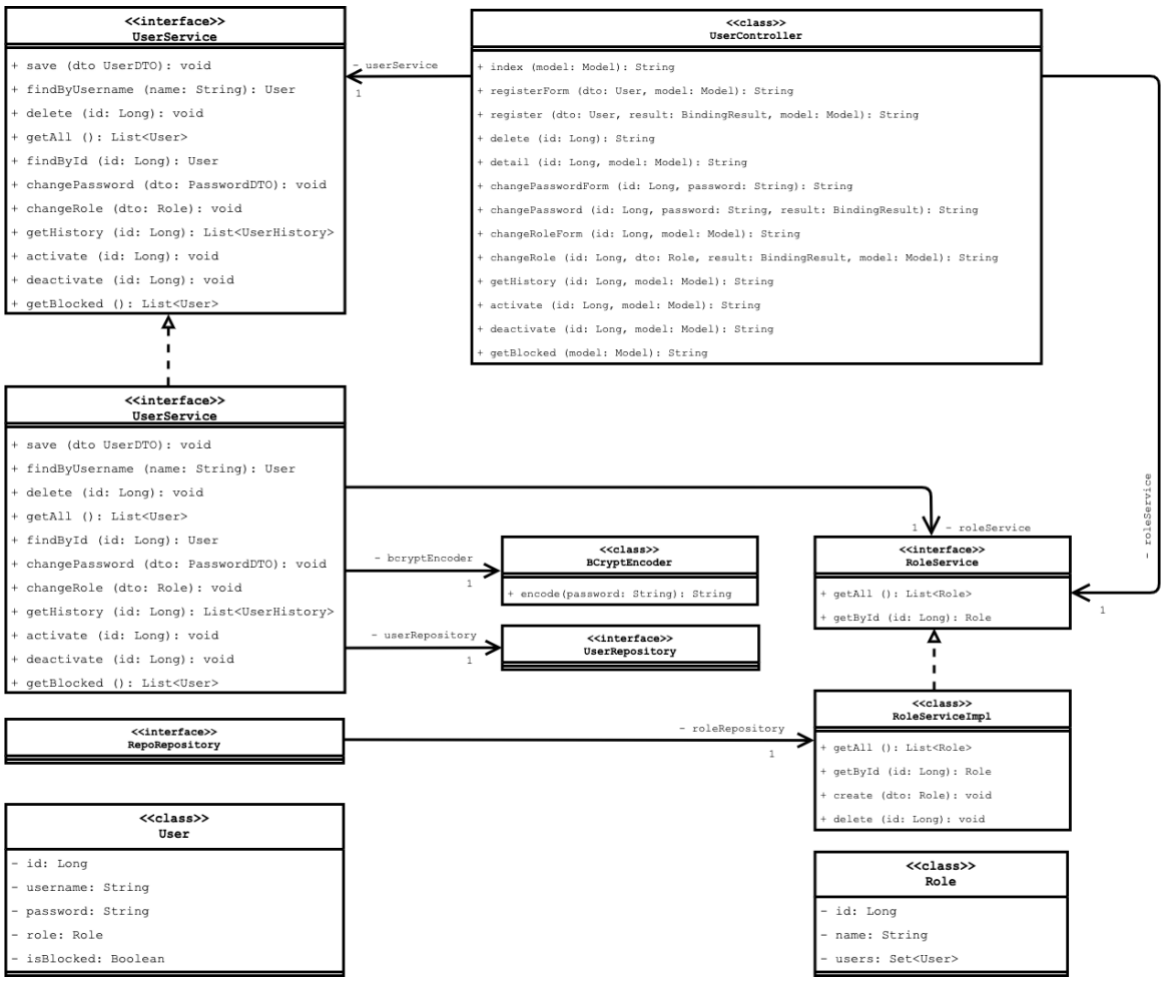
Ke každému stroji je možné přidat náhradní díly, které byly k danému stroji přikoupeny. Pro každý náhradní díl je uložen do databáze jeho název, firemní kód, cena, datum vytvoření a uživatel, který dané náhradní díl vytvořil.



Obrázek 24: Detail diagramu tříd pro náhradní díly (zpracování vlastní)

Detail diagramu tříd pro část uživatel

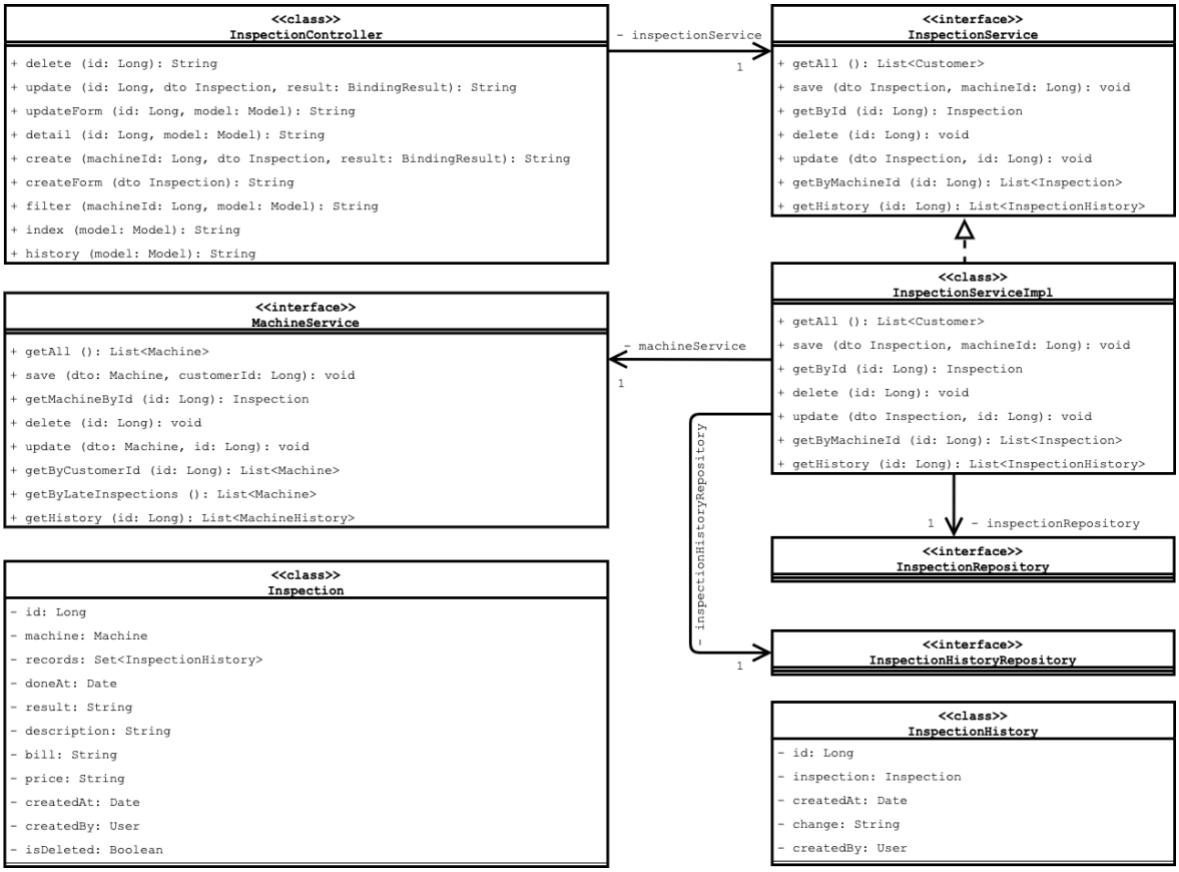
Detail digramu tříd je analýzou systému pro část systému sloužící pro zpracování záznamů o uživatelích. K záznamům o uživatelích je možné přiřadit zvolenou roli. Podle zadané role bude uživatel dále v systému autorizován. Pro každého uživatele je možné ukládat do databáze uživatelské jméno, heslo a roli. Každého uživatele je také možné blokovat, čímž je danému uživateli odepřen přístup do systému.



Obrázek 25: Detail diagramu tříd pro uživatele (zpracování vlastní)

Detail diagramu tříd pro část garanční prohlídky

Detail digramu tříd je analýzou systému pro část systému sloužící pro zpracování záznamů o provedených garančních prohlídkách. Garanční prohlídky jsou přiřazovány k daným strojům. Změny provedené v garančních prohlídkách jsou ukládány do databáze. K provedeným změnám je uložen uživatel, který změnu provedl, datum a popis změny.



Obrázek 26: Detail diagramu tříd pro garanční kontroly (zpracování vlastní)

4.2 Implementace systému

Systém je implementován podle MVC architektury. Pro každý druh záznamu je vytvořen samostatný kontroler, který přijímá požadavky odeslané uživatelem pomocí uživatelského rozhraní. Kontroler odeslaný požadavek zpracuje a podle implementované logiky na tento požadavek zareaguje. Kontroler k tomu využívá servis a templatu uživatelského rozhraní. Kontroler získá potřebná data od servis a vloží je do templatu, který následně odešle uživateli. V dané servise je skryta obchodní logika potřebných procesu. Je zde také implementována logika pro ukládaní záznamů do historie aktivit.

Pro implementaci datové vrstvy je využito objektového relačního mapovače Java Persistence API. K tomu bylo nutné definovat jednotlivé databázové entity a vazby mezi nimi.

4.2.1 Implementace datové vrstvy

Datová vrstva byla implementována pomocí objektového relačního mapovače (ORM) Hibernate. Hibernate nabízí pro práci se Spring Boot aplikacemi knihovnu JPA (Java Persistence API). Spring Boot tuto knihovnu zasadil do předpřipraveného „Spring Boot Starter“ balíčku. Vývojáři už tedy stačí pouze vložit do projektu předpřipravený balíček a může začít knihovnu JPA používat.

Jak název Java Persistence API napovídá, tato knihovna se stará o správnou manipulaci dat. Tato knihovna programátora odstíní od spousty problému, které by jinak musel řešit s vlastní implementací. Zde je uvedena implementace JPA pro tabulku záznamů strojů. Každá entita musí mít anotaci @Entity, podle které knihovna danou třídu vyhledá a vytvoří podle ní databázovou tabulku. Každá entita má vazby s ostatními tabulkami, které jsou označeny anotací podle typu vazby. Například vazba mnoho: mnoho je označena anotací @ManyToMany. K této vazbě je také nutné doplnit další informace, podle kterých bude vytvořena asociační tabulka této vazby a vytvářena objekty z databázových záznamů. Jsou zde implementovány i vazby jeden: mnoho, které jsou označeny anotací @OneToMany. Zde je dále nutné definovat parametr druhé tabulky vazby, podle kterého je tato vazba vytvořena.

```java
@Entity
public class Machine {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @ManyToOne
    @JoinColumn(nullable = false)
    private Customer customer;

    @OneToMany(mappedBy = "machine")
    private Set<Inspection> inspections;

    @ManyToMany(fetch = FetchType.LAZY,
            cascade = {CascadeType.PERSIST,
                    CascadeType.MERGE})
    @JoinTable(name = "category_machine",
            joinColumns = {@JoinColumn(name = "machine_id")},
            inverseJoinColumns = {@JoinColumn(name = "category_id")})
    private Set<Category> categories = new HashSet<>();

    @ManyToMany(fetch = FetchType.LAZY,
            cascade = {CascadeType.PERSIST,
                    CascadeType.MERGE})
    @JoinTable(name = "sparePart_machine",
            joinColumns = {@JoinColumn(name = "machine_id")},
            inverseJoinColumns = {@JoinColumn(name = "sparePart_id")})
    private Set<SparePart> spareParts = new HashSet<>();

    @OneToMany(mappedBy = "machine")
    private Set<MachineHistory> machineHistories;

    @NotBlank
    @Size(max = 255)
    private String name;

    @NotBlank
    @Size(max = 255)
    @Column(unique = true)
    private String code;

    @Column
    private Date nextInspectionAt;

    @Size(max = 255)
    private String note;

    @Column
    private Boolean isBoughtLocally;

    @Column
    private Date boughtAt;

    @Column
    private Date createdAt;

    @Column
    private Boolean isWorking;

    @Column
    private Boolean isDeleted;

    @ManyToOne
    @JoinColumn(nullable = false)
    private SystemUser createdBy;
```

Zdrojový kód 2: Implementace záznamů strojů do JPA (zpracování vlastní)

```java
@Entity
public class Inspection {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @ManyToOne
    @JoinColumn(nullable = false)
    private Machine machine;

    @OneToMany(mappedBy = "inspection")
    private Set<InspectionHistory> inspectionHistories;

    @Column
    private Date doneAt;

    @Size(max = 255)
    private String result;

    @Size(max = 10000)
    private String description;

    @Size(max = 255)
    private String bill;

    @Size(max = 255)
    private String price;

    @CreatedDate
    private Date createdAt;

    @Column
    private Boolean isDeleted;

    @ManyToOne
    @JoinColumn(nullable = false)
    private SystemUser createdBy;
```

Zdrojový kód 3: Implementace záznamů strojů do JPA (zpracování vlastní)

Implementace datové vrstvy pomocí JPA pro záznamy kontrol proběhla obdobně jako u implementace záznamů strojů. Každý záznam má unikátní ID a vazby k jiným objektům ORM. Pro entitu stroje a kontroly jsou také ukládány záznamy o historii záznamů.

4.2.2 Implementace servisní vrstvy

Servisní vrstva slouží pro implementaci logiky celého systému. Servisní vrstva využívá datové vrstvy pro přístup do databáze a na základě implementované obchodní logiky se získanými daty manipuluje. Spring Boot pro označení tříd servisní vrstvy používá anotaci @Service. Spring Boot po spuštění aplikace dokáže najít třídy s danou anotací a vytvořit instance těchto tříd na definovaná místa. V implementovaném systému jsou tyto třídy nejčastěji vkládány do tříd kontrolerové vrstvy.

```java
@Service
public class CustomerServiceImpl implements CustomerService {

    private final CustomerRepository customerRepository;
    private final CustomerHistoryRepository customerHistoryRepository;
    private final UserService userService;

    @Autowired
    public CustomerServiceImpl(CustomerRepository customerRepository,
                               CustomerHistoryRepository customerHistoryRepository,
                               UserService userService) {
        this.customerRepository = customerRepository;
        this.customerHistoryRepository = customerHistoryRepository;
        this.userService = userService;
    }

    public Customer getCustomer(Long customerId) {
        Customer customer = customerRepository.findByIdAndIsDeleted(customerId, false)
                .orElseThrow(() -> new PropertyNotFoundException("Zákazník id: "
                        + customerId));
        if (customer.getDeleted()) {
            throw new PropertyNotFoundException("Zákazník id: " + customerId);
        }
        return customer;
    }

    public List<CustomerDTO> getAll() {
        return ((List<Customer>) customerRepository.findByIsDeleted(false)).stream()
                .map(this::convertToCustomerDTO).collect(Collectors.toList());
    }

    public void save(CustomerDTO customerDTO) {
        Customer customer = convertToCustomer(customerDTO);
        customer.setCreatedAt(new Date(System.currentTimeMillis()));
        customer.setCreatedBy(userService.getCurrentUser());
        customer.setDeleted(false);
        Customer newCustomer = customerRepository.save(customer);
        saveActionToHistory("Vytvořen nový zákazník: " + newCustomer.toString(),
                newCustomer);
    }

    public void update(CustomerDTO customerDTO, Long customerId) {
        Customer customer = this.getCustomer(customerId);
        Customer updatedCustomer = updateCustomer(customer, customerDTO);
        Customer newCustomer = customerRepository.save(updatedCustomer);
        saveActionToHistory("Aktualizován zákazník: " + newCustomer.toString(),
                newCustomer);
    }

    public List<CustomerHistory> getHistory(Long id) {
        return customerHistoryRepository.findByCustomer_Id(id);
    }

    public List<CustomerDTO> getDeleted() {
        return ((List<Customer>) customerRepository.findByIsDeleted(true)).stream()
                .map(this::convertToCustomerDTO).collect(Collectors.toList());
    }

    public void renew(Long customerId) {
        Customer customer = customerRepository.findByIdAndIsDeleted(customerId, true)
                .orElseThrow(() -> new PropertyNotFoundException("Zákazník id: "
                        + customerId));
        customer.setDeleted(false);
        Customer created = customerRepository.save(customer);
        saveActionToHistory("Obnoven zákazník: " + created.toString(), created);
    }
}
```

Zdrojový kód 4: Servisní vrstva pro záznamy o zákaznících (zpracování vlastní)

V zobrazeném kódu je implementovaná servisní vrstva, slouží pro správu záznamů o zákaznících. Do metod třídy jsou data vkládána v objektech třídy CustomerDTO, které slouží pro přenos dat z formulářů, kde v servisní vrstvě jsou data z DTO objektu vyjmuty a vloženy do Customer objektů. Customer objekty mohou být následně uloženy do databáze pomocí customerRepository třídy. Pro každou akci provedenou na záznamu zákazníka je uložen do databáze záznam o této akci. Pokud servisní vrstva dostane do stavu, ve kterém není možné pokračovat, upozorní kontrolerou vrstvu tím, že vyhodí definovanou vyjímku, podle které je dále uživatel informován o vzniklé chybě.

4.2.3 Kontrolerová vrstva

Kontrolerová vstva slouží pro přijímání a zpracování požadavků od klientské části aplikace. Ke zpracování požadavků kontrolerová vrstva většinou využívá servisní vrstvu. Po zpracování přijatého požadavku kontrolerová vrstva vrátí do klientské části html soubor, nebo přesměruje uživatele na jinou adresu. Přesměrování uživatele se stane po každém úspěšném POST požadavku. Druhý případ se děje většinou po zpracování požadavku s POST hlavičkou. Po každé úspěšné vykonané akci je uživatel informován o výsledku akce.

Každý kontroler musí mít anotaci @Controller, podle které je Spring Boot hledá kontrolerové třídy. Každá routa musí mí také svoji anotaci, ve které je definována URL cesta a HTTP metoda požadavku. Například anotaci @GetMapping(„/all/customer“) je definována GET metoda na dané URL.

V části kódu níže je zobrazena třída kontroleru, která slouží pro zpracování požadavků týkajících se záznamů o zákaznících. V konstruktoru třídy je vložena třída servisy, která slouží pro zpracování logiky požadavků, které mají za úkol manipulovat s daty záznamů zákazníků.

```java
@Controller
public class CustomerController {

    private final CustomerService customerService;

    @Autowired
    public CustomerController(CustomerService customerService) {
        this.customerService = customerService;
    }

    @PostMapping("/renew/customer/{id}")
    public String renewCustomer(@PathVariable Long id, Model model,
                                RedirectAttributes redirectAttributes) {
        customerService.renew(id);
        redirectAttributes.addFlashAttribute(SUCCESS_MESSAGE,
                "Zákazník byl úspěšně obnoven.");
        return REDIRECT_TO_DELETED;
    }

    @GetMapping("/new/customer")
    public String createForm(CustomerDTO customerDTO) {
        return CREATE_TEMPLATE;
    }

    @PostMapping("/new/customer")
    public String create(@Valid CustomerDTO customerDTO, BindingResult result,
                         RedirectAttributes redirectAttributes) {
        if (result.hasErrors()) {
            return CREATE_TEMPLATE;
        }
        customerService.save(customerDTO);
        redirectAttributes.addFlashAttribute(SUCCESS_MESSAGE,
                "Zákazník byl úspěšně vytvořen.");
        return REDIRECT_TO_INDEX;
    }

    @GetMapping("/update/customer/{id}")
    public String updateForm(@PathVariable("id") Long customerId, Model model) {
        model.addAttribute(CUSTOMER_DTO, customerService.getById(customerId));
        return UPDATE_TEMPLATE;
    }

    @PostMapping("/update/customer/{id}")
    public String update(@PathVariable("id") Long customerId,
                         @Valid CustomerDTO customerDTO, BindingResult result,
                         RedirectAttributes redirectAttributes) {
        if (result.hasErrors()) {
            customerDTO.setId(customerId);
            return UPDATE_TEMPLATE;
        }
        customerService.update(customerDTO, customerId);
        redirectAttributes.addFlashAttribute(SUCCESS_MESSAGE,
                "Zákazník byl úspěšně aktualizován.");
        return REDIRECT_TO_INDEX;
    }

    @PostMapping("/delete/customer/{id}")
    public String delete(@PathVariable("id") Long customerId,
                         RedirectAttributes redirectAttributes) {
        customerService.delete(customerId);
        redirectAttributes.addFlashAttribute(SUCCESS_MESSAGE,
                "Zákazník byl úspěšně smazán.");
        return REDIRECT_TO_INDEX;
    }

    @GetMapping("/history/customer/{id}")
    public String history(@PathVariable("id") Long id, Model model) {
        model.addAttribute(HISTORIES, customerService.getHistory(id));
        model.addAttribute(HEADER, "zákazníka");
        return HISTORY_TEMPLATE;
    }
}
```

Zdrojový kód 5: Kontrolerová vrstva pro záznamy o zákaznících (zpracování vlastní)

4.2.4 Zabezpečení

Zabezpečení přístupu bylo nastaveno pomocí knihovny Spring Boot Security, pomocí které jsme pro jednotlivé URL cesty určeny potřebná práva uživatele. Třída WebSecurityConfig nese anotaci @EnableWebSecurity, proto ji framework automaticky nastaví jako konfigurační třídu zabezpečení.

```
@Configuration
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

    UserDetailsService userDetailsService;

    @Autowired
    public WebSecurityConfig(@Qualifier("userDetailsServiceImpl")
                                        UserDetailsService userDetailsService)
        this.userDetailsService = userDetailsService;
    }

    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth)
            throws Exception {
        auth.userDetailsService(userDetailsService)
                .passwordEncoder(bCryptPasswordEncoder());
    }

    @Bean
    public BCryptPasswordEncoder bCryptPasswordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Bean
    public AuthenticationManager customAuthenticationManager()
            throws Exception {
        return authenticationManager();
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
                .authorizeRequests()
                .antMatchers("/").hasAnyAuthority(Roles.ADMIN.name(),
                Roles.MANAGER.name(), Roles.EMPLOYEE.name())
                .antMatchers("/delete/**", "/history/**", "/renew/**")
                .hasAnyAuthority(Roles.ADMIN.name(), Roles.MANAGER.name())
                .antMatchers("/user/**").hasAuthority(Roles.ADMIN.name())
                .antMatchers("/register").permitAll()
                .anyRequest().authenticated()
                .and()
                .formLogin().permitAll()
                .and()
                .logout().permitAll();
    }
}
```

Zdrojový kód 6: Nastavení přístupových práv k záznamům (zpracování vlastní)

```java
@Service
public class UserServiceImpl implements UserService {

    private final SystemUserRepository userRepository;
    private final RoleService roleService;
    private final BCryptPasswordEncoder bCryptPasswordEncoder;
    private final SystemUserHistoryRepository systemUserHistoryRepository;

    @Autowired
    public UserServiceImpl(SystemUserRepository userRepository,
                           RoleService roleService,
                           BCryptPasswordEncoder bCryptPasswordEncoder,
                           SystemUserHistoryRepository systemUserHistoryRepository) {
        this.userRepository = userRepository;
        this.roleService = roleService;
        this.bCryptPasswordEncoder = bCryptPasswordEncoder;
        this.systemUserHistoryRepository = systemUserHistoryRepository;
    }

    public SystemUser findByUsername(String username) {
        return userRepository.findByUserName(username);
    }

    public void delete(Long id) {
        SystemUser user = getUserById(id);
        userRepository.delete(user);
    }

    public void changePassword(PasswordDTO passwordDTO) {
        if (!passwordDTO.getNewPassword().equals(passwordDTO.getConfirmNewPassword())) {
            throw new InvalidPasswordChangeException();
        }
        SystemUser user = getUserById(passwordDTO.getUserId());
        user.setPassword(bCryptPasswordEncoder.encode(passwordDTO.getNewPassword()));
        SystemUser created = userRepository.save(user);
        saveActionToHistory("Aktualizováno uživatelské heslo administrátorem: "
                + created.toString(), created);
    }

    public void changeRole(RoleDTO roleDTO) {
        SystemUser user = getUserById(roleDTO.getUserId());
        user.setRole(roleDTO.getNewRole());
        SystemUser created = userRepository.save(user);
        saveActionToHistory("Aktualizována uživatelská role: " +
                created.toString(), created);

    }

    public void activate(Long id) {
        SystemUser user = getUserById(id);
        user.setBlocked(false);
        SystemUser created = userRepository.save(user);
        saveActionToHistory("Aktivován uživatel: " +
                created.toString(), created);
    }

    public void updatePasswordByUser(UserPasswordDTO userPasswordDTO) {
        if (getCurrentUser() != null) {
            SystemUser user = getCurrentUser();
            if (validateUserPasswordForm(userPasswordDTO, user.getPassword())) {
                user.setPassword(bCryptPasswordEncoder.encode(userPasswordDTO
                        .getNewPassword()));
                SystemUser created = userRepository.save(user);
                saveActionToHistory("Aktualizováno heslo uživatelem: " +
                        created.toString(), created);
            }
        }
    }
}
```

Zdrojový kód 7: Obchodní logika správy uživatele (vlastní zpracování)

Bezpečnost je také zajištěna pomocí tříd UserDetailService a UserService. UserDetailService se stará o přihlašování uživatelů udělování práv v systému. Tuto třídu dále používá Framework Spring Boot a programátor nemusí vykonávat žádné další akce. UserService se stará o aktivaci, deaktivaci, změnu hesla a role uživatele.

```
@Service
public class UserDetailsServiceImpl implements UserDetailsService {

    private final SystemUserRepository userRepository;

    @Autowired
    public UserDetailsServiceImpl(SystemUserRepository userRepository) {
        this.userRepository = userRepository;
    }

    public UserDetails loadUserByUsername(String userName) {
        SystemUser user = userRepository.findByUserName(userName);

        if (user == null || user.getBlocked()) {
            throw new UsernameNotFoundException(userName);
        }
        Set<GrantedAuthority> grantedAuthoritySet = new HashSet<>();
        grantedAuthoritySet.add(new SimpleGrantedAuthority(user.getRole().getName()));
        return new org.springframework.security.core.userdetails.User(user.getUserName(),
                user.getPassword(), grantedAuthoritySet);
    }
}
```

Zdrojový kód 8: Obchodní logika přihlášení uživatele (vlastní zpracování)

4.2.5 Uživatelské rozhraní

Pro implementaci klientské části aplikace byl zvolena knihovna Thymeleaf. Pro použití této knihovny stačí do projektu zahrnout „starter pack“ spring-boot-starter-thymeleaf. Poté je možné knihovnu Thymeleaf v projektu použít.

Pro vytvoření klientské části aplikace dále postačí vytvořit html soubor a pro data, která bude nutno doplnit z databáze, označíme pomocí Thymeleaf komponent. V kontrolerové vrstvě do daných Thymeleaf komponentů vložíme data, která získáme ze třídy servisní vrstvy.

Pro stylování klientské části aplikace bylo použito CSS Frameworku UIKit, která poskytuje stylování, které majiteli firmy nejvíce vyhovovalo.

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head th:fragment="headers">
    <link rel="stylesheet"
          href="https://cdn.jsdelivr.net/npm/uikit@3.5.8/dist/css/uikit.min.css"/>
    <meta name="viewport"
          content="width=device-width, initial-scale=1.0"/>
</head>
<body>
<nav th:fragment="navbar" class="uk-navbar-container uk-margin" uk-navbar="mode: click">
    <div class="uk-navbar-center">
        <ul class="uk-navbar-nav">
            <li class="uk-navbar-item uk-logo">
                MOTOHOBBY
            </li>
            <li class="uk-navbar-item">
                <a th:href="@{/all/customer}" href="#">
                    Zákazníci
                </a></li>
            <li class="uk-navbar-item">
                <a href="#" th:href="@{/all/machine}">
                    Stroje
                </a>
            </li>
            <li class="uk-navbar-item">
                <a href="#" th:href="@{/all/category}">
                    Kategorie strojů
                </a>
            </li>
            <li class="uk-navbar-item">
                <a href="#" th:href="@{/late/machine}">
                    Opožděné kontroly
                </a>
            </li>
            <li class="uk-navbar-item">
                <a href="#" th:href="@{/password}">
                    Změna hesla
                </a>
            </li>
            <li class="uk-navbar-item" sec:authorize="hasAuthority('ADMIN')">
                <a href="#" th:href="@{/user/all}">
                    Uživatelé
                </a>
            </li>
            <li class="uk-navbar-item">
                <a href="#" th:href="@{/logout}">
                    Odhlásit se
                </a>
            </li>
        </ul>
    </div>
</nav>
<div th:fragment="scripts">
    <script src="https://cdn.jsdelivr.net/npm/uikit@3.5.8/dist/js/uikit.min.js">
    </script>
    <script src="https://cdn.jsdelivr.net/npm/uikit@3.5.8/dist/js/uikit-icons.min.js">
    </script>
</div>
</body>
</html>
```

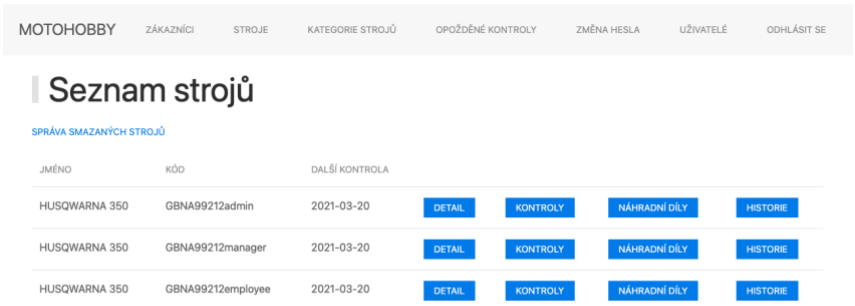
Zdrojový kód 9: Navigační menu implementované v Thymeleaf (zpracování vlastní)

Na obrázku níže je možné vidět vytvořený cyklus z pole hodnot „machines“, protože knihovna poskytuje různé metody manipulaci s daty. Například th:each="machine: ${machines}" umožňuje pomocí cyklu iterovat nad polem dat a následně s těmito daty pracovat. Dále je možné na základě vložených dat vytvářet linky na různá URL, například vlastnost `th:href="@{/new/machine(customerId=${customerId})` vytvoří ve výsledném html dokumentu link na formulář vytvoření stroje. Thymeleaf dále umožňuje propojit s bezpečnostní knihovnou a různé části html dokumentu zobrazovat jen uživateli s definovanými právy v systému. Příkladem této vlastnosti může být `sec:authorize="hasAnyAuthority('ADMIN', 'MANAGER')"`, který umožnuje danou část dokumentu zobrazit jen uživateli s rolí administrátora nebo manažera.

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <th:block th:include="fragments/general.html :: headers"/>
    <title>Seznam strojů</title>
</head>
<body>
<div th:insert="fragments/general.html :: navbar"></div>
<div th:if="${successMessage}" class="uk-alert-success" uk-alert>
    <a class="uk-alert-close" uk-close></a>
    <p th:text="${successMessage}">SuccesMessage</p>
</div>
<div class="uk-container">
    <h1 class="uk-heading-small uk-heading-bullet">Seznam strojů</h1>
    <a sec:authorize="hasAnyAuthority('ADMIN', 'MANAGER')" th:href="@{/renew/machine}"
       class="uk-button uk-button-link uk-button-small">SPRÁVA SMAZANÝCH
        STROJŮ</a>
    <table class="uk-table uk-table-divider">
        <thead>
        <tr>
            <th>Jméno</th>
            <th>Kód</th>
            <th>Další kontrola</th>
        </tr>
        </thead>
        <tbody>
        <tr th:each="machine: ${machines}">
            <td th:text="${machine.name}">John</td>
            <td th:text="${machine.code}">Deer</td>
            <td th:text="${machine.nextInspectionAt}">Miami</td>
            <td><a class="uk-button uk-button-primary uk-button-small"
                   th:href="@{/detail/machine/{id}(id=${machine.id})}">
                DETAIL
            </a>
            </td>
            <td><a class="uk-button uk-button-primary uk-button-small"
                   th:href="@{/filter/inspection(machineId=${machine.id})}">
                KONTROLY
            </a>
            </td>
            <td><a class="uk-button uk-button-primary uk-button-small"
                   th:href="@{/detail/part/machine/{id}(id=${machine.id})}">
                NÁHRADNÍ DÍLY
            </a>
            </td>
            <td sec:authorize="hasAnyAuthority('ADMIN', 'MANAGER')"><a
                    class="uk-button uk-button-primary uk-button-small"
                    th:href="@{/history/machine/{id}(id=${machine.id})}">
                HISTORIE
            </a>
            </td>
        </tr>
        </tbody>
    </table>
    <div th:if="${customerId}">
        <a th:href="@{/new/machine(customerId=${customerId})}">
            <span uk-icon="plus"></span></a>
    </div>
</div>
<div th:insert="fragments/general.html :: scripts"></div>
</body>
</html>
```

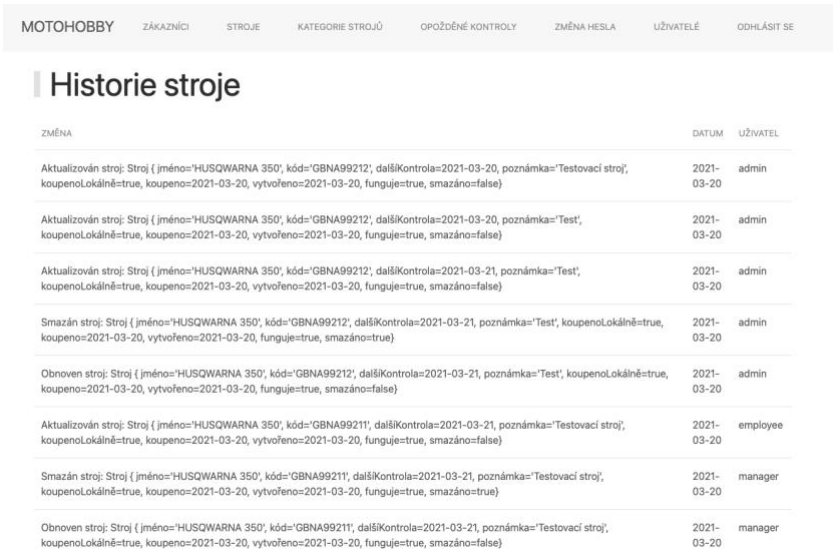
Zdrojový kód 10: Zobrazení seznamu strojů (zpracování vlastní)

Na obrázku níže je možné vidět zobrazení sekce Seznamu strojů. Na seznamu jsou zobrazeny základní údaji o stroji a odkazy na stránky, kde je možné najít detailní informace o stroji. Na obrázku níže je zobrazený pohled na Seznam strojů s právy administrátora nebo manažera. Zaměstnanci nemají možnost sledovat historii strojů, proto je pro ně odkaz na historii strojů skryt.



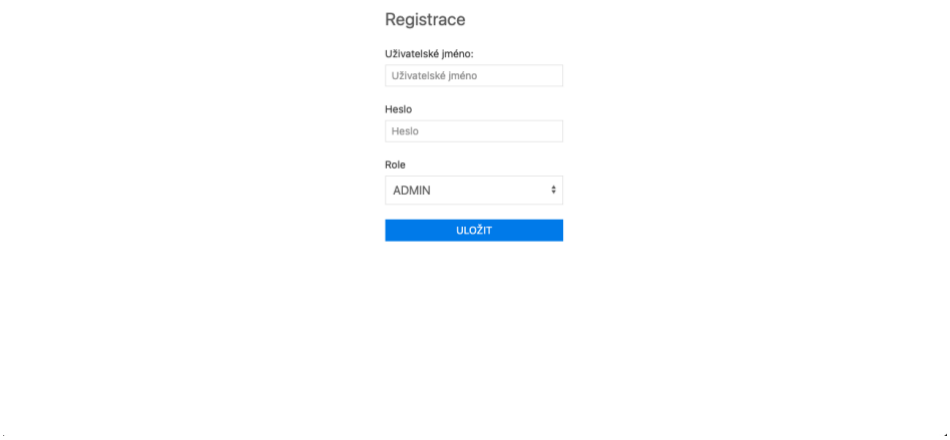
Obrázek 27: Zobrazení seznamu strojů (zpracování vlastní)

Na obrázku níže je možné vidět sekci Historii stroje. Pro každý historický záznam je ukládán nový stav stroje, datum změny a jméno uživatele, který změnu provedl.



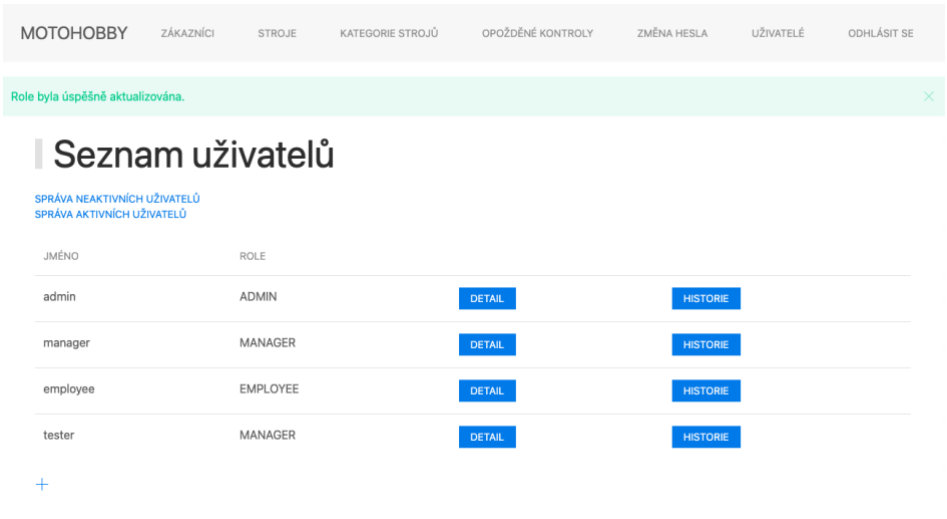
Obrázek 28: Historie stroje (zpracování vlastní)

Na obrázku 29 je možné viděl registrační formulář. Tento formulář je dostupný i pro neregistrované uživatele. Zde si mohou neregistrovaní uživatelé vytvořit účet, který následně projde schvalovacím procesem.



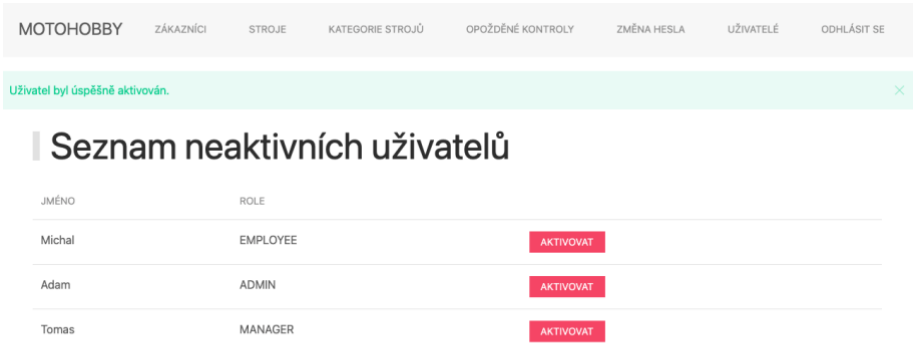
Obrázek 29: Registrace uživatele (zpracování vlastní)

Na obrázku 30 je možné vidět sekci Seznam uživatelů, která je přístupná pouze pro administrátory. Zde mohou administrátoři vidět všechny uživatele a jejich základní údaje. Jsou zde například odkazy na správu aktivních a neaktivních uživatelů, kde mohou administrátoři aktivovat, nebo deaktivovat uživatelské účty.



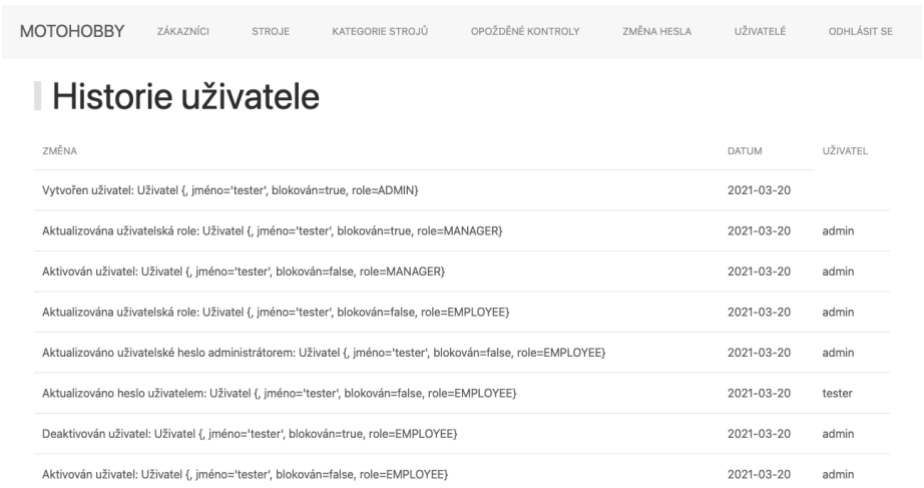
Obrázek 30: Správa uživatelů (zpracování vlastní)

Na obrázku níže je možné vidět sekci Seznam neaktivních uživatelů. Jsou zde zobrazeni všichni čerstvě registrovaní uživatelé, kteří ještě nejsou aktivování a také uživatelé, kteří byli z nějakého důvodu deaktivování.



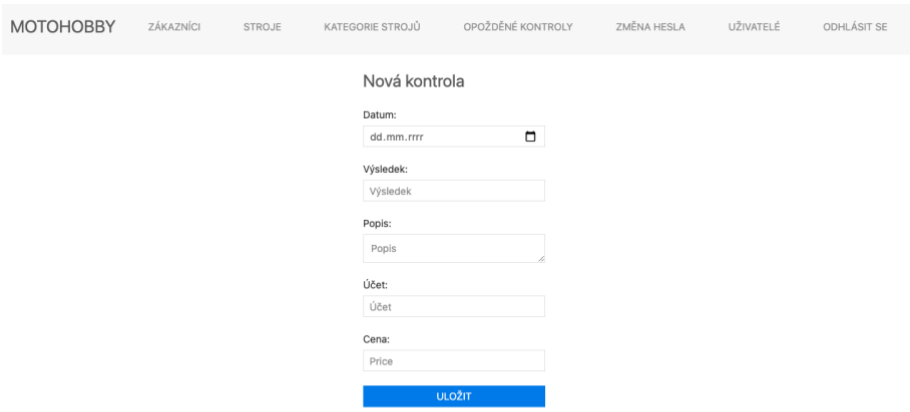
Obrázek 31: Správa neaktivních uživatelů (zpracování vlastní)

Na obrázku 32 je možné vidět zobrazení sekce Historie uživatele. Jsou zde zobrazeny všechny změny, které byly pro uživatele provedeny. Pro každou změnu je ukládán nový stav uživatele, datum provedení a uživatel, který změnu vykonal. U historie je možné vidět prázdné pole pro uživatele změny. Toto pole je prázdné, protože se jedná o registraci, kterou provedl nepřihlášený uživatel.



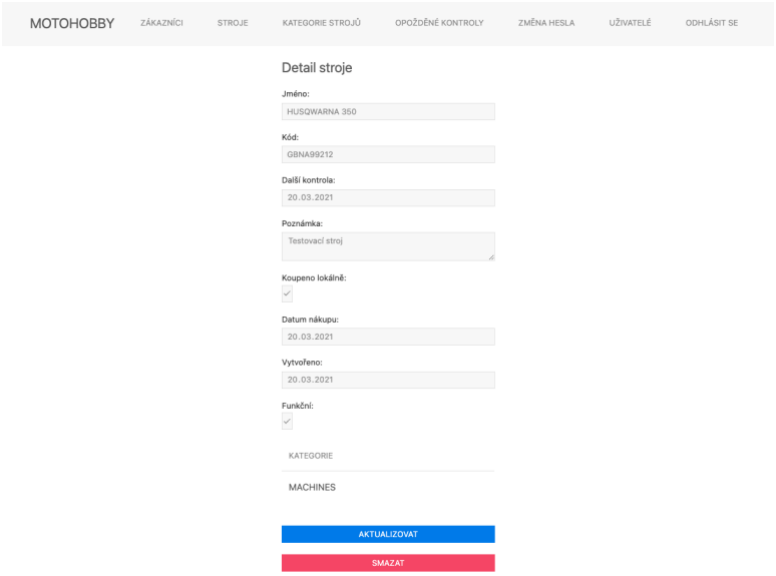
Obrázek 32: Historie uživatele (zpracování vlastní)

Na obrázku 33 je možné vidět zobrazení formuláře pro vytvoření záznamu o garanční prohlídce. Pro každou garanční prohlídku je možné uložit údaj o datu provedení, výsledku kontroly, popisu kontroly, čísla účtenky a ceny za kontrolu.



Obrázek 33: Vytvoření garanční prohlídky (zpracování vlastní)

Na obrázku 34 je možné vidět zobrazení detailu záznamu stroje. Pro každý stroj je možné ukládat všechny důležité záznamy o stroji. Stroje je také možné přiřazovat do různých kategorií. Kategorie stroje je možné vidět na spodní části detailu.



Obrázek 34: Detail stroje (zpracování vlastní)

4.2.6 Připojení k databázi

Propojení mezi vrstvou aplikační logiky a databázovou vrstvou je implementováno pomocí knihovny Java Persistence API. JPA podle URL připojení vytvoří zabezpečené spojení s databází a vytvoří databázové schéma podle definice vytvořené v aplikaci. Jako databázový relační systém je použit objektový a relační databázový systém PostgreSQL.

Připojení k databázi je třeba definovat v souboru application.properties, který se nachází v kořenovém souboru aplikace. V konfiguračním souboru je také nutné nastavit strategii vytváření databáze. V tomto konfiguračním souboru je databáze aktualizována po každé změně ve schématu databáze nastavené pomocí JPA.

```
spring.jpa.hibernate.dll-auto=update
spring.datasource.platform=postgres
spring.datasource.url=jdbc:postgresql://localhost:5432/inspekce
spring.datasource.username=uzivatel
spring.datasource.password=heslo
```

Zdrojový kód 11: Nastavení připojení k databázi (zpracování vlastní)

Závěr a výsledky

V první části diplomové práce je vysvětlena teorie, která je důležitá pro správné zpracování praktické části diplomové práce. Teoretická část práce je zaměřena především na analýzu systému pomocí jazyka UML. Jsou zde vysvětleny základy objektové analýzy, statického a dynamického modelování. Dále jsou zde vysvětleny základní principy vývoje Java aplikací a základní rozdělení Java knihoven. V teoretické části práce jsou také vysvětleny základní principy použití frameworku Spring Boot a databázového systému PostgreSQL.

Ve druhé části diplomové práce byly shromážděny požadavky na systém na základě kterých, bylo možné začít provádět analýzu požadovaného systému. Pro analýzu systému byly vytvořeny diagramy tříd, diagramy případů užití a diagramy aktivit. Na základě vytvořených diagramů bylo poté možné začít s implementací systému. Tím byly oba dílčí cíle splněny.

Implementace systému proběhla pomocí stanovených technologií, a to pomocí frameworku Spring Boot, knihovny Thymeleaf a databázového systému PostgreSQL. Po dokončení implementace byl systém otestován a nalezené chyby odstraněny. Při implementaci systému autor čerpal ze znalostí, které získal během studia a také z externích zdrojů, které jsou uvedeny v kapitole Seznam použitých zdrojů.

Hlavními cíli práce bylo navržení a implementace systému podle zadaných kritérii. Výsledkem této práce je funkční systém, který je připrav k použití. Tím byly splněny hlavní cíle práce.

Do budoucna by bylo možné systém rozšířit o přístup zákazníků do systému, kteří by mohli v systému sledovat stav kontrol jejich servisovaných strojů a data nadcházejících kontrol.

5 Seznam použitých zdrojů

Arrington, C T. 2002. *Enterprise Java with UML*. Wiley. ISBN 9780471013754

Bruckner Tomáš, V.J.B.A. 2012. *Tvorba Informačních Systémů*. Grada. ISBN 9788024741536

Buyaa. 2009. *Object-Oriented Programming with Java: Essentials and Applications*. Tata McGraw-Hill. ISBN 9780070669086

Conaway, C F, M Page-Jones, and L L Constantine. 2000. *Fundamentals of Object-Oriented Design in UML*. Dorset House Pub. ISBN 9780201699463

de Oliveira, C E, G L Turnquist, and A Antonov. 2018. *Developing Java Applications with Spring and Spring Boot: Practical Spring and Spring Boot Solutions for Building Effective Applications*. Packt Publishing. ISBN 9781789539134

Downey, A B, and C Mayfield. 2019. *Think Java: How to Think Like a Computer Scientist*. O'Reilly Media. ISBN 9781492072478

Drake, J D, and J C Worsley. 2002. *Practical PostgreSQL*. O'Reilly Media. ISBN 9781449310103

Fowler, M. *Destilované UML*. Grada Publishing. ISBN 9788027128464

Ganeshan, A. 2016. *Spring MVC: Beginner's Guide*. Packt Publishing. ISBN 9781785885648

Herout, P. 2008. *Učebnice Jazyka Java*. Kopp. ISBN 9788072323555

Chiarelli, A. 2016. *Mastering JavaScript Object-Oriented Programming*. Packt Publishing. ISBN 9781785888267

Chonoles, M J, and J A Schardt. 2011. *UML 2 For Dummies*. Wiley. ISBN 9781118085387

Loy, M, P Niemeyer, and D Leuck. 2020. *Learning Java: An Introduction to Real-World Programming with Java*. O'Reilly Media. ISBN 9781492056249

Masterson, C. 2017. *Java: Advanced Guide to Programming Code with Java*. E.C.Publishing. ISBN 9781543043242

MATHA, M P. 2008. *Object-Oriented Analysis and Design Using UML: An Introduction to Unified Process and Design Patterns*. PHI Learning. ISBN 9788120333222

Niemeyer, P, and D Leuck. 2013. *Learning Java: A Bestselling Hands-On Java Tutorial*. O'Reilly Media. ISBN 9781449372491

Obe, R, and L S Hsu. 2012. *PostgreSQL: Up and Running*. O'Reilly. ISBN 9781449326333

Pecinovsky, R. 2013. *OOP - Learn Object Oriented Thinking & Programming*. Tomas Bruckner. ISBN 9788090466180

Pilone, D, and N Pitman. 2005. *UML 2.0 in a Nutshell: A Desktop Quick Reference*. O'Reilly Media. ISBN 9780596552312

Merunka, V, Pergl R, and Pícka M. 2005. *Objektově orientovaný přístup v projektování informačních systémů.* Česká zemědělská univerzita v Praze. ISBN 80-213-1090-1

Quatrani, T. 2000. *Visual Modeling with Rational Rose 2000 and UML*. Addison-Wesley. ISBN 9780201699616

Rajput, D. 2018. *Mastering Spring Boot 2.0: Build Modern, Cloud-Native, and Distributed Systems Using Spring Boot*. Packt Publishing. ISBN 9781787125148

Roubalová, E. 2015. *Java*. Albatros Media a.s. ISBN 9788025145838

Rumbaugh, J R, and M R Blaha. 2011. *Object-Oriented Modeling and Design with UML*. Pearson Education. ISBN 9780133002171

Sarin, A, and J Sharma. 2012. *Getting Started With Spring Framework*. CreateSpace Independent Publishing Platform. ISBN 9781480013971

Scarioni, C, and M Nardone. 2019. *Pro Spring Security: Securing Spring Framework 5 and Boot 2-Based Java Applications*. Apress. ISBN 9781484250525

Schildt, H. 2014. *Mistrovství - Java*. Computer Press. ISBN 9788025141458

Sierra, K, and B Bates. 2005. *Head First Java*. O'Reilly. ISBN 9780596800765

Sintes, A. 1997. *Sams Teach Yourself Object Oriented Programming in 21 Days*. Pearson Education. ISBN 9780132715546

Tapper, J, J Talbot, and R Haffner. 2004. *Object-Oriented Programming with ActionScript 2.0*. New Riders. ISBN 9780735713802

Tarandach, I, and M J Coles. 2020. *Threat Modeling*. O'Reilly Media. ISBN 9781492056522

Unhelkar, B. 2005. *Verification and Validation for Quality of UML 2.0 Models*. Wiley. ISBN 9780471734314

6 Přílohy

Příloha 1 – zdrojový kód aplikace (repair_manager.zip)