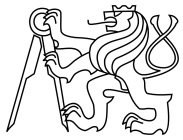
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Návrh a implementace aplikace pro údržbářské služby hotelu
Student:	Bc. Evgenia Filkina
Vedoucí:	Ing. David Buchtela, Ph.D.
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce zimního semestru 2016/17

Pokyny pro vypracování

Cílem této diplomové práce je analýza, návrh, implementace a testování aplikace pro údržbářské služby hotelu. Výsledkem bude aplikace pro hlavního manažera úklidové a údržbářské služby hotelu a jeho podřízené. Typ aplikace (desktopová-mobilní-webová) pro jednotlivé role (manažer pracovník) bude určen během úvodní analýzy.

Metodika:
1) Proveďte analýzu uživatelských potřeb.
2) Pro jednolivé role (manažer, pracovník) zhodnoťte a porovnejte možné způsoby řešení (desktopová / mobilní / webová) aplikace.
3) Proveďte návrh vlastního řešení systému.
4) Implementujte návrh řešení.
5) Proveďte akceptační testy a zhodnoťte navržené řešení.

Seznam odborné literatury

Dodá vedoucí práce.

L.S.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 16. dubna 2014

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra softwarového inženýrství

Diplomová práce

Návrh a implementace údržbářského systému hotelu

Bc. Evgenia Filkina

Vedoucí práce: Ing. David Buchtela, Ph.D.

10. května 2016

Poděkování

Tímto bych chtěla poděkovat vedoucímu Ing. Davidu Buchtelovi, Ph.D.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 10. května 2016 .....

Odkaz na tuto práci

Filkina, Evegnia. *Návrh a implementace údržbářského systému hotelu.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2016.

Abstrakt

Tato práce se zabývá zejména analýzou a návrhem informačního systému pro údržbářské a úklidové služby hotelu. Na základě analýzy a průzkumu trhu je vybráno a porovnáno několik existujících řešení. Následující část se zabývá návrhem konkrétního nového řešení pro Many Glacier Hotel. Práce postupně přestavuje požadavky na systém, diagramy užití a analytický model systému. Dále rozebírá návrh testů a testovacích scénářů. Poslední částí práce je část samotné implementace informačního systému.

Klíčová slova Grails, Scrum, Hotel, Údržba

Abstract

The aim of this work is especially analysis and design of information system for maintenance and cleaning services of hotel. Based on analysis and market research there is chosen and compared a few existing solutions. Following chapter is about design of specific new solution for Many Glacier Hotel. This work gradually presents demands for system, use case diagrams and analytical model of system. Follow up chapter describes design of tests and test scenarios. Last part is dedicated to implemetation of described and analyzed infromation system.

Keywords Grails, Scrum, Hotel, Maintenance

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Projekt je zpracováván pro společnost Glacier Park Inc, která spravuje 5 hotelů v USA a Kanadě. 4 z 5 hotelů jsou umístěný v národním parku Glacier v USA ještě jeden hotel se nachází v národním parku Waterton Lakes v Kanadě. Glacier Park Inc má dlouhou historii, první hotel byl otevřen v roce 1913, na což je dáván velký důraz. Převážně to je vidět v designu každého hotelu nebo z nabízených služeb jako výlety retro autobusem nebo lodí. Hotely jsou umístěny v národních parcích, ve kterých je zakázaná jakékoli lidská činnost, která by mohla způsobit podstatnou změnu ekosystému, také v okolí hotelů nenabízí svoje služby žádný mobilní operátor a je hodně pomalé internetové připojení, což je hodně výhodné pro vedení společnosti, které se snaží zanechat původní stav hotelů a vyvolat u hostů pocit, že se nachází na počátku 20. století. Další vlastností hotelů Glacier Park Inc je to, že nabízí svoje služby jenom od začátku leta do půlky září. Glacier Park Inc. zajišťuje celou infrastrukturu hotelů například: praní prádla, restaurace, úklid, údržba. Zatím nejsou využity žádné externí služby. Na dané etapě společnost potřebuje vnést do svého provozu trochu inovací, protože kvůli zastaralému přístupu vykonávají hodně neefektivních a zbytečných činností a proto nesou finanční ztráty. Jednou z variant řešení daného problému je přechod na procesní řízení, které potřebuje mít za sebou určitou podporu ze strany informačních technologií.

Kapitola 1

Analýza a návrh

1.1 Vize

Vedení společnosti Glacier Park Inc. se snaží začít renovovat stávající běh hotelů. Kvůli své době trvání, rozsáhlosti a potřebným konzultacím byla jako první cíl vybrána implementace nedostačujících částí informačního systému. Jelikož v každém hotelu již jsou nasazeny odpovídající systémy pro rezervace pokojů, recepce, restaurace a účetnictví, bylo rozhodnuto nejdřív investovat do projektu podpory úklidových a údržbářských služeb. Kvůli specifice provozu a rozmístění hotelů společnosti ze všech možných byla vybrána varianta objednání řešení na míru a následná případná úprava pro každý z hotelů, i když jejich struktura je téměř stejná. Spravování a údržba systému by měla byt prováděna vlastními sílami v každém hotelu. Omezená doba provozu hotelů je v dané situaci výhodou. Sběr požadavků a analýza budou prováděny během letní sezóny, ve které je hotel otevřen, nasazení a pilotní provoz proběhne při přípravách na otevření hotelu. Pro jednoduchost v práci bude jako příklad použit jeden z hotelů společnosti.

1.1.1 Všeobecné požadavky na systém

- Jednoduchost
- Spolehlivost
- Snadná udržitelnost
- Bezpečnost
- Pružnost (schopnost rozvoje)
- Efektivní provozovatelnost

Výseledná aplikace by měla mít následující základní funkcionalitu

- Rozdělení do dvou částí mobilní a desktopové
- Možnost objednání potřebného zboží k práci
- Možnost vytváření rozvrhu pro zaměstnance
- Integrace se vstupním karetním systémem
- Možnost upravování stavu hotelu a jeho součástí

1.1.2 SWOT analýza

SWOT je metoda strategického plánování, která umožňuje odhalit a strukturovat případné silné (Strengths) a slabé (Weaknesses) strany objektu a jeho potenciální příležitosti (Opportunities) a hrozby (Threats).

- Strengths jsou vlastnosti projektu, které mu dávají přednost před ostatními v oblasti
- Weaknesses jsou vlastnosti, které projekt oslabují
- Opportunities vnější možné faktory, které dávají projektu dodatečné možnosti k dosažení cíle
- Threats vnější možné faktory, které můžou zkomplikovat dosažení cíle

Silné a slabé strany jsou faktory vnitřního prostředí předmětu analýzy. Příležitosti a hrozby jsou faktory vnějšího prostředí, které dokážou ovlivnit objekt zevně a nejsou objektem kontrolované. Objektem SWOT analýzy v daném případě bude projekt systému podpory pro úklidové a údržbářské služby hotelu.

1.1.2.1 Strenghts

- Projekt je realizován pro síť hotelů
- Výsledný systém zvýší rychlost provádění práce a její kvalitu
- Výsledný systém zjednoduší práci s dodavateli
- Výsledný systém zjednoduší spolupráci mezi hotely, posílání a získávání reportů o zaměstnancích, komunikaci s úsekem účetnictví
- Výsledný systém zaručí zvýšení efektivity práci na jednotlivých úsecích

1.1.2.2 Weaknesses

- Cena
- Doba trvání

1.1.2.3 Opportunities

- Zvyšující se zájem lidí o hotel
- Otevření dalších hotelů společností
- Vytváření informačního systému celého hotelu
- Snížení kvality a rychlosti poskytování služeb a zvětšující se kvůli tomu počet zaměstnanců

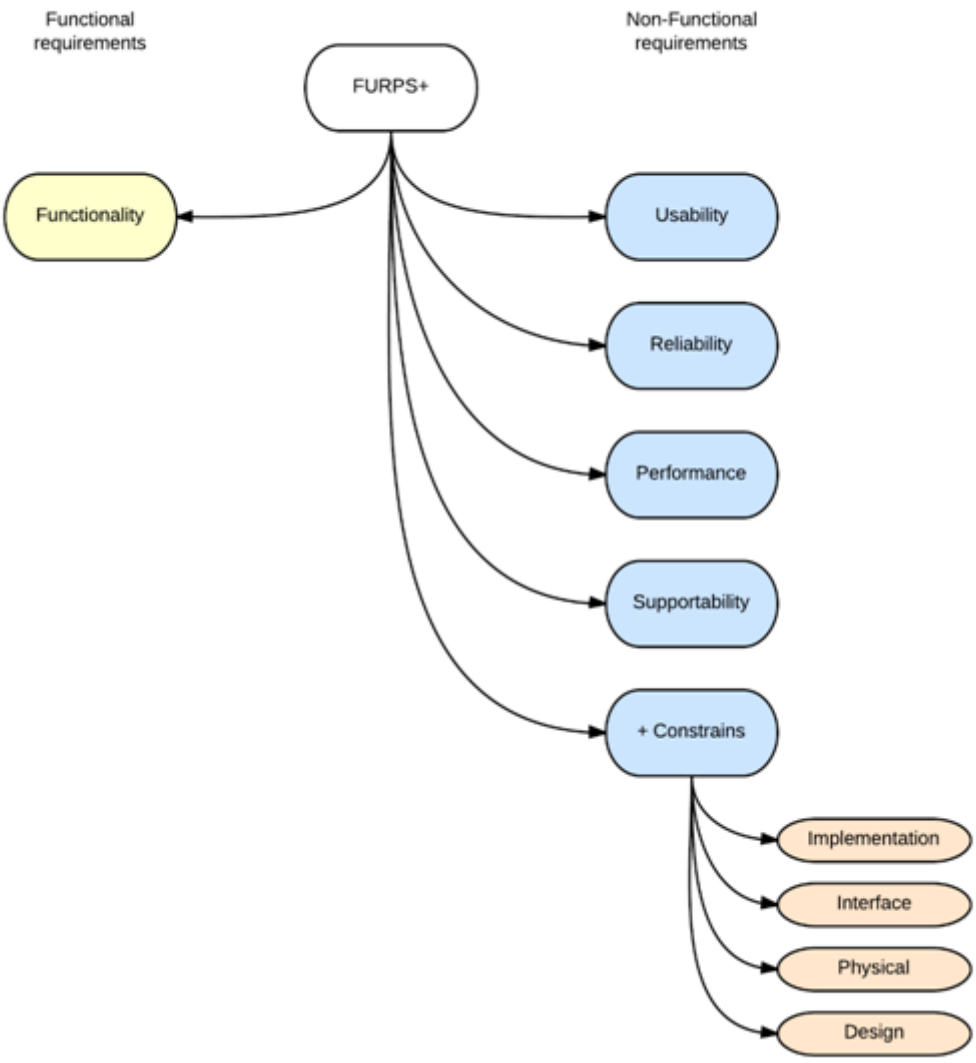
1.1.2.4 Threats

- Nedostatečné financování
- Přechod společnosti na outsourcování úklidových a údržbářských služeb
- Snížení zájmu klientu o hotel
- Zvyšující se konkurence

1.2 FURPS +

FURPS je metoda klasifikace funkcionálních a nefunkcionálních požadavků na software. Rozdělení požadavků je znázorněno na obr. 1.1.

- Funkčnost
- Vhodnost k použití
- Spolehlivost
- Výkon
- Udržovatelnost
- + nesmí se zapomenout na případné omezení
  - Omezení designu
  - Omezení implementace
  - Omezení rozhrání
  - Omezení fyzické



Obrázek 1.1: FURPS+: funkcionální a nefunkcionální požadavky

1.2.1 Funkcionální požadavky

Funkcionální požadavky popisují hlavní vlastnosti a funkce produktu, ale nemusí se týkat hlavního účelu systému.

- Audit — Nástroje pro sledování činnosti uživatele a systému prostřednictvím ukládání konkrétních typů událostí do logu.
- Licencování — Nástroje pro sledování, získávání, instalace a kontrolu používání licencí.
- Lokalizace — Podpora různých světových jazyků.
- Mail — Služby pro posílání a obdržení mailů.
- Online podpora —Možnost poskytování online podpory uživatelům.
- Tisk — Prostředky pro tisk dokladů.
- Evidence — Nástroje pro vytváření a obdržení zpráv.
- Bezpečnost — Nástroje obrany přístupu k určitým zdrojům informací.
- System management — Nástroje podporující řízení aplikací v rozděleném prostředí.
- Workflow —Podpora papírování a administrativ, které kromě jiné obsahují procesy ověřování a schvalovaní.

1.2.2 Usability

K požadavkům na vhodnost k použití patří:

- design a logika uživatelského rozhraní
- obrana před lidským faktorem
- dokumentace
- školení uživatelů

1.2.3 Spolehlivost

Spolehlivost zahrnuje následující charakteristiky systému:

- výpadky:
  - přípustná četnost výpadků,
  - průměrná délka výpadků a jejich závažnost,
  - možnost obnovy systému po výpadku a možnost předchozího rezervního zálohování dat
- předvídatelnost chování
- čas, za který je systém připravený k práci, časová dostupnost systému
- přesnost výpočtů

1.2.4 Výkon

Výkon systému reprezentují následující charakteristiky:

- rychlost práce, doba odezvy systému
- rezultativita, efektivita
- propustné schopnosti jako například přípustné množství uživatelů současně používajících produkt, počet požadavků, počet dotazů na DB a objem dotazovaných a posílaných dat za jednotku času.
- čas potřebný na obnovu systému po výpadku – rychlost obnovy, je nutné rozlišovat danou charakteristiku výkonu od charakteristiky spolehlivosti časové dostupnosti nebo možnosti obnovy systému po výpadku
- čas potřebný ke startu a ukončení práce v systému
- spotřeba zdrojů

1.2.5 Udržovatelnost

K udržovatelnosti můžeme přiřadit tyto charakteristiky:

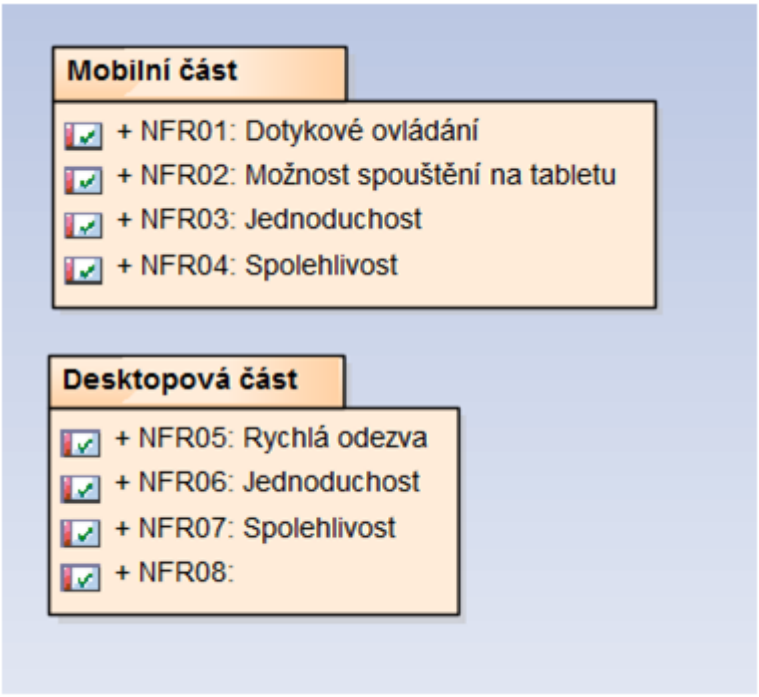
- testování
- rozšíření hranic systému, přidávání další funkcionality
- adaptace, přizpůsobení používání v konkrétním prostředí, například prostřednictvím předběžného ladění
- konfigurování
- kompatibilita
- udržování: oprava chyb, obnova dat, počet archivace a zálohování
- nasazení
- lokalizace
- přenosnost
- soulad s mezinárodními standardy

1.2.6 + Omezení

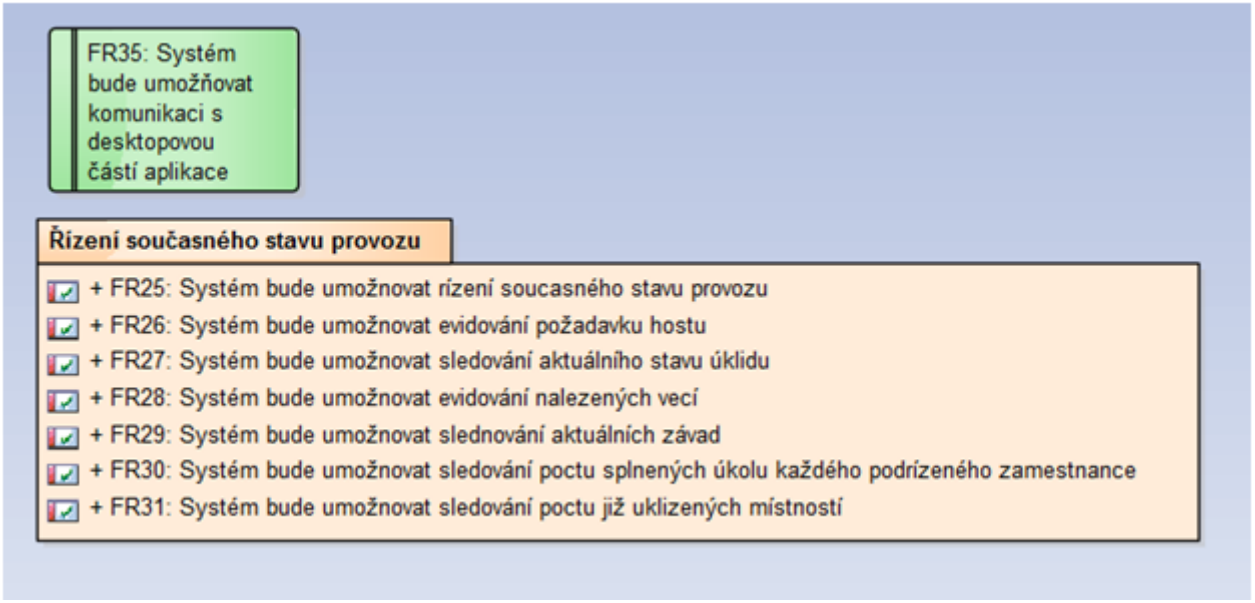
Metoda FURPS+ předpokládá 4 typy omezení:

- Omezení na design:
  - technologie
  - metoda
  - používané prostředky (programy)
  - ostatní
- Omezení na vývoj:
  - standardy vývoje,
  - standardy kvality SW, kódu
  - programovací jazyky
  - používané prostředky (programy, D a pod.)
  - omezení zdrojů
  - licenční omezení
  - omezeni na HW
  - ostatní
- Omezení na rozhraní:
  - formát dat
  - protokoly spolupráce
  - externí systémy
  - ostatní
- Fyzická omezení:
  - tvar
  - velikost
  - váha
  - teplota
  - vlhkost
  - omezení na vibraci
  - ostatní

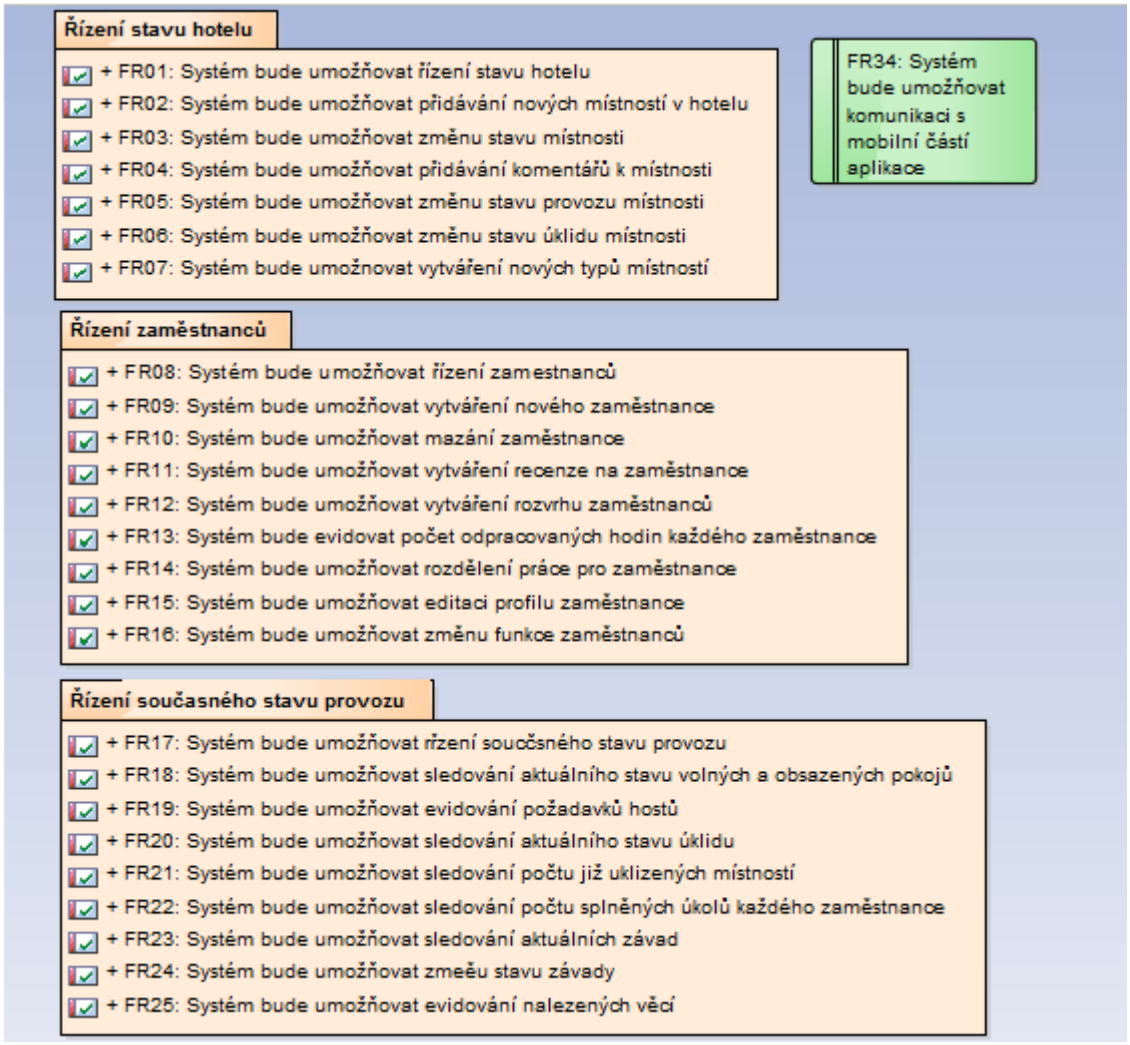
V závislosti na typu systému, místu jeho používání a jiných charakteristikách jsou zavaděná další omezení:



Obrázek 1.2: Nefunkční požadavky na systém



Obrázek 1.3: Funkční požadavky na mobilní aplikaci



Obrázek 1.4: Funkční požadavky na desktopovou aplikaci

- mezinárodní dohody: jazyk, jednotky měření
- autorské právo
- licencování
- legislativa
- omezení konkrétního odvětví

1.3 Účel produktu

1.3.1 Popis pracovního kontextu a situace

Zmíněný hotel je rozmístěn v přírodní rezervaci v horách, je otevřen pouze od května do září. Nejbližší město se nachází ve vzdálenosti 80 km. Místo nepokrývá žádná mobilní síť, připojení k internetu je pomalé a nestabilní. Hotel má dvě budovy a celkem 214 pokojů. Není provozován žádný systém pro vedení úklidu a údržby. Data o současném stavu hotelu, počtu volných a obsazených pokojů jsou dostupná na recepci. Množství různých prostředků pro úklid není monitorováno, občas se doplňuje. Všechna komunikace je prováděna telefonicky, přes vysílačku nebo osobně.

1.3.2 Cíle projektu

Cílem práce je vytvoření systému pro úklidové a údržbářské služby hotelu.

1.4 Uživatelé produktu

Oddělení úklidu ve vedení má hlavního manažera a jeho pomocníka. Kromě přímé zodpovědnosti za plnění všech činností úseku provádí objednání potřebného počtu kusů prádla, čistících prostředků, chybějícího vybavení pokojů, rozděluje práci na den podle počtů zaměstnanců, sestavuje rozvrhy, kontroluje kvalitu provedené práce. Další pozice jsou inspector, day porter, night porter, house man, housekeeper. Každý housekeeper ráno dostane vytištěný seznam pokojů, které musí uklidit do konce dne. Inspektor také obdrží seznam pokojů, ve kterých musí provést inspekci, v případě nesouladu se standardem upozornit housekeepera, aby to předělal a zapsat do seznamu. Závady nahlásí vedení oddělení, které to přepošle údržbářům. Day porter se stará o společenské místnosti hotelu jako záchody, chodby apod. Night porter vyřizuje žádosti klientů po skončení pracovní doby ostatních zaměstnanců, například donáší větráky, extra deky apod. House man zajišťuje včasné plnění skladů a odnáší ložní prádlo z pokojů.

1.4.1 Předmětná zkušenost

Kromě hlavního manažera předmětné zkušenosti nikdo ze zaměstnanců mít nemusí. Většina pracovníků jsou školáci nebo studenti pracující v hotelu přes léto. Žádné zkušenosti v oboru nemají a nepotřebují.

1.4.2 Zkušenost s technologií

Všichni zaměstnanci mají zkušenosti práce na PC, používání jednoho nebo několika mobilních operačních systému jako Android, iOS, Windows Mobile na úrovni uživatele.

1.4.3 Vzdělání, věk, jazyk

Většina pracovníku má jenom středoškolské vzdělání, část chodí na vysokou školu. Věkově jsou skoro všichni zaměstnanci v rozmezí 17-25 let. Hlavní manažer je starší. Každý zaměstnanec umí anglicky aspoň na střední úrovni.

1.4.4 Priorita uživatelů

- Key users - Klíčoví uživatelé budou používat výsledný produkt nejvíce času a v největším rozsahu. Klíčovými uživateli desktopové části systému bude hlavní manažer a jeho pomocník. Mobilní část především je určena pro inspektory.
- Secondary users
- Unimportant users

1.5 Nezbytná omezení

1.5.1 Omezení v oblasti řešení

- Desktopová část produktu musí používat operační systém Windows 7.
- Mobilní část produktu musí používat mobilní operační systém Android 4.4 «KitKat»
- Produkt musí podporovat komunikaci se současným rezervačním systémem hotelu.
- Produkt musí spolupracovat s karetním systémem vstupní kontroly.

1.5.2 Partnerské aplikace

- Nejdůležitější partnerskou aplikaci výsledného produktu bude interní rezervační a recepční systém hotelu. Dány systém bude poskytovat data o počtu uvolněných a obsazených pokojů, požadavcích hostů a možných závadách ve formátu XML.
- Další partnerskou aplikaci je karetní systém kontroly vstupu, pomocí kterého budou počítány odpracované hodiny každého zaměstnance a umožňován vstup do pracovních místností.

1.5.3 Předpokládané pracovní prostředí

- Většinu pracovního času zaměstnanci provádějí vestoje a v docela hlučném prostředí, což znamená, že zvukové signály v daném případě nejsou moc spolehlivé.

- Během dne je potřeba hodně častá úprava stavu pokojů, která se většinou provádí ve spěchu, z čehož plyne, že aplikace musí být hodně přehledná a kontrastní, nesmí obsahovat moc hluboké stromy záložek a všechno, co se používá nejčastěji, má být hned pod rukou.

- Hlavní nároky na práci inspektora jsou rychlost a spolehlivost, aby nemusel stejnou činnost provádět vícekrát, v případě zpoždění dokázal pomoct ostatním členům týmu. Jelikož hotel má několik budov a každý inspektor může být zodpovědný za pokoje v jeho různých částech a termín odevzdání pokojů je 15:00, práce inspektora je často spojená s běháním, rychlou prohlídkou a popřípadě úpravou pokojů.

- Práce na zařizování různých produktů potřebných k udržování hotelu v příslušném stavu probíhá většinou v dopoledních hodinách v hlavní kanceláři konkrétního úseku, ve které se nachází stolní počítač hlavního manažera, tiskárna. Tato také slouží jako místnost ke každodenním ranním schůzkám se zaměstnanci s rozdělením práce na den.

1.6 Hotový software

1.6.1 SumIT Front Office System Housekeeping Module

SumIT Front Office System je kompletním systémem pro podporu vedení hotelu, který se skládá z několika modulů. Například rezervační modul a front desk modul. SumIT také nabízí řešení pro úklidové služby - housekeeping module. Housekeeping module podporuje následující funkce :

- zjištění stavu úklidu hotelu celkem nebo konkrétního pokoje
- rozvrhy úklidu
- mapy prostorů
- historie úklidu
- vytváření různých druhů zpráv a seznamů

1.6.2 Digital Housekeeper

Aplikace Digital Housekeeper na rozdíl od předchozí je webová, což je obecně výhodou, ale ne v daném případe kvůli pomalému internet připojení. Digital Housekeeper umožňuje jednoduché a rozšířené hledání pokojů, vytváření zpráv o historii pokojů, zprávu kalendáře úklidu celkem a konkrétního pokoje zvlášť, přiřazení úkolů zaměstnancům. Novinkou aplikace je možnost vytváření a posílání dotazníku zákazníkům e-mailem ohledně kvality služeb. Výsledky jsou automaticky zaznamenány, když zákazník dotazník vyplní.

1.7 Rizika projektu

Riziko je neurčitá událost nebo podmínka, která může ovlivnit cíle projektu. Každé riziko má svoje zdroje a důsledky. Riziko se liší od problémů tím, že má dopad na budoucí potenciálně možné negativní výsledky. Na rozdíl od rizika problém reprezentuje potíže, které má projekt v současné době. Z rizika se může vyvinout v problém, jestli není správně a efektivně řízené. Proto je během vývoje IT projektu nutné dávat velkou pozornost rozpracování a využití metod řízení rizik. Pro řízení rizik v projektech jsou používané různé způsoby identifikace rizik, jejich kvalitativní a kvantitativní analýza. Také v etapě plánování projektu se provádí odhad stavů, aby se zabránilo výskytu rizik a byly vytvořeny možné způsoby reakce na ně.

1.8 Identifikace rizik

Především musíme určit rizika, která mohou mít negativní dopad na projekt – identifikovat je. Nejčastější používané zdroje informace pro identifikaci rizik jsou zkušenosti z předchozích projektů, lidí pracujících na projektu, odborníků, uživatelů, manažerů jiných projektu atd. Identifikace rizik je iterativní proces, to znamená, že první iteraci může provést projektový tým, další projektoví manažeři a poslední lidi přímo se nepodílející na projektu. Během dané etapy je důležité naznačit plány potenciální reakce a rozdělit rizika na skupiny:

- Organizační rizika
- Správní rizika
- Technická rizika
- Vnější rizika

Výsledkem identifikace rizik jsou:

- Rizika a jejich podmínky (činnost nebo okolí projektu, které mohou zvýšit pravděpodobnost výskytu rizika
- Triggery nebo příznaky rizik, které ukazuji na to, že se riziko už uskutečnilo nebo se uskutečni v nejbližší době

V praxi technická rizika v IT projektech nejsou nejzávažnější, protože jsou vyřešitelná. Největší dopad obvykle mají rizika jako nedostatečná účast nebo podpora vedení, snížení financování, neochota uživatelů apod.

ID a název rizika	01:Vysoké procento chyb při vývoji
Kategorie rizika	Projektový tým
Dopad rizika	Nízká kvalita výsledného produktu
Pravděpodobnost výskytu	Střední
Závažnost dopadu na projekt	Vysoká
Stupeň kontroly rizika	Vysoká
Preventivní opatření	Testování jednotlivých částí během vývoje a testování celého produktu před jeho vydáním: • Unit testy • Regresní testování • UAT • Smoke testování • Integrační testy
Rizikový plán	Nasazení funkčních části systému v termínu a oprava chyb, postupování podle SLA

ID a název rizika	02:Nízký rozpočet
Kategorie rizika	Produkt
Dopad rizika	Menší rozsah projektu
Pravděpodobnost výskytu	Nízká
Závažnost dopadu na projekt	Vysoká
Stupeň kontroly rizika	Vysoká
Preventivní opatření	Vytvoření detailního rozpočtu projektu a podmínek spolupráce v dané situaci v rámci SLA. Provedení analýzy a odhadu doby vývoje v MD, sestavení harmonogramu projektu
Rizikový plán	Ukončení práce na projektu podle podmínek v SLA

1.9 Případy užití

Provedení analýzy funkčních požadavků na systém pomáhá určit hranice systému a seznam aktérů. Při vytváření případů užití je důležité oddělit systém od jeho okolí, stanovit účastníky, jejích spolupráci se systémem a výsledek takové spolupráce.

- Hlavní manažer-úklidová služba
- Hlavní manažer-údržba

ID a název rizika	03:Překročení termínu odevzdání produktu
Kategorie rizika	Projektový tým
Dopad rizika	Penalizace za zpoždění
Pravděpodobnost výskytu	Střední
Závažnost dopadu na projekt	Nízká
Stupeň kontroly rizika	Vysoká
Preventivní opatření	Vytvoření detailního harmonogramu projektu a podmínek spolupráce v dané situaci v rámci SLA. Provedení analýzy a odhadu doby vývoje v MD
Rizikový plán	Nasazení funkčních částí systému v termínu a pokračování ve vývoji zbylých částí, postupování podle SLA

ID a název rizika	04:Nedostatečná analýza
Kategorie rizika	Projektový tým
Dopad rizika	Špatný návrh, možné chybějící nebo nadbytečné funkce
Pravděpodobnost výskytu	Střední
Závažnost dopadu na projekt	Vysoká
Stupeň kontroly rizika	Vysoká
Preventivní opatření	Konzultace se zákazníkem, uživateli výsledného produktu, odborníky
Rizikový plán	Doplnění původní analýzy o chybějící části nebo odstranění nadbytečných částí

ID a název rizika	05:Požadavky klienta nad rámec dohodnutého výsledku
Kategorie rizika	Produkt
Dopad rizika	Prodloužení doby vývoje projektu
Pravděpodobnost výskytu	Vysoká
Závažnost dopadu na projekt	Střední
Stupeň kontroly rizika	Nízká
Preventivní opatření	Vytvoření detailních požadavků, konfigurací projektu a podmínek spolupráce v dané situaci v rámci SLA.
Rizikový plán	Vytvoření nového SLA pro potřebnou funkcionalitu, postupování podle existujícího SLA

ID a název rizika	06:Nedostatečná spolupráce zákazníka
Kategorie rizika	Produkt
Dopad rizika	Prodloužení doby vývoje projektu, nedostatečná analýza
Pravděpodobnost výskytu	Nízká
Závažnost dopadu na projekt	Vysoká
Stupeň kontroly rizika	Nízká
Preventivní opatření	Vytvoření detailních postupu a podmínek spolupráce v dané situaci v rámci SLA
Rizikový plán	Postupování podle SLA

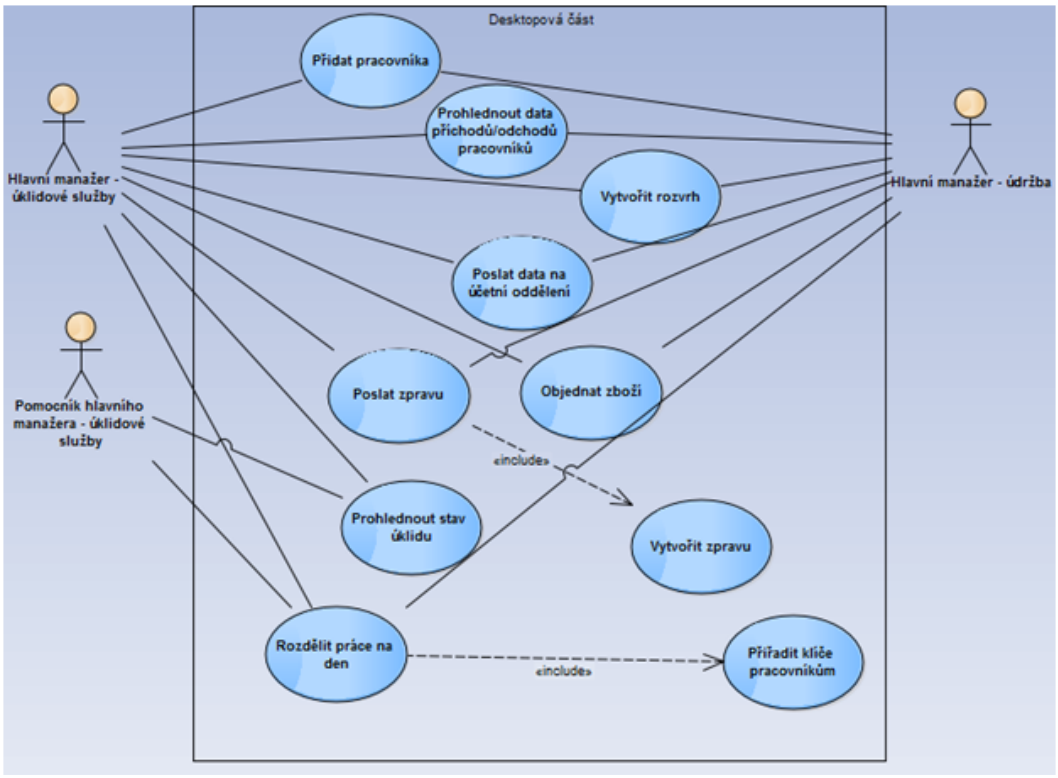
ID a název rizika	07:Nedostatečná kapacita nebo zkušenost projektového týmu
Kategorie rizika	Projektový tým
Dopad rizika	Prodloužení doby vývoje projektu, snížení kvality produktu
Pravděpodobnost výskytu	Střední
Závažnost dopadu na projekt	Vysoká
Stupeň kontroly rizika	vysoká
Preventivní opatření	Přidělení na projekt zkušenějších zaměstnanců, konzultace s externími odborníky
Rizikový plán	Přidělení dalších členů projektu, využití externích služeb

- Pomocník hlavního manažera
- Day Porter
- Night Porter
- Inspektor
- Údržbář

1.9.1 Příklady scénářů případu užití desktopové a mobilní aplikace

1.10 Analytický model tříd

Diagram tříd slouží pro představení statické struktury modelu systému pomocí tříd objektově orientovaného programování. Diagram tříd zobrazuje různé druhy vztahů mezi jednotlivými entitami byznys domény. Daný typ diagramu neobsahuje žádné informace o časovém průběhu systému. Mezi nejdůležitější vlastnosti dobré analytické třídy patří:

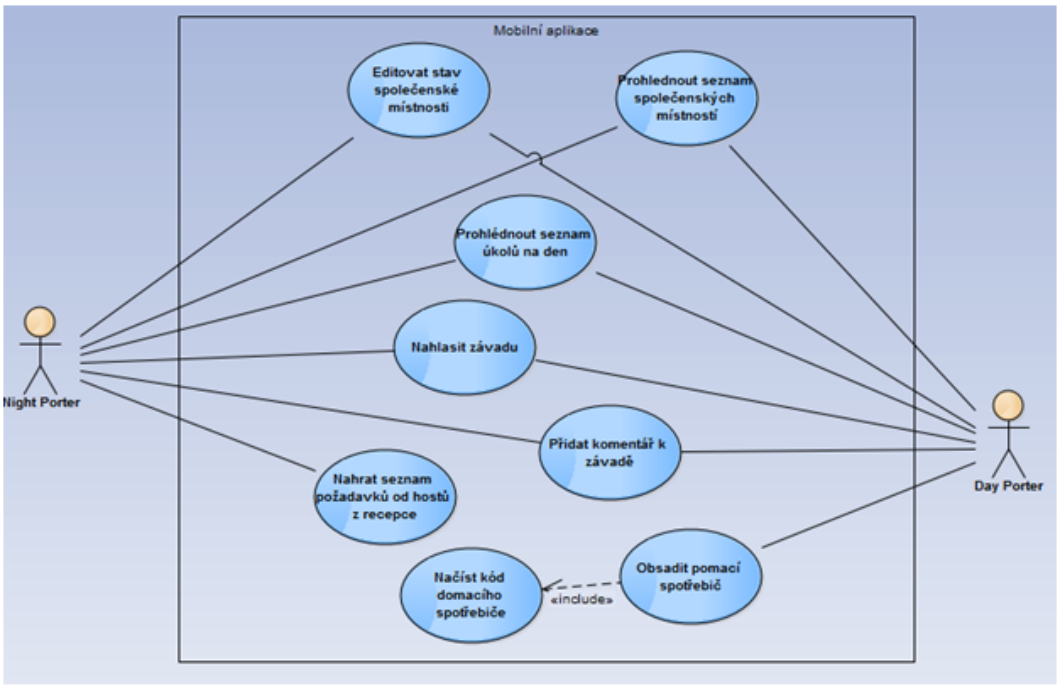


Obrázek 1.5: Případy užití pro desktopovou aplikaci

- Soudržnost
- Jednoduchost
- Abstrakce
- Správné pojmenování

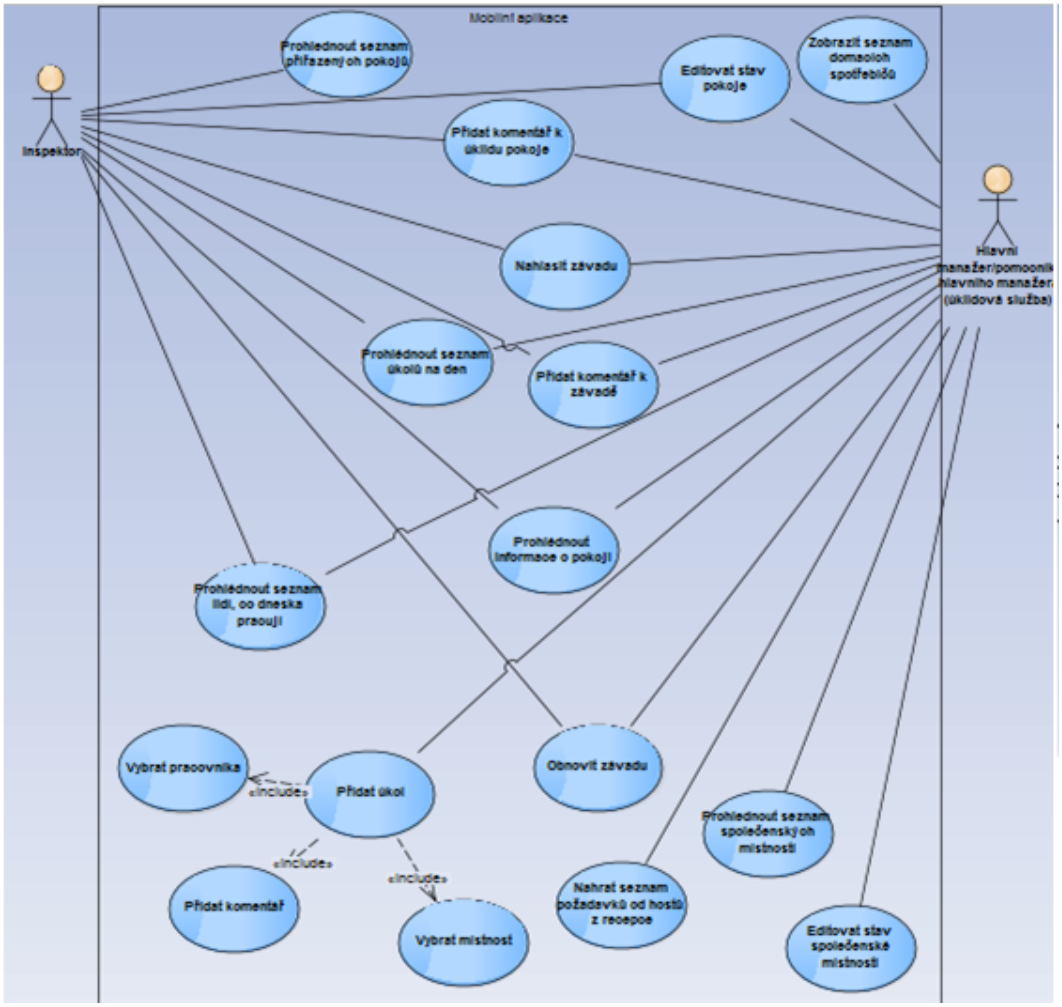
Základní entity a způsoby spolupráce mezi nimi se zjistí během sběru požadavků, jejich analýzy a analýzy případu užiti.

- Hotel
- Místnost – Třída místnost je předkem pro třídy Pokoj, Společenská místnost a Služební místnost.
- Nábytek
- Klíč – Každý pracovník dostane univerzální klíč, kterým může otevřít pokoje ve své pracovní oblasti.

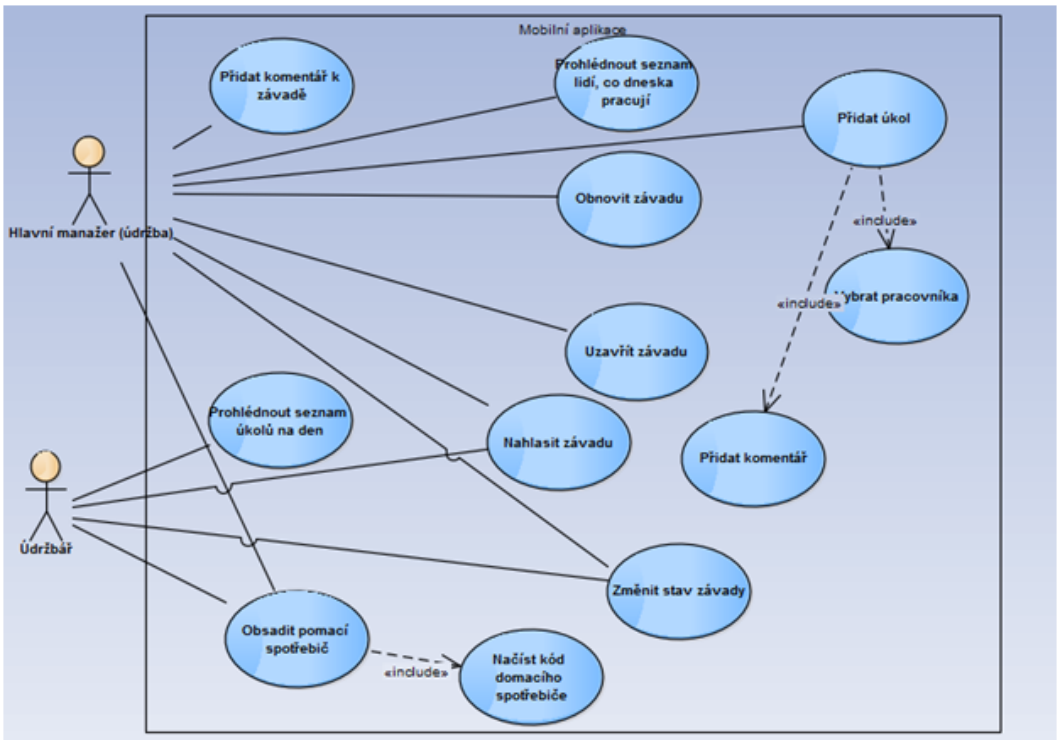


Obrázek 1.6: Případy užití pro mobilní aplikaci - day porter/night porter

- Nález
- Závada
- Host
- Pracovní úsek – V daném případe to jsou úsek údržby a úsek úklidu.
- Zaměstnanec – Třída zaměstnanec má reflexivní asociaci sama na sebe. V daném případě to popisuje vztah mezi nadřízeným a podřízeným zaměstnancem.
- Úkol
- Rozvrh
- Domácí spotřebič – Mezi domácí spotřebiče patří různé elektronické stroje používané pro práce na jednotlivých úsecích (vysavač..).
- Inventář – Různé druhy čisticích prostředků, nářadí, povlečení apod.



Obrázek 1.7: Případy užití pro mobilní aplikace - úklidové služby, hlavní manažer, inspektor



Obrázek 1.8: Případy užití pro mobilní aplikaci - údržba

1.11 Metodika vývoje systému

Metoda vývoje softwaru je souborem pravidel, prostředků a postupu používaných během životního cyklu vývoje softwaru, který se skládá z následujících kroků: definice požadavků, analýza a plánování, návrh, implementace, testování, nasazení a údržba. Různé druhy podobných frameworků se vyvíjely během druhé poloviny 20 století. Na vývoji některých z nich se podílely velké podniky. Ve většině případů konkrétní přístup nebo kombinaci více přístupů vybere vedení projektu nebo jeho vývojový tým. Toto rozhodnutí je hodně důležité, protože značně ovlivňuje následující běh projektu, jeho správu a testování produktu. Většina metod vývoje softwaru používá inkrementační nebo iterativní přístup.

- Inkrementační přístup – požadavky jsou rozdělené na skupiny, což znamená, že celý vývoj probíhá taky v několika cyklech a vypadá jako několik vodopádových modelů za sebou. Funkcionalita přibývá s každým dalším cyklem. Daný proces pokračuje, než systém nebude kompletně hotový. Výhodami inkrementačního modelu jsou flexibilita, možnost vracení částečně hotového produktu, který splňuje některé požadavky zá-

Název případu užití	Rozdělit práce na den		
Identifikace případu užití	UC01-desktopová část		
Cíl případu užití	Rozdělit úkoly pro pracující podřízené na konkrétní den		
Primární aktér	Hlavní manažer		
Sekundární aktér	Pomocník hlavního manažera		
Vstupní podmínky	Hlavní manažer nebo jeho pomocník je přihlášen do systému		
Výstupní podmínky	Každý pracovník má vytvořený seznam úkolů na den		
Základní scénář	Krok	Role	Akce
	1.	Systém	Zobrazí seznam pokojů
	2.	Aktér	Vybere oblast a přiřadí ji zaměstnanci, popřípadě napíše komentář, opakuje, než každá oblast nebude někomu přiřazená
	3.	Systém	Zobrazí seznam zaměstnanců a přiřazených oblastí
	4.	Aktér	Uloží seznam
	5.	Systém	Zkontroluje, jestli každá oblast patří někomu a uloží, pokud je to tak.
	6 .	Aktér	Vytiskne seznam úkolů pro každého pracovníka

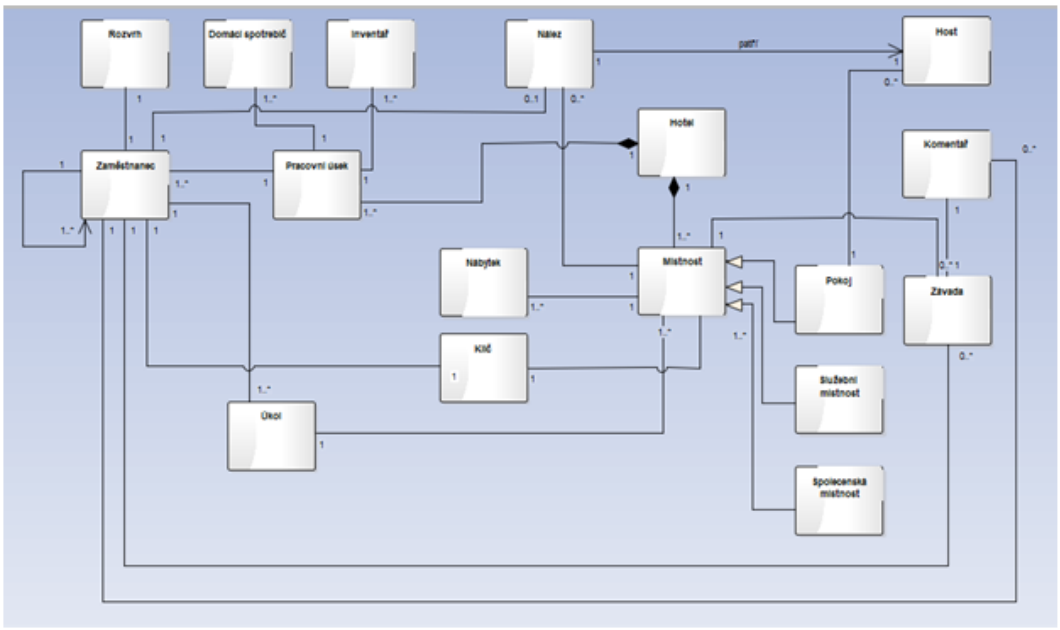
kazníka, po každé iteraci, možnost aplikace zkušenosti z předchozí iterace. Ale daný přístup má i svoje nevýhody, například větší celková cena projektu, špatná realizovatelnost bez dobrého plánování a návrhu. Během vývoje s použitím inkrementační metody se musí dávat velký důraz na to, aby v každém dalším cyklu se jenom neopravovaly chyby z předchozího.

- Iterativní přístup – se skládá z několika iterací, které většinou trvají několik týdnů. Konečným výstupem každé iteraci je funkční program splňující část požadavků, který prošel všemi etapami (kromě nasazení a údržby) životního cyklu softwaru. Za výhody iterativního přístupu mohou být považované dobrá a častá oboustranná komunikace se zákazníkem a potenciálním uživatelem, výsledkem které je produkt přesně odpovídající požadavkům, testování během celé doby práce na projektu, objevení konfliktů s požadavky na ranních etapách projektu, rovnoměrná zátěž všech účastníků v průběhu vývoje, flexibilita.

Název případu užití	Nahlásit závadu		
Identifikace případu užití	UC01-mobilní část		
Cíl případu užití	Nahlásit závadu v konkrétní místnosti		
Primární aktér	Hlavní manažer, pomocník hlavního manažera, day porter, night porter, instruktor, údržbář		
Sekundární aktér	-		
Vstupní podmínky	Aktér je přihlášen do systému		
Výstupní podmínky	Závada je vytvořená a je vidět při prohlížení pokoje nebo seznamu závad		
Základní scénář	Krok	Role	Akce
	1.	Systém	Zobrazí informace o konkrétním pokoji
	2.	Aktér	Vybere možnost „Přidat závadu“
	3.	Systém	Zobrazí formulář pro přidání nové závady
	4.	Aktér	Zadá typ závady a ostatní údaje, popřípadě napíše komentář a vybere možnost „Uložit“
	5.	Systém	Zkontroluje, jestli jsou vyplněné všechny povinné položky

Existující metody vývoje softwaru můžeme rozdělit na tradiční a agilní, které mají svoje výhody a nevýhody. Za neznámější tradiční metody jsou považované:

- Vodopádový model – daná metoda je jednoduchá na pochopení a používání. Každá část musí být kompletně hotová, než se může začít pracovat na další. Po ukončení práce na úseku projekt je revidován na soulad s požadavky. Velkým plusem daného přístupu je rozsáhlá dokumentace a jednoduchost – najednou se děla jenom jedna fáze. Minusem je fixní posloupnost kroků a malá flexibilita.
- Spirálový model – charakteristickým rysem spirálového modelu je zvláštní přístup k rizikům, které můžou ovlivnit organizaci životního cyklu vývoje. Příkladem takových rizik může být nedostatek odborníků, trvalé změny požadavků nebo špatné porozumění zadání. Každý závit spirály odpovídá části nebo verzi softwaru, během kterého se upřesňují cíle a charakteristiky projektu, dodávaná kvalita produktu a probíhá plánování dalšího úseku. Takovým způsobem se postupně konkretizují detaily projektu a nakonec se vybere nejvhodnější varianta, která bude realizovaná. Každý závit je rozdělen do 4 sektory:

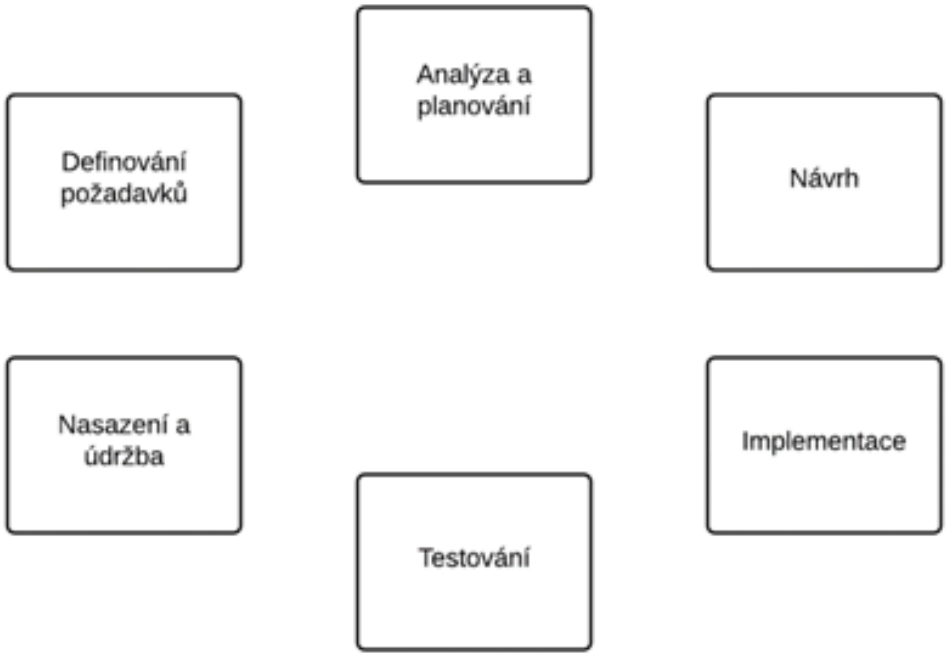


Obrázek 1.9: Analytický diagram tříd

- Ohodnoceni a zabráněni rizikům
- Definice cílů
- Implementace a testování
- Plánování Spirálový model se nejvíce hodí pro velké, drahé složité projekty.
- Rational Unified Process – RUP je dobře formalizovaný model, který jako modelovací jazyk používá UML. Největší důraz je kladen na počáteční fáze životního cyklu projektu – analýzu a návrh, což pomáhá snížit komerční rizika díky nalezení chyb na počátečních etapách vývoje. Vývoj je rozdělen na několik fází: zahájení, příprava, konstrukce a předávání, které se pak můžou skládat z iterací. Každá iterace je určená sadou cílů, které má splňovat.

Nejvíce populární agilní metody jsou:

- SCRUM metoda je postavená na seznamu praktik, které umožňují dodávat zákazníkovi plně funkční verzi produktu, která je implementovaná během striktně časově omezené iterace zvané sprint. Délka dané iterace je jasně stavena, může trvat od 1 do 4 týdnů, jednotlivé týmy si jí přizpůsobují svým potřebám. Takový přístup práce rozdělené na fixní a poměrně krátké časové intervaly zajišťuje SCRUM metodě vývoje softwaru



Obrázek 1.10: Postup vývoje systému

větší předvídatelnost a pružnost. V každém sprintu je rozpracovávaná a na konci přidaná do průběžné verze další funkcionalita podle priorit zákazníka. Daná funkcionalita je definovaná před začátkem sprintu na etapě plánování a není možné jí měnit během celkové doby trvání následující iteraci. SCRUM tým musí obsahovat členy třech rolí [2] :

- Product owner – je zodpovědný za definování náplně práce a její priority pro tým.
- SCRUM master – je vedoucí, který pomáhá týmu dodržovat pravidla SCRUMU a používat ho správným způsobem.
- Členy vývojového týmu – nejsou jenom programátoři ale i ostatní lidi podílející se na vývoji.

Další důležitou části SCRUM metodologie vývoje softwaru jsou schůzky, které zaručují průhlednost práce nejenom pro tým ale i pro ostatní zaujaté lidi. Neoddělitelnou části SCRUMU jsou následující typy setkání [2] :

- Backlog grooming

  - Plánování sprintu
  - Každodenní status – stand up
  - Demonstrace
  - Retrospektiva

  Všeobecně známým problémem SCRUM metodologie je velká ztráta času kvůli různým schůzkám jednáním a také během doby ukončení jednoho a začátkem dalšího sprintu, kdy se na tom stráví minimálně jeden den. Skoro třetina času při využívání SCRUM je použitá pro dodržení samotného procesu – každodenní stand upy, groomingy, retrospektivy, demo atd.

- Lean development – hlavní charakteristickou vlastnosti Lean development metody je v tom, že práce vykonávána týmem musí přinášet hodnotu, všechno, co bude implementováno, musí být projednáno se zákazníkem a použito v konečném produktu. Lean development řídí se následujícími principy [3] :
  - Všechno musí přinášet hodnotu
  - Odkládat přijetí rozhodnuti na co nejdelší možnou dobu
  - Co nejkratší doba dodání produktu
  - Motivace vývojového týmu
  - Kvalita systému na všech úrovních
  - Důraz na rozvoj a učení se z chyb
  - Vnímání produktu nejenom jako malých jeho částí ale jako celku
- Kanban – metoda vývoje Kanban je adaptací Toyota Production System, systému používaném na továrnách firmy Toyota, na vývoj softwaru. Během vývoje metodou Kanban musí se dodržovat 3 pravidla:
  - zmenšení počtů současně běžících úkolů (work in progress)
  - vizualizace progresu
    * fyzická nebo online nástěnka rozdělená na sloupce odpovídající fázi, ve které se nachází konkrétní požadavek (například: ve frontě, vývoj, testování, nasazení, ukončen)
    * lístečky s požadavky, umístěné do fronty podle priority. V každém sloupci se současně může nacházet jenom na začátku nastavený počet lístečků.
    * oddělené místo pro mimořádné požadavky, ve kterém se zároveň může nacházet jenom jeden požadavek.
  - měření a případné zmenšení doby strávené na vývoj jednotlivého požadavku

- Extreme Programming (Extrémní programování) – daná metoda vznikla z problému způsobených dlouhými cykly tradičních metod vývoje softwaru a jmenuje se tak, protože zvedá obvyklé principy a postupy na extrémní úroveň. Extrémní programování je charakterizováno krátkými cykly, iterativním plánováním (plánovací hra), neustálou komunikaci s konečným uživatelem. Extrémní programování používá následující praktiky [4] :
  - Plánovaní
  - Krátké vývojové cykly
  - Metafory – zákazník a programátor definují sadu metafor, které popisují, jak systém má fungovat
  - Jednoduchý design
  - Refaktoring
  - Párové programování – kód je psán dvěma programátory za jedním počítačem
  - Kolektivní zodpovědnost
  - Průběžná integrace
  - 40 hodinový pracovní týden
  - Sledování programovacích konvencí
- Future Driven Development – daná metodologie byla poprvé použita pro velké a složité projekty pro bankovní aplikace. Na rozdíl od ostatních metod Future Driven Developement nepokrývá celý životní cyklus projektu a nejvíce se soustřeďuje na návrhovou a realizační části vývoje. Celá doba vývoje se děli na 5 fázi, tří první se provádí jenom jednou na začátku práce na projektu a dvě ostatní se provádí iterativně, což realizuje agilní praktiky jako vetší flexibilita vůči požadavkům a jiným byznys potřebám.
  - Návrh high level designu celkového sytému
  - Sestavení seznamu požadavků (vlastností)
  - Rozdělení požadavků mezi programátory a seřazení podle priority
  - Detailnější rozpracování designu požadavků pro danou iteraci – vytváření sekvenčních diagramu a jejich synchronizace s ostatními části systému
  - Implementace požadavků z předchozího bodu

Pro výběr metodologie vývoje nejdůležitější je vědět

- cíle projektu, jeho druh a téma

- podstatným bodem pro ohodnocení je výše a způsob financování, které značně ovlivňuje všechny strany životního cyklu
- rozsah projektu, jeho seznam požadavků
- definitivnost požadavků, jestli se během vývoje budou přidávat nějaké další požadavky nebo měnit se stávající
- dobu, za kterou zákazník chce mít hotový produkt
- specifikum oblasti, pro kterou je projekt rozpracováván
- počet a závažnost rizik souvisejících s projektem během jeho životního cyklu

V daném případě můžeme říct, že se jedná o:

- projekt, cílem kterého je vytváření informačního systému pro úklidové a údržbářské služby hotelu
- omezené financování
- poměrně malý projekt
- požadavky se můžou měnit během vývoje
- čas je omezen 1 rokem - byznys analýza je možná během 5 měsíců letní sezony
- specifickými požadavky pro danou oblast můžou být navázání komunikace s karetním systémem vstupu a výstupu, rezervačním systémem hotelu; složitější situace s internetovým připojením v oblasti přírodní rezervace
- jediným větším rizikem projektu je doba trvání, plně funkční produkt musí být nasazen a v provozu před začátkem další letní sezony provozu hotelu

Z výše uvedených bodu můžeme posoudit, že pro daný projekt se nejvíce hodí jedna nebo kombinace více různých agilních metod vývoje softwaru. V současné době SCRUM je jednou z nejpopulárnějších a nejvíce popsaných agilních metod. Hodně firem se teď snaží svůj vývoj udělat více agilním a vybírají SCRUM. Daná metoda se dá nejčastěji potkat jako již implementovaná a úspěšné používaná. SCRUM jako agilní metoda dokáže rychle reagovat na změny a na případné chyby v implementaci nebo návrhu, což snižuje počet a závažnost některých rizik. Po analýze různých metodologií vývoje softwaru a požadavků na produkt bylo rozhodnuto se řídit během vývoje metodou SCRUM, která nehledě na její nevýhody má hodně plusu, které můžou být

dobře aplikovatelné v daném případě. Jedním z podstatných důvodu je omezený časový interval. Ale kvůli tomu, že SCRUM má svůj zvláštní proces a celá doba vývoje se skládá z iterací, fixní doba odvedená na rozpracování bude lépe zvládnutelná. Možná neviditelné na první pohled ale ne méně důležité je to, že s práci v SCRUM týmu má zkušenosti více lidi, což oblehči případné hledání členů do týmu nebo Scrum mastera. Pro lepší porozumění metodě SCRUM dole jsou popsané nejdůležitější pojmy používané během vývoje [5]. Většina z nich není přeložena do češtiny a skoro vždy se zmiňuje v anglickém originálu, proto nebudou přeloženy ani v této práci.

- Burndown Charts – burndown chart je graf, který ukazuje vztah mezi objemem zbývající práce na ose Y a časem na ose X. Jelikož od začátku ke konci Sprintu objem práce by se měl zmenšovat, graf se musí blížit k 0 na obou osách, což znamená, že úkoly naplánované na sprint byly splněné. Daný způsob se dobře hodí k plánování doby trvání projektu a počítání průměrné velocity daného teamu během sprintu.

- Daily Scrum Meetings – každodenní krátká schůzka (kolem 15 minut), cílem které je synchronizace a aktualizace informací mezí členy teamu. Doporučuje se daily scrum meeting provádět na začátku dne a za jeho běhu neprojednávat žádné technické detaily. Každý účastník by měl odpovědět na 3 otázky:

  - Co jsem udělal od posledního daily scrum meetingu?
  - Co plánují udělat do příštího daily scrum meetingu?
  - Co mi brání dělat svou práci co nejefektivnějším způsobem?

- Definition of Done – je nástrojem ve větší míře se týkající kvality produktu než jeho funkcionality. Definition of done je seznam popisující požadavky, které musí být splněné na konci sprintu, většinou se připravuje během sprint planning meetingu. Daná metoda umožňuje scrum teamu lépe vidět, co konkrétně se musí udělat a jestli udělaná práce je připravena k releasu nebo ne.

- Epic – se říká velké user story, která pak může být rozdělena na více menších story rozpracovaných během několika sprintů kvůli své velikosti. Epic může popisovat nějakou velkou a podstatnou funkcionalitu nebo požadavek, která ještě nebyla podrobně rozpracovaná do konce.

- Fibonacci Sequence – Fibonacciho posloupnost je používaná pro odhady složitosti úkolu (story pointů), nebo doby na jeho vypracování, jelikož přesnost odhadů nebývá moc správná, když se jedná o velké a složité user story. Výhoda dané metody je v tom, že posloupnost začíná rychle růst, což nutí team více přemýšlet o složitosti úkolu.

- Impediment – různé druhy překážek, které zdržují nebo nedovolují teamu plnit úkoly.
- Increment – je funkční část softwaru, která je vypracovaná během jednoho sprintu a pak přidaná k celkovému produktu.
- Product Backlog – je seřazený podle priority seznam požadavků na produkt (user story). Za prioritizaci product backlogu je zodpovědný product owner daného projektu. Priorita jednotlivých user story se může lišit během času a může být ovlivněná třeba zbývající do releasu dobou, počtem členů v teamu nebo zavaděním nových důležitějších pro byznys požadavků. Udržování product backlogu v pořádku (seřazený podle priority, ohodnocené podle složitosti a doby na vypracování user story, kvalitně popsané požadavky s definition of done) je hodně prospěšní pro efektivní práci teamu, protože v nestandardních situacích je zdroj pravdy, o který se team může opřít a sledovat.
- Product Backlog Grooming – schůzka, během které členu scrum teamu upřesňují, definují a hodnotí user story, každý ze svého pohledu. Jako výsledek product backlog gtoomingu product owner má jasnější představu o technologické a ostatních stranách práce na konkrétní user story, což mu pomůže později seřadit požadavky s větší přesnosti. Daný způsob práce je hodně efektivní, pokud se dodržuje během schůzky její rámec a členy teamu jsou předem připravení a vědí, o co se jedná. Jinak to může být jenom plýtvaní časů, který by mohl být použit na vývoj produktu.
- Product Backlog Item – je požadavek nebo část požadavku, která může být rozpracovaná (implementována a otestována) během jednoho sprintu. Také se do product backlogu můžou přidávat chyby (bugy) nebo jakákoliv značná pro vývoj projektu činnost z pohledu product ownera.
- Product Backlog Item Effort – odhad složitosti nebo někdy doby práce na jednotlivém požadavku. Product backlog item effort metrika pak může být použita při vyplnění burndown chartu. Při odhadech členy teamu musí být hodně opatrní, aby se nepřecenili a počítali se i s nejmenšími detaily, jelikož pak by mohlo dojit k neodevzdání naplánované práci včas a ke skluzu. Proto se používá výše popsaná metoda Fibonaccího posloupnosti.
- Product Owner Role – je jedná ze tři rolí v scrum metodologii. Product owner reprezentuje v teamu zákazníka, jeho zájmy a požadavky. Člověk na dané pozici musí přesně vědět, co od produktu potřebuje a kdy to potřebuje mít. Neméně důležité je schopnost dobře popsat a vysvětlit svoje požadavky programátorům nebo ostatním členům teamu.
- Release – je proces instalace a nastavení, tak aby produktu byl připravený k použití zákazníkem. Frekvenci releasů každý team si nastavuje

podle svého rytmu a potřeb zákazníka, může být po každém sprintu nebo ob sprint atd.

- Scrum Board – je často využívanou praktikou v hodně scrum teamech fyzická nástěnka synchronizována se systémem řízení projektu. Jejím velkým přínosem je to, že fyzické vytváření a přesouvání jednotlivých úkolu donucuje členy scrum teamu k časté komunikace. Scrum board je nástroj, který se nejvíce hodí a používá během daily scrum meetingů, jelikož obsahuje stejné informace jako systém řízení projektu a je více interaktivní. Synchronizace se systémem řízení projektu je největším požadavkem na scrum board, jinak v nejlepším případě nebude mít žádný přínos.

- Scrum Roles – role, co může plnit každý z členů teamu. Jsou tři druhy roli z scrum projektu:

  - Scrum Master
  - Product owner
  - Člen vývojového týmu

  Ve většině případů v scrum teamech bývá jeden scrum master, jeden product owner a více členů vývojového týmu.

- Scrum Master Role – Scrum master je člověk zodpovědný za dodržování scrum kultury v teamu. Jeho práce je směrována na zjednodušení práce ostatních členu týmu, zlepšování komunikace mezi product ownerem a teamem, vyřešení a odstraňování impedimentů.

- Sprint – je název pro iteraci v Scrumu, což většinou trvá maximálně 30 dnů. Délka takové iteraci je fixní a musí se dodržovat, nastaví se na začátku práce na projektu, ale může být změněná podle potřeb vývojového týmu. Během každého sprintu se má vyvíjet předem vybraná funkcionalita.

- Sprint Demo Meeting – je schůzka na konci každé iteraci, na které jednotlivé členy scrum teamu říkají a popřípadě i ukazuje, které z patřících do sprintu a do releasu úkolů udělal.

- Sprint Planning Meeting – sprint planning meeting je schůzka, na které se řeší, které user story budou naplánované do sprintu, hodnotí se (pokud nebyly ohodnocené dřív na sprint backlog groomingu) a přiřazují se členům teamu, které na tom budou pracovat. Také na sprint planning meetingu pro zpřesnění odhadů a průhlednější plánování může se hodit počítání pracovních hodin, které každý člen teamu může strávit v daném sprintu na práci na uvedených úkolech.

1.11.1 Sprint Retrospective Meeting

Z názvu tohoto meetingu vyplývá, že se jedná o schůzku, která se koná na konci každého sprintu. Jejím cílem je to, aby tým zhodnotil ukončený sprint. Zároveň otevírá diskuzi nad tím zda je možné zlepšit některé procesy, za cílem dosažení větší efektivnosti při vývoji. Pro Scrum Mastera je to také příležitost zjistit od týmu překážky, které blokovaly jeho práci a pracovat na jejich odstranění.

1.11.2 Sprint Task

Základní jednotka každého sprintu. Tasky mohou obsahovat tradiční kroky vývojového cyklu (vývoj, testování, design, UAT, tvorba dokumentace..). Je důležité, aby všechny tasky byly měřitelné, nejlépe v hodinách, kolik bude trvat jejich realizace. Pro účely Scrum je vhodné, aby tyto tasky neměly trvat více jak jeden den. Je to z toho důvodu, že jsou pak lépe měřitelné na denních stand-upech (naplánovalo se 10 tasků, udělalo se 8)

1.11.3 Stakeholder

Stakeholder ve smyslu Scrum technologie je osoba (skupina lidí, zákazník. . .), která má zájem na tom aby byl projekt úspěšný.

1.11.4 Story Point

Story point je jednotka, kterou se měří složitost/úsilí, které je potřeba pro realizaci User Stories. Story points slouží k výpočtu toho, kolik User Stories může zvládnout tým za jednu iteraci.

1.11.5 Team

Scrum team je skupina 5 až 9 osob, které pracují na dodávce pro zákazníka. Tyto týmy jsou obvykle složeny napříč jednotlivými odděleními (pozicemi) tzn. Programátory, analytiky, QA experty, testery, Designery atd. Je vhodné aby takový tým pracoval pohromadě, nejlépe v jedné místnosti.

1.11.6 Team Member

Jedná se o člena Scrum teamu.

1.11.7 User Story

User story je krátký a jednoduchý popis dané funkcionality systému z pohledu osoby, která ji požaduje (typicky uživatel systému nebo zákazník). Obvykle se používá šablona definovaná Mikem Cohnem: Jako [role] potřebuji [funkcionalita], protože [důvod]. Příkladem může být: “Jako uživatel potřebuji mít možnost změnit své přihlašovací heslo, protože se může stát, že ho zapomenu”.

Nebo “Jako správce systému potřebuji mít možnost přidělovat práva jednotlivým uživatelům, protože každý nesmí mít přístup k editaci obsahu”.

1.11.8 Velocity

Velocity indikuje průměrný objem Product Backlogu proměněný do Incrementu během jednoho Sprintu Scrum teamem. Daná metrika je hodně užitečná za běhu Sprint Planning Meetingu, jelikož představuje střední produktivitu Teamu a může zabránit naplánování do Sprintu Sprint Tasků, které převyšují pracovní kapacitu Teamu. Také pomocí Velocity jsou dobře počitatelné datumy případných releasů.

1.12 Návrh uživatelského rozhraní

Návrhem uživatelského rozhraní (UI – User Interface) je rozuměn návrh té části aplikace, která interaguje s jejím uživatelem. Za úkol si klade několik cílů:

- Snížit co nejvíce dobu, za kterou se uživatel naučí se systémem pracovat
- Efektivně zrychlit práci uživatele se systémem
- Snížit množství chyb, kterých se uživatel při práci se systémem dopustí
- Konzistence jednotlivých částí systému

Schopnost naučit se ovládat daný program co nejrychleji získává stále více na důležitosti. Jako příklad je možné uvést restaurační aplikaci, pomocí které si zákazník od stolu objednává jídlo, aniž by musel čekat na číšníka. Pokud zákazník není schopný během několika chvil zjistit jak se aplikace ovládá, je taková aplikace k ničemu a podnik tak pouze přichází o hosty, jelikož část dá přednost jiným restauracím, kde si jsou jídlo schopni objednat. Jiným příkladem mohou být naopak rozsáhlé systémy, kde se neočekává, že při školení zákazníka s ním budou vyzkoušeny všechny funkce systému. V takovém případě by měl zákazník být schopný použít už nabyté znalosti z předchozích částí. Zrychlení práce se systémem je jednou z podstatných částí návrhu uživatelského rozhraní a také pravděpodobně nejdůležitější částí návrhu aplikace pro hotely. Jelikož se jedná o nový systém, který nahrazuje používání telefonů (vysílaček), papírových seznamů úkolů atd. musí být jeho uživatelé schopni vykonat jednotlivé úkoly rychleji, než doposud, protože jinak hrozí riziko, že aplikaci používat nebudou a vrátí se zpět ke starému „papírovému“ systému. Snížit množství chyb, které uživatel při interakci se systémem udělá je možné pomocí několika těchto bodů [6]:

- Aplikace má být samopopisná – kromě toho, že musí vždy existovat kontextová nápověda, by Každé tlačítko či ikona měla být popsaná, případně při další interakci by měl systém dát uživateli zpětnou vazbu a tím pádem ho informuje o daném stavu

- Pokud se uživatel chystá provést nějakou akci, která je nevratná (např. odeslání objednávky na čistící prostředky) je nutné aby tuto akci potvrdil

- Obecně složitost celého sytému by měla být taková, jaký je předpokládaný stupeň znalostí jeho uživatelů. Z popisu uživatelů viz. kapitola „Uživatelé produktu“ vyplývá, že se jedná o lidi, z nichž má drtivá většina zkušenosti s ovládáním PC a mobilních platforem, proto je vhodné aby byla aplikace navržena ve stejném duchu (např. klávesa F1 vyvolá nápovědu, ikony „zpět“ a „vpřed“ budou vizuálně odpovídat operačnímu systému Windows atd..)

- Pokud uživatel provede akci, která není nevratná (například přidání položky do nákupního seznamu, přidělení úkolu někomu ze zaměstnanců) měl by mít vždy možnost vrátit své rozhodnutí.

- Pokud uživatel nějakou chybu udělá, měla by se objevit konstruktivní chybová hláška s návrhem řešení. Už při návrhu je nutné aby se aplikace dostala do nekonzistentního stavu v co nejméně případech (např. nepovolit záporná čísla při zadávání platby nebo zakázání tlačítek, která v daném okamžiku nemohou být použita). Je důležité aby chyby aplikace uživateli nezabránily v jejím dalším používání a měl by vždy být schopen ji vrátit do konzistentního stavu

Podle [7] existuje 8 pravidel, která by měla být dodržována při návrhu uživatelského rozhraní:

- Konzistence

- Respektování skupiny uživatelů

- Poskytování zpětné vazby

- Navigace uživatele

- Vyvarování se chyb

- Možnost vrácení akce

- Předvídatelnost rozhraní

- Nezatěžování uživatelovy krátkodobé paměti

1.12.1 Konzistence prostředí

Konzistencí systému napříč jeho částmi je myšleno, že stejné věci se v daném prostředí dělají stejně a ovládání aplikace by mělo uživatele vést k zavedení stereotypů. Pro uživatele systému může být matoucí například pokud v různých částech systému vyplňuje datum a v první části je ve formátu mm/dd/yyyy a v druhé dd/mm/yyyy. Konzistence systému úzce souvisí s předchozími třemi body. Uživatel si totiž nebude muset pamatovat různá chování stejně pojmenovaných tlačítek, vizuální strukturu jednotlivých modulů a jiné. Tím pádem je jeho práce rychlejší a zároveň méně náchylná k chybám (viz. příklad s formátem datumu).

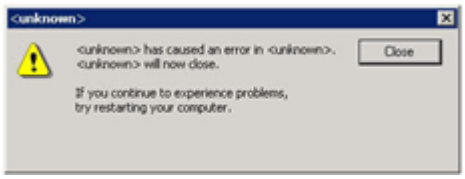
1.12.2 Respektování skupiny uživatelů

Osoba navrhující rozhraní musí vědět, pro jakou skupinu uživatelů daný program slouží a kteří ji budou používat (začátečníci, pokročili, starší lidé atd..) a podle toho dané rozhraní vytvořit. Pro starší uživatele může být například obtížné přečíst malý text a proto aplikace musí umožňovat přiblížení, aniž by to neblaze ovlivnilo jeho funkcionalitu. Také by v takové aplikaci nemělo být příliš mnoho cizích a odborných termínů. Naopak pokročilý uživatel ocení možnost využívání různých klávesových zkratek, úprava layoutu podle jeho přání a jiné.

Obrázek 1.11: Příklad vhodného uživatelského rozhraní mobilního telefonu pro starší osoby

1.12.3 Poskytování zpětné vazby

bylo lehce zmíněno v předchozí kapitole. Uživatel musí být informovaný o výsledcích akcí, které v systému provádí a to v obou případech – jestli akce byla úspěšná i pokud byla neúspěšná. V druhém případě je také nutné, aby byl uživatel informován o tom proč jeho akce úspěšná nebyla. Na obr. 1.12 je příklad chybové zprávy, která je očividně špatně, jelikož uživateli neříká co se



Obrázek 1.12: Příklad špatné chybové zprávy

stalo a ani co způsobilo chybu. Existují dva druhy zpětné vazby. Prvním typem je vazba volná, kdy je uživatel pouze informován o výsledku akce, druhým typem je vazba silná, kdy uživatel musí potvrdit, že se s danou informací seznámil.

1.12.4 Navigace uživatele

Řada akcí, které uživatel bude chtít provést se skládá z posloupnosti několika kroků. Je proto vhodné tyto kroky rozčlenit do několika samostatných logických celků, které budou pro uživatele příjemnější a pochopitelnější. Příkladem může registrace na webových stránkách firmy JumpShot a nákup jednoho z jejích produktů: V prvním kroku uživatel zadá přihlašovací údaje, email, a

Obrázek 1.13: Objednávka produktu na jumpshot.com 1. krok

vybere jaký typ produktu chce objednat. V dalším kroku uživatel zadá údaje ke své platební kartě. Na horní liště je názorně vidět, kolik kroků uživateli zbývá k dokončení objednávky a zároveň zde existuje možnost vrátit se na předchozí krok. Zároveň jsou všude označená pole, která jsou povinná a v případě, že uživatel nějaké z nich nevyplní nebo vyplní špatně, je na to aplikací

Obrázek 1.14: Objednávka produktu na jumpshot.com 2. krok

upozorněn. Tato obrazovky je tedy dobrým příkladem, jak by mělo UI vypadat. Uživatel přesně ví v jakém stavu se aplikace nachází jaké budou další kroky a má i možnost vrátit své rozhodnutí zpět. Zároveň splňuje i bod, že by měla být samopopisná, jelikož všechna textová jsou označená tak aby uživatel přesně věděl jakou informaci do nich má vyplnit.

1.12.5 Vyvarování se chyb

Vyvarování se chyb úzce souvisí s bodem poskytování zpětné vazby. Uživatelské rozhraní by se mělo snažit minimalizovat chyby, kterých se uživatel může dopustit. Toho se dá docílit jak vhodným popisem formulářů, tak zakázáním vkládání nepřípustných hodnot. V případě, že někde není možné takové chybě předejít a uživatel ji udělá, uživatelské rozhraní by mělo takovou chybu odhalit, nabídnout její řešení a případně kroky, které je potřeba k její nápravě udělat. Chybové hlášky by také měly být psané srozumitelnou řečí a rozhodně by neměly být technického typu, kterému běžný uživatel nemůže rozumět viz. obr. 1.15.

1.12.6 Možnost vrátit zpět provedenou akci

Vrácení akcí, které uživatel provedl, je důležitým faktorem návrhu dobrého rozhraní. Když uživatel ví, že své kroky může vrátit zpět a tím opravit například chybu, kterou provedl, podvědomě se přestane bát zkoušet různé možnosti aplikace, kterých by se jinak vyvaroval a zároveň má vetší komfort z toho

Obrázek 1.15: Chybová hláška technického typu

Obrázek 1.16: Dvě různá tlačítka pro načítání – v prvním případě se cítí uživatel jako ten, kdo podléhá aplikaci, v druhém jako ten, který nad ní má kontrolu

pohledu, že pokud například provádí složitější operaci, nemusí ji po jednom chybném kroku dělat celou od začátku.

1.12.7 Předvídatelnost rozhraní

Uživatelské rozhraní by mělo být navržené takový způsobem, aby při jeho ovládání měl uživatel pocit, že je to on, kdo je iniciátorem akcí a řídí aplikaci a ne že to je aplikace, které podléhá. Příkladem toho kdy uživatel cítí, že aplikaci ovládá a není on ovládán jí je na obr. 1.16. Malý rozdíl mezi dvěma tlačítky nahrávání, dokáže uživateli zpříjemnit práci se zařízením.Zobecněně se dá říci, že uživatel nemá rád překvapení. Dobrým příkladem porušování tohoto pravidla je navigace na některých webových stránkách, která by měla uživateli ukázat hierarchii stránek, ale namísto toho ho nečekaně přesměruje na jinou podstránku nebo na kompletně jinou webovou stránku.

1.12.8 Nezatěžování uživatelovy krátkodobé paměti

Uživatelova krátkodobá paměť by neměla být přetěžována nutností pamatovat si množství údajů, tento bod souvisí se známou poučkou „Rozpoznání namísto pamatování“, což znamená že je jednodušší určitou věc poznat a ne si pamatovat jak vypadá. Toto je důležité zejména při složitějších operacích,

kde údaje, které uživatel bude potřebovat k provedení dalšího kroku a které zadal v některém z předchozích, by měly být dobře viditelné, aby si je uživatel nemusel pamatovat nebo případně se vracet zpět zkontrolovat svoji volbu. Pro návrh uživatelského rozhraní platí tzv. pravidlo 7+-2, které říká, že člověk si snadno krátkodobě zapamatuje 5-9 údajů a proto by neměl být uživatelským rozhraním nucen pamatovat si více.

1.12.9 Uživatelské rozhraní mobilních aplikací

Jelikož systém navrhovaný v této práci bude mít dvě části – mobilní a desktopovou část, je vhodné se podívat na to, jakým způsobem by mělo vypadat rozhraní mobilní aplikace, protože i když oboje spadá do skupiny počítačových zařízení, mobilní a desktopové aplikace se významně liší (dotykové ovládání vs. Ovládání pomocí počítačové myši, velikost obrazovky..). Telefony jsou zároveň více osobní, vždy po ruce a mají nespočet funkcí, které normální desktopový počítač nemá (např. určování polohy, ovládání pomocí gest a jiné). Vzhledem k tomuto výčtu tedy není překvapivé, že i design uživatelského rozhraní se od toho desktopového bude v mnohém lišit [8].

1.12.9.1 Použitelnost

Uživateli by mělo být jasné jak rozhraní používat hned od první chvíle. S tím souvisí jeho jednoduchost. Jelikož chytré telefony mají oproti normální stolním počítačům výrazně menší obrazovku, je pravděpodobné, že potřeba aby množství informací, které mu rozhraní poskytuje je přesně to, které uživatel potřebuje k dosažení svého cíle.

1.12.9.2 Efektivnost

Množství kroků, které uživatel musí provést k dokončení úlohy by mělo být co nejmenší a klíčové úlohy by měly být co nejefektivnější. Dalším důležitým bodem z hlediska funkčnosti aplikace je její setrvání ve stavu, ve kterém ji uživatel naposledy zanechal. Tím je myšleno, když s ní například pracuje a v průběhu mu někdo zavolá, po ukončení hovoru a návratu zpět by měla zůstat v tom stavu, ve kterém byla před zahájením hovoru.

1.12.9.3 Zapamatovatelnost

Rozhraní by se pro uživatele mělo stávat tím více jednodušší na používání, čím častěji aplikaci používá.

1.12.9.4 Vyhnutí se chybám

V ideálním případě by aplikace neměla dovolit uživateli udělat chybu.

1.12.9.5 Jednoduchost

Úlohy, které uživatel vykonává nejčastěji, by měly být dobře viditelné a dostupné a úlohy, které nepotřebuje dělat každou chvíli by měly být dostupné, nicméně neměly by zbytečně zabírat místo na hlavní obrazovce. U mobilních aplikací je zejména důležité se vyhnout přidávání zbytečné funkcionality a snažit se udržet rozhraní co nejpřehlednější.

1.12.9.6 Očekávání uživatele

Pokud uživatel provede nějakou akci, mělo by se stát přesně to co očekává, že se stane. Tím se vyhne momentu překvapení.

1.12.9.7 Viditelnost/dosažitelnost

Viditelnost je pravděpodobné nejdůležitější z vlastností, kterou by mělo mít rozhraní mobilní aplikace. Znamená, že nejvíce důležité informace by měly být vždy nejlépe viditelné. Jedno z doporučení, které návrháři UI pro mobilní platformy dávají je snažit se co nejvíce vyhnout posuvným obrazovkám, jelikož uživatel nemusí odhalit, že se na obrazovce skrývá více, než je momentálně viditelné. V některých případech to samozřejmě není možné, např. při vybírání volajícího v seznamu kontaktů. Pro takové případy je dobře navržené UI uděláno například tak, že na straně obrazovky je vidět posuvník, případně při přejití na posuvnou obrazovku je spuštěna animace „obráceného posunu“, kde se uživateli zobrazí položky níže a jednoduchou animací se posune na začátek seznamu a tím dá uživateli najevo, že zde je skrytý obsah. Dosažitelností je v tomto kontextu myšleno rozložení jednotlivých tlačítek na obrazovce. Většina lidí používá mobilní zařízení jednou rukou a proto by ovládací tlačítka měla být umístěna ve spodní části obrazovky, která je lehce dosažitelná palcem ruky. Toto je velký rozdíl oproti klasickým desktopovým aplikacím, kde jsou ovládací tlačítka umístěna většinou v horní části obrazovky což dává smysl pouze za použití myši. Dalším příkladem může být, pokud očekáváme, že uživatel bude aplikaci používat často k psaní textů, měla by aplikace podporovat otočení na šířku, jelikož v takovém případě jsou jednotlivá tlačítka na klávesnici dále od sebe a lépe se na ní píše. Způsoby jak dosáhnout těchto cílů:

- Snížit funkcionalitu na nepotřebnější úkoly
- Odstranit vedlejší informace na obrazovkách
- Minimalizovat uživatelský vstup
- Prioritizovat informace na každé obrazovce
- Použít existující standardy návrhu mobilních aplikací

1.12.9.8 Gesta

Používání gest je jedním z fenoménů moderních telefonních přístrojů. Přestože mohou v mnoha případech ulehčit práci, je zapotřebí brát v potaz několik věcí:

- Gesta jsou pro uživatele neviditelná a proto je jejich odhalení uživatelem často problém. Jedním z dobrých příkladů jak uživatele upozornit na to, že nějaká gesta může používat je na promočních iPadech firmy Apple v ochodech. Jakmile se nějaká stránka načte, na všech částech které jsou posouvatelné se provede „obráceny scroll“ do jejich defaultní pozice. Tento posun uživateli okamžitě indikuje, že může použít tzv. „swipe“ gesto, bez toho aby bylo nějak označeno, které oblasti jsou posuvné.
- Multi-touch gesta – u takových gest je obecně problém, že uživatel musí k jejich provedení používat obě ruce, což se v řadě případů jeví jako nevhodné. Taková gesta jsou dobra například u her, kde se předpokládá, že uživatel sedí někde v klidu a má obě ruce volné. Jelikož navrhovanou aplikaci budou používat lidé při práci, dá se předpokládat, že taková gesta nebudou moci používat a proto je nevhodné aby na jejich provedení závisely její důležité funkce. Lze říci, že u navrhované mobilní aplikace gesta v podstatě nebudou potřeba (kromě například už zmíněného swipe), jelikož většina uživatelů je využívat nebude a i kdyby existovaly, tak pravděpodobně nebudou vědět o jejich existenci.

1.13 Řešení pro mobilní zařízení

Jedním z požadavků zákazníka je implementace systému na mobilní telefony. Jako řešení jsou možné dvě varianty:

- Implementace vlastní mobilní aplikace
- Adaptace webového rozhraní systému pro prohlížeče na mobilních telefonech

Z několika důvodů byla vybrána adaptace webového rozhraní na prohlížeč mobilních telefonů:

- Jednodušší implementace, kde se mění pouze vzhled jednotlivých stránek a není nutné vyvíjet novou aplikaci.
- Kratší doba vývoje.
- Není nutnost dalších zdrojů pro vývoj nové aplikace.

Kapitola 2

Testování

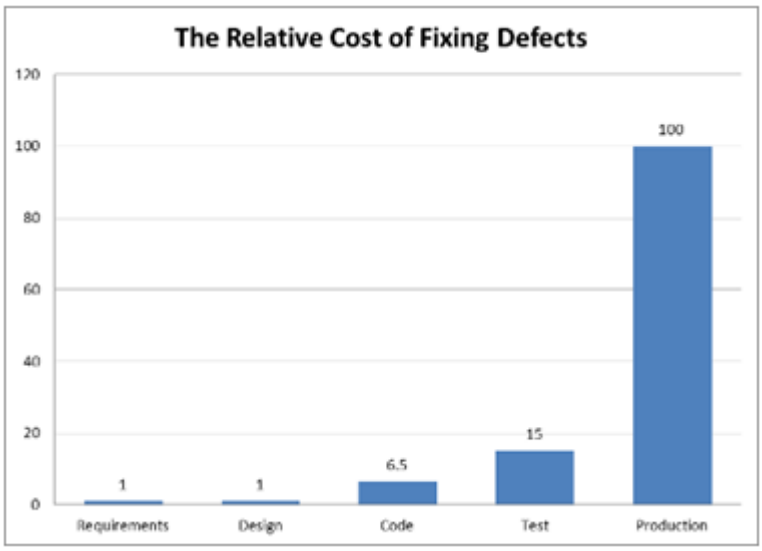
Testování je nedílná součást vývojového cyklu softwaru, která se prolíná všemi jeho částmi. Některé zdroje uvádí, že testování je kontrola toho, že software pracuje tak jak je napsáno ve specifikaci, což je velmi zjednodušený pohled na věc. Lepší definice testování: Testování softwaru je soubor procesů zaměřených na investigaci, ohodnocení a zjištění kompletnosti a kvality počítačového softwaru. Testování zajišťuje, že výsledný produkt je v souladu s regulatorními, byznysovými, technickými, funkčními a uživatelskými požadavky.

2.1 Testování v jednotlivých fázích vývojového cyklu

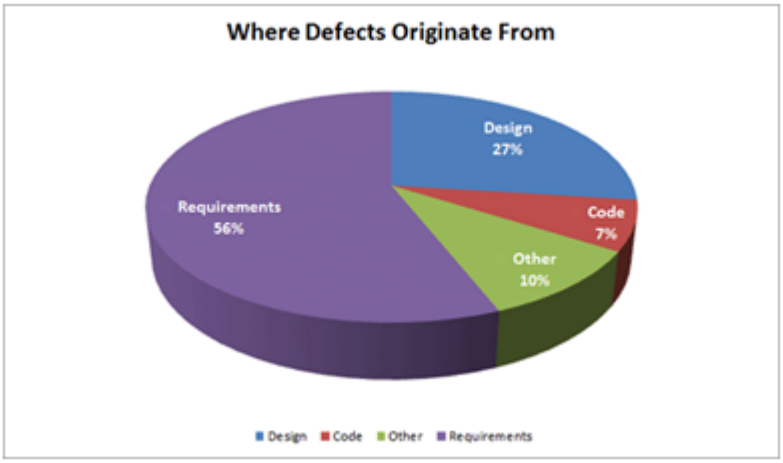
Jak již bylo řečeno výše, testování probíhá ve všech fázích vývojového cyklu. Je tomu tak z toho důvodu, že už v raných vývojových fázích vývojového cyklu, kdy neexistuje ani část reálné implementace, je množství věcí které se dají testovat. Na obr. 2.1 je uveden graf, který znázorňuje jakou cenu má oprava chyby v jednotlivých fázích vývoje.

2.1.1 Testování ve fázi analýzy

Na úvod této kapitoly je vhodné ukázat graf 2.2 , který podle analýzy Jamese Martina ukazuje, že více jak polovina všech chyb vzniká ze špatné, nedostatečné nebo nejednoznačné specifikace požadavků, což je výstupem analýzy. Z toho vyplývá, že testováním je nutné začít co nejdříve. Ve fázi analýzy by se mělo testovat, zda požadavky, které zákazník specifikuje jsou srozumitelně napsané, jednoznačné (toto je vhodné konzultovat s programátory, kteří budou software vyvíjet). A zároveň se zákazníkem neustále ověřovat, zda byly všechny jeho požadavky pochopeny správně. Pokud je totiž některý požadavek naspecifikován špatně, k odhalení, že nefunguje podle zákazníkových představ většinou dochází až ve fázi prvních tréninků nebo akceptačních testů. V této



Obrázek 2.1: Relativní cena opravy chyby v průběhu vývoje



Obrázek 2.2: Rozdělení chyb podle fáze, ve které vznikly, převzato z [1]

fázi zároveň z pohledu testů vznikají první dokumenty – testovací strategie a testovací plán.

2.1.2 Testovací strategie

Testovací strategie je dokument, který vzniká na začátku životního cyklu vývoje softwaru. Jeho vytvoření zajišťuje obvykle Test manažer, neboli osoba, která má na starosti řízení testů v daném projektu. Důvodu pro jeho vznik je několik. Prvním a nejdůležitějším důvodem je určit úsilí, potřebné k prove-

dení ověření kvality aplikace. Dalším důležitým důvodem je pomoci porozumět osobám mimo testovací tým manažerům, vývojářům, business analytikům..) detailům testovacího procesu a jeho nutnosti. Pokud je také testovací strategie jednou vytvořena, dá se s malým úsilím znovu použít na nových projektech. Jedná se high-level dokument, jehož účelem je popsat přístup k přípravě testů, jejich exekuci a organizaci v daném projektu. Jedná se o statický dokument, což znamená, že poté co je odsouhlasen všemi zainteresovanými stranami nebývá příliš často aktualizován. V úvodu tohoto dokumentu by měl být napsáno k čemu a kým bude dodávaný systém používán a jaké jsou jeho hlavní části. Dále by zde mělo být uvedeno na jaké HW a SW konfiguraci bude testování provozováno a také kdo kdy a jak bude testování provádět. Kromě úvodu by obsahovat tyto části:

- Test scope k jednotlivým fázím testů
- Matici zodpovědností za jednotlivé role v testovacím týmu
- Komunikační matici
- Lidské zdroje, požadované v průběhu testů
- Požadavky na ostatní zdroje
- Nástroje, pomocí kterých se bude testovat
- Způsob jakým bude stav testů reportován
- Rizika
- Vstupní a výstupní kritéria pro jednotlivé fáze testů
- Způsob řízení oprav defektů
- Plán tréninků

2.1.2.1 Test scope

Test scope je kapitola, která je připojena ke každé z fází testování (Assembly, SIT, UAT, Penetrační testy, Výkonnostní testy atd.) a popisuje, co a proč bude v každé z nich testováno a zároveň jaký bude výstup těchto testů.

2.1.2.2 Matice zodpovědností

Matice zodpovědností je vhodným nástrojem, který definuje, kdo je zodpovědný za jednotlivé procesy související s přípravou a exekucí testů.

Aktivita	Produktový manažer	Manažer vývoje	Test Manažer	Tester
Poskytnutí dokumentace k exekuci testů	X	X	X	
Plánování testů a jejich odhad			X	X
Revize projektové dokumentace	X	X	X	
Příprava testů a jejich exekuce				X
Řízení testovacích prostředí			X	X
Řízení oprav defektů a jejich opětovné předání testům		X		
Řízení změn v produktu	X	X	X	
Pravidelné reportování stavu testů			X	
Shrnující reportování testu			X	

2.1.2.3 Komunikační matice

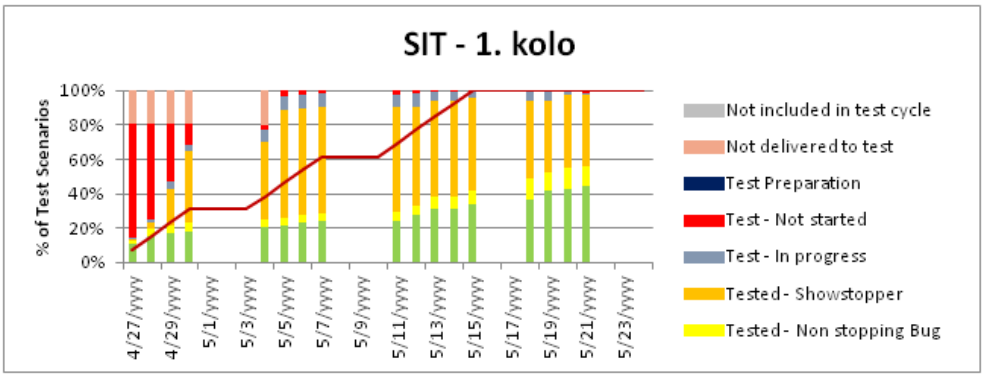
Při implementaci rozsáhlých projektů se často stává, že lidé znají pouze úzkou skupinu lidí, se kterými spolupracují na dodávce a neznají osoby mimo svůj tým. Proto je také kromě matice zodpovědností vhodné do testovací strategie zahrnout komunikační matici, která definuje osoby dosazené na jednotlivé funkce v projektu a jejich kontakt. Komunikační matice nemusí obsahovat všechny osoby zainteresované v projektu, nicméně postačí, když obsahuje hlavní projektové role.

2.1.2.4 Požadavky na lidské zdroje

Tato část testovací strategie je důležitá z toho důvodu, že už v počátečních fázích projektu ukáže, zda má testovací oddělení kapacity na to, aby provedlo testy v daném časovém období a případně umožňuje projekt přeplánovat, aniž by to mělo takové dopady, jako v pozdějších fázích nebo případně připravit se na období největší vytíženosti testerů např. dočasným najmutím externích kapacit nebo včasným náborem nových lidí.

2.1.2.5 Požadavky na ostatní zdroje

V této kapitole by měly být uvedeny všechny nároky na HW a SW vybavení, přístupy do jednotlivých systémů, požadavky na data a dokumentace.



Obrázek 2.3: Příklad jednoho z reportů, reflektující stav testů v 1. kole SIT (System integration test)

2.1.2.6 Testovací nástroje

Tato část říká, jaké části testů budou automatické a jaké části budou prováděny ručně. Příkladem automatických testů mohou být například Smoke testy, prováděné po každém nasazení.

2.1.2.7 Způsob reportování

Kapitola reportování popisuje, jakým způsobem bude reflektován stav testů, to znamená typy reportů, jejich zpracovatele a cílové skupiny, kterým se budou jednotlivé reporty posílat. Příklady reportů:

- Report reflektující počet úspěšně otestovaných scénářů v daném cyklu/fázi
- Report reflektující počet nalezených chyb podle priority a oblasti, ve které se vyskytují
- Report reflektující rychlost odbavování nalezených chyb (počet denně zavřených)

I když je reporting velmi často považován za zbytečnou práci (každý přeci ví jak na tom projekt je), jeho příprava má dobrá opodstatnění. Kromě skutečného stavu může například ukázat, že na některé z částí systému je diametrálně větší množství chyb než na ostatních a tím pádem je potřeba posílit množství lidí v jeho týmu. Reporting je zároveň dobrý nastroj na to, jak informovat o stavu projektu zákazníka, jelikož ten obvykle není na denní bázi v kontaktu se zainteresovanými lidmi.

2.1.2.8 Rizika

Jedním z nejdůležitějších faktorů řízení projektů obecně je řízení rizik, jelikož aplikace se stávají více složitými, postavenými na různých technologiích, integrovanými mezi sebou a na vývojové firmy je kladen stále větší důraz na včasné a bezchybné dodávky. Pro testování jsou typickým příkladem tato dvě rizika:

- Časové problémy – nedodržení daných termínů je nejčastějším rizikem vyskytujícím se ve vývoji systému. Jelikož hlavní část testování probíhá jako poslední fáze vývojového cyklu, je pro test toto riziko zároveň nekritičtější. Příčina vzniku tohoto rizika není většinou v samotném testování, ale v některé z předchozích fázi, kdy většinou není takový tlak na dodržování termínů, protože samotný release je daleko. Proto by v testovací strategii měl být vždy navrhnutý přístup, jak se k takovému riziku postavit. To znamená mít připravenou prioritizaci testů, případně alternativní pořadí. Dále paralelní provedení jednotlivých částí – příkladem může být částečně se překrývající exekuce SIT a UAT testů, což vyžaduje dobrou koordinaci v rámci týmů (např. odbavováni duplicitních chyb...). Další z možností, která stojí za zvážení, je testování i po dodání projektu.
- Organizační problémy – velmi často se stává, že do řad testerů jsou najímáni lidé, kteří nemají potřebné technické schopnosti, což vyústí v problémy s rychlostí testování nebo v případě složitějších systému dokonce nemožnost provést některé specializované testy. Dalším organizačním problémem, který úzce souvisí s časováním, je velká zátěž na testery v posledních fázích projektu a tím pádem vzniká přepracovanost a demotivanost testovacího týmu, což vede k horším výsledkům testování.

2.1.2.9 Vstupní a výstupní kritéria

Pro každou fázi testů je vhodné definovat vstupní a výstupní kritéria, za kterých lze tuto fázi započít/ukončit. Pro rané fáze testování budou vstupní kritéria například:

- připravenost testovacích dat
- dokončení testovacích scénářů
- připravenost testovacího prostředí

Pro vstupní fázi akceptačních testů zákazníkem to jsou například:

- Připravenost testovacích prostředí pro zákazníka (mohou se velmi lišit od interních prostředí)
- Všechny části systému na kritické testě jsou otestované a průchozí

- Je úspěšně otestováno 100% akceptačních scénářů
- Všechny nahlášené kritické defekty systému jsou opravené
- Všechny defekty s vysokou prioritou mají definovaný plán oprav
- V celém systému je nahlášeno maximálně 100 defektů

2.1.2.10 Testovací plán

Testovácí plán je dokument, který vychází z testovací strategie. Jedná se o stěžejní dokument pro celý proces testování. Obsahuje zejména:

- Druhy a kategorie testů, které budou prováděny
- Harmonogram plánovaných testu
- Rozsah testování
- Požadavky na testovací data
- Vlastnosti systému, které budou testovány
- Vlastnosti systému, které testovány nebudou

Testovací plán je na rozdíl od Testovací strategie živý dokument, který je průběžně aktualizován podle stavu celého projektu.

2.2 Testovací strategie pro implementaci projektu Úklidové a údržbářské služby v hotelu

2.2.1 Účel dokumentu

Účelem tohoto dokumentu je popsat přístup, exekuci a organizaci projektu Úklidové a údržbářské služby v hotelu (dále jen projektu) ve smyslu definice rozsahu testů, jeho fází, požadavků (na personální, datové, HW a SW zdroje), časování a řízení defektů. Dokument dále specifikuje organizaci SIT a UAT testů, jejich datové potřeby a infrastrukturní požadavky. Pro každou z těchto fází jsou definována vstupní a výstupní kritéria, požadavky na zdroje a dokumentaci.

2.2.2 Rozsah testů

Cílem testů je otestovat veškerou funkcionalitu implementovaného SW. Rozsah testů je definován jeho businessovými požadavky viz. kapitola 1.9. Na základě těchto požadavků je připravena dokumentace pro exekuci jednotlivých fází:

- Assembly testy
- Systémové a integrační testy
- Akceptační testy

Testy budou zahrnovat veškeré funkční oblasti:

- Řízení současného stavu provozu
- Řízení stavu hotelu
- Řízení zaměstnanců

Řešení bude testováno na operačním systému Windows 7 a webovém prohlížeči Google Chrome verze 5 a vyšší.

2.2.3 Assembly testy

Rozsah

Cílem assembly testů je ověřit:

- Všechny komponenty systému a operace mezi nimi (moduly, interface, atd.) fungují korektně
- Hlavní funkcionalita řešení (co je hlavní funkcionalita bude definováno v průběhu projektu) pracuje správně

Rozvrh

Bude provedeno jedno kolo Assembly testů.

Testovací data

Testovací data budou manuálně připravena v rámci testovacích scénářů.

Vstupní kritéria

- Vývoj základních byznysových požadavků je dokončen
- Testovací data jsou připravená a validní
- Je dostupné testovací prostředí A, na kterém budou testy probíhat

Výstupní kritéria / Akceptace testů

- Všechny testovací scénáře Assembly testů jsou otestovány
- Výsledky akceptačních testů jsou zdokumentovány
- Výsledky testů a případné zbylé defekty jsou akceptovány vedoucím SIT testů
- Jsou připravena prostředí pro začátek SIT testů

2.2.4 Systémové a integrační testy (SIT)

Rozsah

Cílem SIT testů je otestovat kompletní funkcionalitu projektu a připravit předání řešení do fáze UAT. Řešení bude testováno na infrastruktuře, která odpovídá zákazníkově. Aplikace třetích stran, které používá zákazník (rezervační a přístupový systém) nejsou součástí testování a jakmile budou propojeny s vyvíjeným systémem, jejich test bude proveden na straně zákazníka.

Rozvrh

Budou provedena tři kola SIT testů, v každém z nich bude otestována úplná sada testovacích scénářů.

Vstupní kritéria

- Vývoj požadavků je alespoň z 90 % hotov
- Výsledek Assembly testů byl akceptován vedoucím SIT testů
- Je dostupné testovací prostředí B, na kterém budou testy probíhat

Výstupní kritéria / Akceptace testů

- Všechny testovací scénáře SIT testů jsou otestovány
- Výsledky SIT testů jsou zdokumentovány
- Výsledky testů a případné zbylé defekty jsou akceptovány zákazníkem. Je vytvořen plán oprav zbylých defektů, které nebyly opraveny v rámci SIT testů a také je akceptován zákazníkem
- Systém je zintegrován s aplikacemi třetích stran
- Jsou připravena prostředí pro začátek UAT testů

2.2.5 Akceptační testy

Rozsah

Akceptační testy jsou finálními testy před nasazením systému do reálného provozu. Prováděny jsou na straně zákazníka, který by měl mít připravenou svoji sadu testovacích scénářů. Tyto scénáře jsou orientovány více byznysovým směrem, než je tomu v případě předchozích fází. Tím je myšleno to, že tyto scénáře představují reálné příklady toho, jak uživatelé budou s daným systémem pracovat. Z toho například také vyplývá jejich složitost a délka, která je obvykle větší, než je tomu u SIT testů.

Rozvrh

Bude provedeno jedno kolo akceptačních scénářů.

Vstupní kritéria

- Vývoj všech požadavků je hotov
- Zákazník akceptoval výsledek SIT testů
- Je dostupné testovací prostředí u zákazníka, na kterém budou testy probíhat

Výstupní kritéria / Akceptace testů

- Všechny testovací scénáře UAT testů jsou otestovány
- Výsledky UAT testů jsou zdokumentovány
- Výsledky testů a případné zbylé defekty jsou akceptovány zákazníkem. Je vytvořen plán oprav zbylých defektů, které nebyly opraveny v rámci UAT testů a také je akceptován zákazníkem
- Nasazení systému do ostrého provozu je schválené zákazníkem
- V rámci SLA je vytvořen a schválen dokument, v jakém intervalu budou opraveny defekty, které se objeví v reálném provozu

Kapitola 3

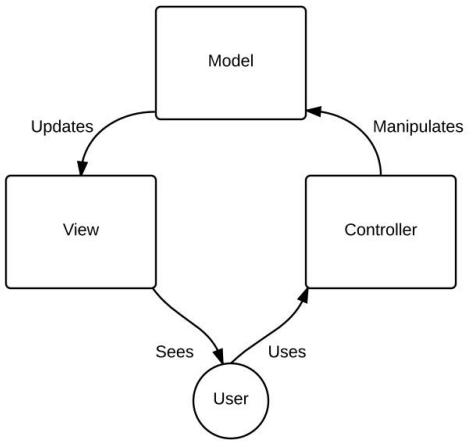
Implementace

Tato kapitola je věnována samotné implementaci (části) informačního systému, navrženého v předchozích kapitolách. Jako implementační jazyk je použit Groovy. Groovy byl vybrán z několika důvodů, hlavním je jeho alternativnost s programovacím jazykem Java a zároveň jeho schopnost skriptování, což z něj činí dobrý doplněk právě k Javě. Další obecnou výhodou tohoto jazyka je, že běží na Java Virtual Machine, z čehož plyne, že je ho možné použít kdekoli kde běží Java, podporuje použití veškerých knihoven napsaných v Javě a dále je rozšiřuje.

3.1 Grails

Tento informační systém je vyvíjen pomocí frameworku Grails. Grails je webový framework určený pro jazyk Groovy. Z velké části je postaven na na technologiích Spring, Hibernate atd. Mezi nejdůležitější charakteristiky tohoto frameworku, které ho činí tolik populárním patří tyto:

- Generátor kódu, který velmi urychluje vývojářovu práci
- Koncepční stavba aplikace na architektuře MVC
- Vysoká podpora unit a integračních testů
- Podpora Spring Security
- Velké množství pluginů
- Konfigurace pomocí anotací



Obrázek 3.1: Architektura MVC

3.2 MVC

Jak bylo zmíněno výše, Grails je na architektuře MVC založen, což znamená, že tuto architekturu využívá i implementovaná aplikace. MVC rozděluje aplikaci do třech vrstev:

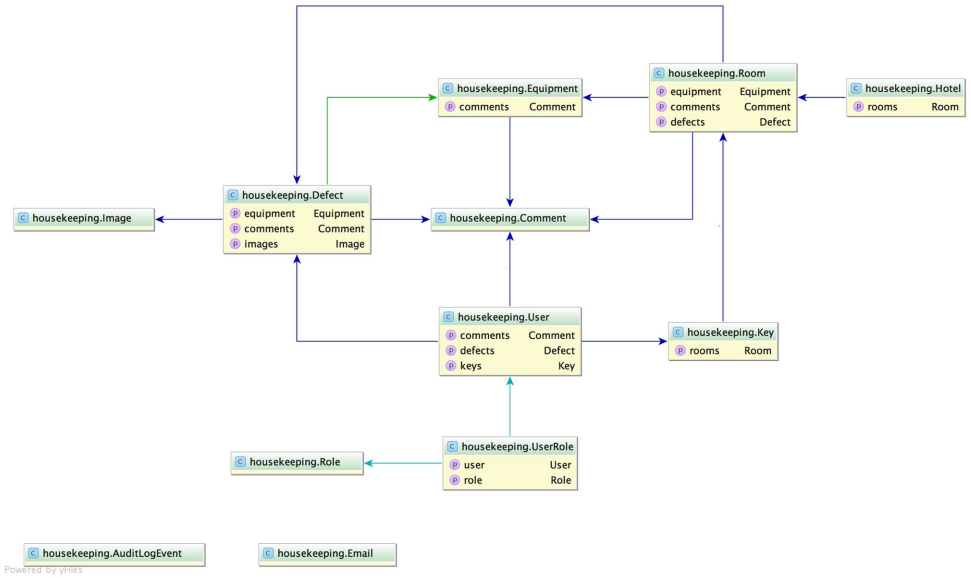
- Model - tato vrstva poskytuje data a metody pro práci s nimi a reaguje na požadavky změnou svého stavu. Neobsahuje informace o vizualizaci dat
- View - vrstva view je zodpovědná za zobrazení dat uživateli. Jako view je možné si přestavit například webovou stránku nebo okno nějakého programu
- Controller - controller zajišťuje spojení mezi uživatelem a systémem: kontroluje vstupní data a view používá pro jejich zobrazení

Výhodou této architektury je to, že view i controller jsou závislé na datové vrstvě a datová vrstva závisla na ničem není. To znamená, že pro jeden model je možné použít více view např. mobilní a webová aplikace. Architektura je znázorněna na obr. 3.1.

3.3 Popis implementace

3.3.1 Balíček domain

Tento balíček obsahuje třídy, které představují datový model aplikace. Mezi ně patří tyto třídy:



Obrázek 3.2: Vztahy doménových tříd

- *AuditLogEvent* - Třída pro ukládání změn na ostatních doménových třídách
- *Comment* - Komentáře k vybavení a pokojům
- *Defect* - Závady na vybavení a v pokojích
- *Email* - Poslané objednávky
- *Equipment* - Různé druhy vybavení hotelu a pokojů
- *Hotel* - Entita představující hotel
- *Image* - Nahrané obrázky různých závad
- *Key* - Univerzální klíče k pokojům
- *Role* - Uživatelské role (např. admin)
- *Room* - Pokoj v hotelu
- *User* - Uživatel systému
- *UserRole* - Mapování role na uživatele

Obr. 3.2 ukazuje, jaký je vztah mezi jednotlivými třídami datového modelu.

3.3.2 Balíček controllers

Pro vytvoření controlleru pro výše zmíněné doménové třídy je možné do příkazové řádky napsat příkaz *grails generate−controller* a název doménové třídy. Takto vytvořený kontroler podporuje základní CRUD (Create, Read, Update, Delete) operace. Vytvoření těchto operací zajišťuje nástroj *scaffolding*.

3.3.3 Balíček init

Balíček init obsahuje třídu *BootStrap*, ve které je definována startovní konfigurace aplikace. Tato konfigurace může například obsahovat testovací data, definování módu, ve kterém aplikace poběží (testovací, vývojový, produkční) a jiné.

```
def result = 'running in UNCLEAR mode.'
println "Application starting ... "
switch (Environment.current) {
case Environment.DEVELOPMENT:
addDevData()
addUserData()
break;
case Environment.TEST:
result = 'now running in TEST mode.'
break;
case Environment.PRODUCTION:
result = 'now running in PROD mode.'
break;
}
println "current environment: $Environment.current"
println "$result"
}
```

3.3.4 Balíček services

Pokud je aplikační logika umístěná uvnitř kontroleru, je porušené rozdělení v MVC. Z tohoto důvodu je v Grails balíček services, ve kterém se nachází celá aplikační logika. Kontrolery jsou zodpovědné za údržbu toku požadavků, přesměrování apod. Služba se v Grails vytvoří pomocí příkazu *create − service*. Příkladem využití služby v aplikaci je například služba *OrderingService*, která je popsána v kapitole 3.9.

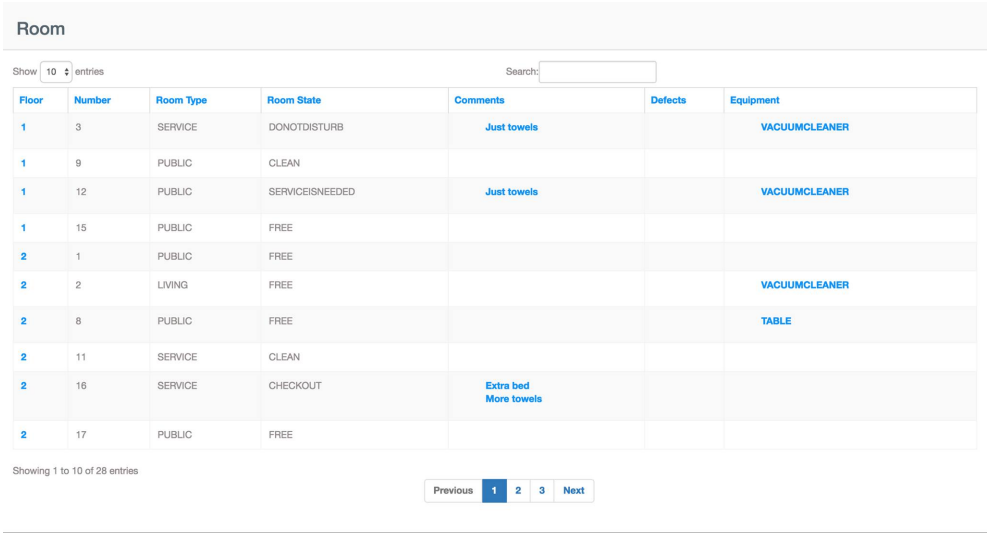
3.3.5 Složka views

Tato složka obsahuje soubory, které jsou potřebné pro vizualizaci celého systému. Grails umožňuje vytváření view souborů pomocí JSP (Java Server Pages)

nebo GSP (Groovy Server Pages). Každý kontroler má svůj balíček s view soubory. Kontrolery v Grails předávají data z modelu do view buď automaticky nebo pomocí metody *render*. Pro zobrazování seznamů objektů je používán tag $< f :tablecollection= "keyList"/ >$, který do tabulky zobrazí např. všechny atributy třídy *Key*. Aby daný tag fungoval, musí se napsat vlastní implementace tabulky. V aplikaci je použita následující implementace:

```
<table class="table table-striped table-bordered datatables-example"
width="100%" cellspacing="0">
<thead>
<tr>
<g:each in="${domainProperties}" var="p" status="i">
<g:set var="propTitle">${domainClass.propertyName}
.${p.name}.label</g:set>
<g:sortableColumn property="${p.name}"
title="${message(code: propTitle,
default: p.naturalName)}" />
</g:each>
</tr>
</thead>
<tbody>
<g:each in="${collection}" var="bean" status="i">
<tr class="${(i % 2) == 0 ? 'even' : 'odd'}">
<g:each in="${domainProperties}" var="p" status="j">
<g:if test="${j==0}">
<td><g:link method="GET" resource="${bean}">
<f:display bean="${bean}" property="${p.name}"
displayStyle="${displayStyle?:'table'}"/>
</g:link></td>
</g:if>
<g:else>
<td><f:display bean="${bean}" property="${p.name}"
displayStyle="${displayStyle?:'table'}"/>
</td>
</g:else>
</g:each>
</tr>
</g:each>
</tbody>
</table>
```

Tabulka vytvořená pomocí kódu výše je ukázáná na obr. 3.3. Další ukázkou tagů je $< f : displaybean = "key"/ >$, který vypíše atributy zadané domény. Poslední tag *uploadForm* slouží k nahrávání souborů. Použití tohoto tagu je ukázáno níže:



Obrázek 3.3: Tabulka pokojů v hotelu

```
<g:uploadForm action="upload">
<span class="input-group-btn">
<button class="btn fileupload-v1-btn" type="button"><i
class="fa fa-folder"></i> Choose File</button>
</span>
<input type="file" name="myFile" class="fileupload-v1-file hidden"/>
<input type="text" class="form-control fileupload-v1-path">
<span class="input-group-btn">
<input type="submit" class="submit btn btn-danger"/>
</span>
</g:uploadForm>
```

3.3.5.1 Šablony

Koncepce šablon rozděluje view na samostatné celky, kombinuje je s layouty a tím poskytuje znovupoužitelný mechanizmus pro složené view.

3.3.6 Ostatní

Dále je dobré zmínit složku *assets*, která obsahuje CSS a JavaScript soubory pro view. Další složkou je *conf* obsahující různé druhy nastavení (např. databáze, mailové služby a další).

3.4 Spring Security

Spring security je framework, který se jak je vidět již z jeho názvu specializuje na zabezpečení aplikace. V této práci je použit k dvěma hlavním problémům z pohledu zabezpečení - k autorizaci a autentizaci. Autentizací je myšleno přihlášení a odhlášení uživatele do systému a určení jeho identity. Autorizace je poté proces, který ověřuje, zda má aktuální uživatel právo provádět určitou akci (např. zobrazení specifického obsahu na stránce, mazání uživatelů atd.). Mezi další oblasti, které Spring Security podporuje, ale nejsou využity v této práci patří také zabezpečení proti některým druhům kybernetických útoků. Třída Role je jedna ze tříd, která je poskytována frameworkem Spring Security pro správu uživatelských rolí, přidělování práv a bezpečnost.

```
class Role implements Serializable {
private static final long serialVersionUID = 1
String authority
Role(String authority) {
this()
this.authority = authority
}
@Override
int hashCode() {
authority?.hashCode() ?: 0
}
@Override
boolean equals(other) {
is(other) || (other instanceof Role &&
other.authority == authority)
}
@Override
String toString() {
authority
}
static constraints = {
authority blank: false, unique: true
}
static mapping = {
cache true
}
}
```

Pro zajištění přístupových práv pro uživatele k různým funkcionalitám systému se používají anotace.

```
@Secured(['ROLE_USER'])
```

Další možností je statická mapa v souboru application.groovy, jejíž výhodou je to, že všechny přístupy a pravidla jsou vidět na jednom místě, což zpřehledňuje orientování se v nich.

```
grails.plugin.springsecurity.userLookup.userDomainClassName =
'diplomka.Auth.User'
grails.plugin.springsecurity.userLookup.authorityJoinClassName =
'diplomka.Auth.UserRole'
grails.plugin.springsecurity.authority.className =
'diplomka.Auth.Role'
grails.plugin.springsecurity.controllerAnnotations.staticRules = [
[pattern: '/', access: ['permitAll']],
[pattern: '/error', access: ['permitAll']],
[pattern: '/index', access: ['permitAll']],
[pattern: '/index.gsp', access: ['permitAll']],
[pattern: '/shutdown', access: ['permitAll']],
[pattern: '/assets/**', access: ['permitAll']],
[pattern: '/**/js/**', access: ['permitAll']],
[pattern: '/**/css/**', access: ['permitAll']],
[pattern: '/**/images/**', access: ['permitAll']],
[pattern: '/**/favicon.ico', access: ['permitAll']],
[pattern: '/user/**', access: 'ROLE_USER'],
[pattern: '/admin/**', access: ['ROLE_ADMIN',
'isFullyAuthenticated()']],
[pattern: '/thing/register', access: 'isAuthenticated()',
httpMethod: 'PUT']
]
```

V případě implementace autentifikace typu Basic, což je případ této práce, není implementována funkcionalita logout. Dále následuje ukázka vytváření základních rolí, uživatelů a jejich vzájemné mapování.

```
def user = new User(username: 'user', password: 'user', enabled: true,
accountExpired: false, accountLocked: false, credentialsExpired: false)
.save(failOnError: true)
def admin = new User(username: 'admin', password: 'admin',
enabled: true, accountExpired: false, accountLocked: false,
credentialsExpired: false)
.save(failOnError: true)
def userRole = Role.findByAuthority("ROLE_USER") ?:
new Role(authority: "ROLE_USER")
. save(failOnError: true)
def adminRole = Role.findByAuthority("ROLE_ADMIN") ?:
new Role(authority: "ROLE_ADMIN")
.save(failOnError: true)
```

```
UserRole.create(user, userRole, true)
UserRole.create(admin, userRole, true)
UserRole.create(admin, adminRole, true)
```

3.5 GORM

GORM je implementace ORM v Groovy, která používa framework Hibernate. Jedná se o framewrok, který poskytuje způsob jak konvergovat data mezi relační databází a objektově orientovaným jazykem. Jejím hlavním cílem je zajistit, aby entity v aplikaci měly odpovídající reprezentaci v databázovém sytému a aby zároveň byla zajištěna persistence těchto dat. GORM poskytuje dále pokročilé techniky jako je například:

- mapování dedičností
- mapování vztahů (1-n, 1-1, n-n..)
- základní CRUD databázové operace bez nutnosti psát selecty v SQL
- dynamické dotazy
- skládání primárních klíčů z několika sloupců
- kaskádové operace update a delete

Příklad využití asociace v GORM je zobrazen níže. HasOne a belongsTo jsou příkladem one to one asociace a hasMany je příklad many to many asociace. V daném příkladě je vidět, že objekt třídy Room může mít několik defektů, několik kusů nábytku a několik komentářů. Patří zároveň k jednomu patru a také k jednomu hotelu. Constraints v příkladě znamená omezení, které říká, že objekt typu Room musí mít přiřazenou právě jeden klíč a nemůže mít nepřiřazený žádný.

```
class Room {
static hasMany = [defects: Defect, equipment: Equipment,
comments:Comment]
static belongsTo = [hotel: Hotel]
static hasOne = [key: Key]
static constraints = {
key nullable: false
}
}
```

Jak bylo řečeno výše, GORM dále poskytuje metody pro CRUD operace, níže na příkladu je vidět použití metody Update. Tento konkrétní příklad přidá do místnosti PublicSpace nový nabytek - pohovku.

```
Room publicSpace = new Room(floor: floor, hotel: hotel,
key: key)
publicSpace.addToEquipment(new Equipment(equipmentType:
Equipment.EquipmentType.COUCH))
```

3.6 Gradle

Pro automatický build projektu byl použit jazyk Gradle. Jedná se o DSL (Domain Specific Language) jazyk založený na Groovy, který na rozdíl od hojně rozšířeného Apache Maven nepoužívá XML. Gradle je designován pro tzv. kontinuální integraci, což znamená, že podporuje inkrementální build aplikace, kde určí, které části aplikace jsou aktuální a proto není nutné dělat nový build[9]. Oproti klasickému XML formátu zdrojového kódu build skriptu je formát, který zavádí Gradle mnohem čitelnější a kratší. Některé výhody Gradle oproti Maven:

- Vyjmutí procesů při buildu - jakoukoli úlohu lze z procesu buildu vyloučit za pomocí -x příkazu
- Dry run - spuštění buildu bez toho aniž by se prováděly akce v jednotlivých procesech - tato vlastnost zajišťuje odhalení velkého množství chyb pouze v jednom spuštění buildu
- Využití Gradle GUI - kromě integrace s IDE a příkazové řádky Gradle nabízí také vlastní GUI

3.7 Audit logging plugin

Audit logging je plugin pro framework Grails, který jak již vyplývá z jeho názvu, slouží k zaznamenávání událostí na zvolených objektech. Události na které reaguje zahrnují uložení, změnu a smazání objektu. každá tato změna je poté uložena do databáze, čímž je vytvářen log událostí provedených na daném objektu. Instalace pluginu se do projektu provede přidáním kódu níže do souboru *build.gradle*.

```
compile "org.grails.plugins:audit-logging:2.0.0"
```

Následujícím příkazem je vytvořena třída, která reprezentuje změnu na daném objektu.

```
grails audit-quickstart org.myaudit.example AuditLogEvent
```

Níže následuje ukázka použití pluginu pro logování v implementaci této práce. Plugin je možné používat třemi způsoby. Příkazem *static auditable = true* umožníme audit doménové třídy pomocí zavedené třídy *AuditLogEvent*, která

bude ukládat změny do databáze při provádění operací vytváření, editování a mazání. Záznam v databázi vypadá poté takto:

```
{
"_id": NumberLong(11770),
"actor": "SYS",
"className": "Room",
"dateCreated": new Date(1462797051152),
"eventName": "INSERT",
"lastUpdated": new Date(1462797051152),
"persistedObjectId": "573082fb8df70c0d46bc92c0",
"persistedObjectVersion": NumberLong(0),
"propertyName": "comments"
}
```

Druhý způsob využití pluginu je kombinace auditu a odchycení událostí na objektu. Příklad využití je možné vidět na kódu níže, kde metody *onSave*, *onDelete* a *onChange* zachycují související události.

```
def onSave = {
log.info "onSave called: ${it}"
}
def onDelete = {
log.info "onDelete called ..."
}
def onChange = { oldMap,newMap ->
log.info "onChange called ..."
oldMap.each({ key, oldVal ->
if(oldVal != newMap[key]) {
println " * $key changed from $oldVal to " + newMap[key]
}
})
}
```

Třetí způsob použití pouze handlerů, které je zajištěno příkazem *static auditable* = [*handlersOnly* : *true*] Na následující ukázce je možné vidět, že v aplikaci je implementován první způsob použití pluginu.

```
class Room {
static auditable = true
ObjectId id
static mapWith = "mongo"
int number
int floor
RoomType roomType
```

```
RoomState roomState
static belongsTo = [hotel:Hotel]
static hasOne = [key:Key]
static hasMany = [equipment:Equipment, comments:Comment,
defects:Defect]
static constraints = {
floor min: 1
number min: 1
roomType nullable: false
roomState nullable: false
}
enum RoomType {
LIVING, PUBLIC, SERVICE
public static RoomType getRandom() {
return values()[(int) (Math.random() * values().length)];
}
}
enum RoomState {
FREE, CHECKOUT, SERVICEISNEEDED, CLEAN, DONOTDISTURB
public static RoomState getRandom() {
return values()[(int) (Math.random() * values().length)];
}
}
}
```

3.8 MongoDB

Pro implementaci byla použita MongoDB je databáze, která se řadí do skupiny NoSQL databází. Tato databáze ukládá data ve formátu BSON, což jsou dokumenty, které mají formát podobný JSON. Tyto dokumenty jsou velmi podobné klasickým objekům, tak jak jsou chápány v objektovém programování. Tato vlastnost MongoDB umožňuje jednoduché mapování databázových objektů na objekty v aplikaci a zároveň umožňuje snadné změny v databázi, aniž by bylo nutné měnit databázové schéma. Základním typem dotazu je dotaz klíč-hodnota, který vrací výsledek podle zadaného klíče. Stejně jako klasické relační databáze podporuje MongoDB dále dotazy zaměřené na porovnávání hodnot, součet, maximum a minimum a jiné. Indexování je v Mongo DB možné několika způsoby. Jedná se o použití unikátních nebo složených indexů, dále indexování pole a nebo Time To Live indexy. Jednou z dalších možností jak využít MongoDB je nahrazení souborového systému. Instalaci GORM pro MongoDB do projektu provedeme přidáním kódu *compile* "$org.grails.plugins : mongodb : 5.0.0.RC1$" do souboru *build.gradle*. Dále do souboru *application.yml* přidáme kód níže.

```
environments:
development:
grails:
mongodb:
connectionString: "mongodb://localhost:27017/project-db"
```

Do každé doménové třídy (třída, jejíž instance chceme ukládat do databáze) je nutné přidat kód *static mapWith* = "*mongo*". na ukázce kódu níže je možné vidět atribut *ObjectId*, který reprezentuje ID objektu v databázi.

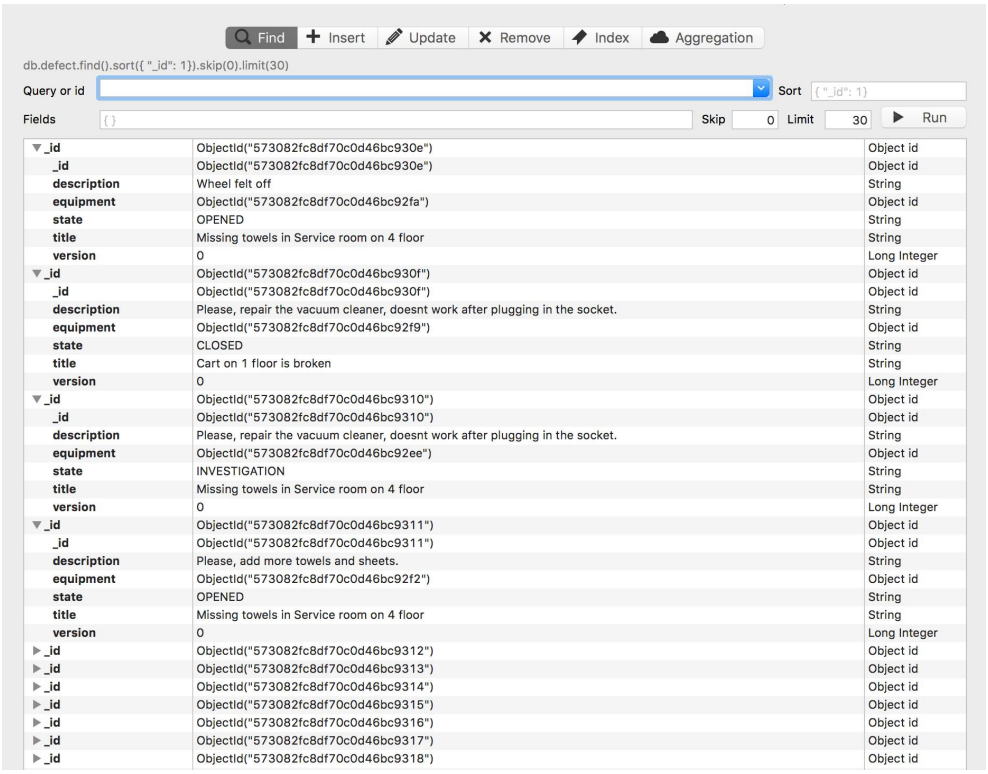
```
import org.bson.types.ObjectId
class Defect {
ObjectId id
static mapWith = "mongo"
String title
String description
static belongsTo = [room: Room, equipment: Equipment]
static hasMany = [comments: Comment, images: Image]
User user
State state
static constraints = {
title blank: false
description blank: false
room nullable: true
equipment nullable: true
user nullable: true
images nullable: true
}
enum State {
OPENED, CLOSED, INVESTIGATION
public static State getRandom() {
return values()[(int) (Math.random() * values().length)];
}
}
def String toString() {
return title;
}
}
```

Ukázku obsahu kolekce *Defect*, která reprezentuje závady na pokojích nebo na zařízení je možné vidět na obr. 3.4.

3.9 Grails mail plugin

Jedním z požadavků na aplikaci bylo zasílání mailů týkajících se například objednávek čistících prostředků nebo ložního prádla. Tato funkcionalita je v aplikaci realizována pomocí pluginu Grails mail. Instalace pluginu je provedena stejně jako tomu bylo u předchozího pluginu pro logování - do souboru *build.gradle* je přidán kód *compile* "*org.grails.plugins* : *mail* : 1.0.7". Grails mail plugin poskytuje funkci send mail, která je využita v kódu níže. Parametry *title*, *body* a *email* představují předmět mailu, text mailu a mailovou adresu příjemce. Kvůli možnosti znovupoužití byla tato funkcionalita realizována pomocí služby *OrderingService* v servisní vrstvě aplikace. Tato implementace zároveň dodržuje princip oddělení zodpovědnosti.

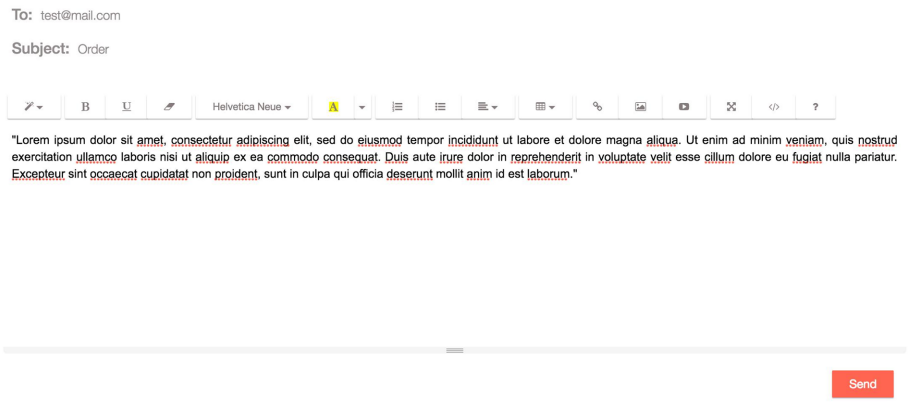
```
package housekeeping
import grails.transaction.Transactional
@Transactional
class OrderingService {
def mailService
```



Obrázek 3.4: Ukázka obsahu databáze

```
def sendOrder(address, title, text) {
Email email = new Email(title: title, body: text,
email: address)
assert email.save(failOnError: true, flush: true,
insert: true)
mailService.sendMail {
to email.email
subject email.title
body email.body
}
}
}
```

Implementovaná funkcionalita aplikace zasílání mailu je vidět na obrázku 3.5. Testování mailové funkcionality bylo provedené pomocí služby *mailtrap.io*.



Obrázek 3.5: Funkce zasílání mailu v aplikaci

Tato služba představuje falešný SMTP server, který umožňuje základní mailové operace. Pro přesměrování odchozí pošty na tento server je potřeba do souboru *application.yml* přidat následující kód:

```
grails:
mail:
host: mailtrap.io
port: 465
username: username
password: password
```

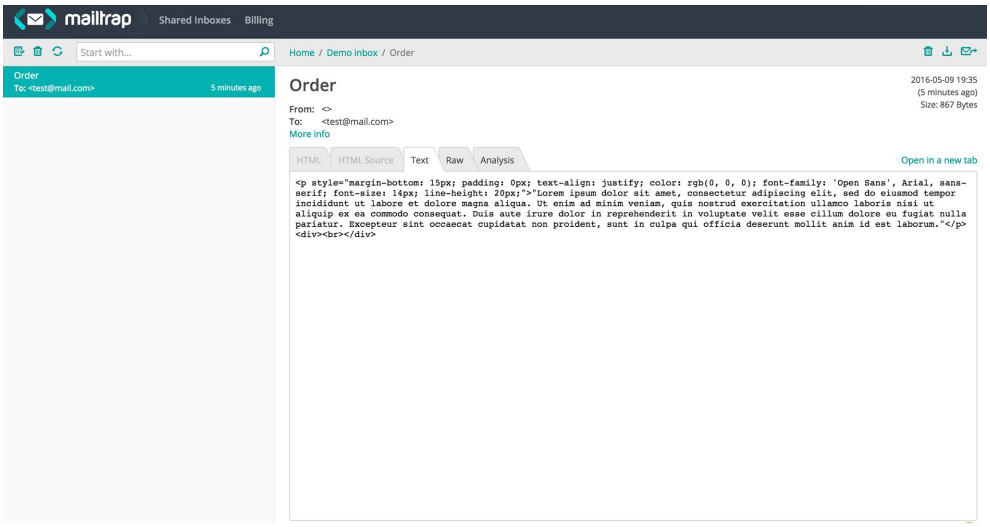
Na obr.3.6 je vidět zachycení testovacího mailu službou mailtrap.

3.10 Bootstrap CSS

Bootstrap CSS populární HTML, CSS a JavaScript framework pro vývoj webových aplikací. Instalace se provádí pomocí přidáním kódu níže do souboru *build.gradle*.

```
ext {
bootstrapFrameworkVersion = "1.0.3"
bootstrapFramework = [
useIndividualJs: true,
useLess : true,
fontAwesome : [
install: true,
useLess: true,
]
]
grailsVersion = project.grailsVersion
}
classpath "com.siprell.plugins:bootstrap-framework:
$bootstrapFrameworkVersion"
apply plugin: "com.siprell.plugins.bootstrap-framework"
```

Pro uživatelské rozhraní byla použita opensource šablona Miminium, která je založená na tomto frameworku.



Obrázek 3.6: Zachycení testovacího mailu

3.11 Frameworky/nástroje pro testování systému

K testování aplikace na všech jejich úrovních je použito několik frameworků a nástrojů, které kromě výrazného ulehčení práce umožňují i automatické testy, což je díky stále rostoucí poptávce po kvalitně a rychle dodaném produktu velmi žádoucí.

3.11.1 Spock

Spock je testovací framework na unit testy, rovněž založený na Groovy syntaxi. Přestože se jedná o technologii Groovy je pomocí něho možné testovat i klasické Java aplikace. Psaní testů ve Spocku zabírá méně času než psaní klasických jUnit testů spolu s použitím mock frameworku. Díky syntaxi Groovy je zároveň zlepšena čitelnost testů. Základní struktura je velmi jednoduchá a uživatelsky přívětivá. Není nutné dodržovat standardní jmenné konvence pro metody a místo toho je název testu uveden v uvozovkách. Tím je docíleno právě lepší čitelnosti testů, které dovoluje programátorovi přesně napsat jaký je účel tohoto testu. Spock je framework, který podporuje metodiku Vývoje řízeného pomocí požadavků na chování neboli Behaviour driven development (BDD), což je agilní metodika odvozená z metodiky řízení vývoje pomocí testů (test driven devlopment). Princip BDD spočívá ve vývoji aplikace pomocí popisu jejího chování na základě požadavků od jejich uživatelů. V souladu s principy této techniky, které nejsou v této práci dále rozebírány, je každý test ve Spocku rozdělen na tři části [10]:

Given sekce

Sekce *given* určuje kontext v rámci kterého bude daná funkcionalita testována. V této části jsou specifikovány parametry komponent, které ovlivňují funkcionalitu, která má být testována.

When sekce

Sekce *when* určuje co konkrétně má být vůbec testováno.

Then sekce

Sekce *then* je místo, ve kterém ověřujeme výsledek akce která byla vykonána v bloku *when*. Příklad použití frameworku Spock v této práci je uveden níže.

```
void "Test the index action returns the correct model"() {
when:"The index action is executed"
controller.index()
then:"The model is correct"
!model.hotelList
model.hotelCount == 0
```

```
}
void "Test the create action returns the correct model"() {
when:"The create action is executed"
controller.create()
then:"The model is correctly created"
model.hotel!= null
}
```

3.11.2 Geb

Geb je nástroj zaměřený na automatické testování webových aplikací. Konkrétně se zaměřuje na funkční testování, což z něj v kombinaci s frameworkem Spock tvoří silnou dvojici pro pokrytí automatických funkčních testů webového rozhraní.

3.11.3 Page object

Page object je návrhový vzor, který je využíván při testování webových aplikací. Page object reprezentuje webovou stránku a zároveň dovoluje ovládat prvky na stránce bez toho, aniž by se muselo používat HTML [11].

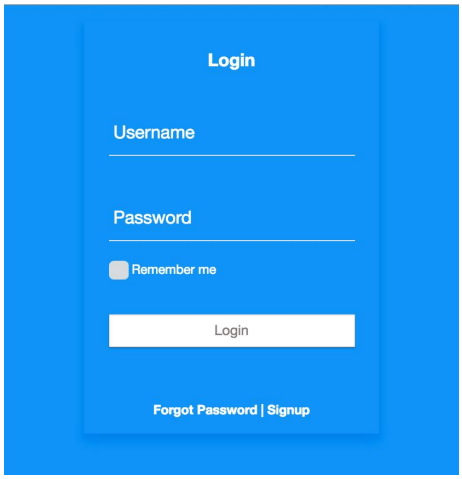
- Page Object reprezentuje obrazovku webové aplikace jako skupinu objektů a zároveň obaluje její funkce.
- Dovoluje modelovat UI v rámci testů.
- Zmenšuje duplikaci kódu.
- Zvyšuje udržovatelnost testů tzn. snižuje změny, které jsou v nich nutné udělat v případě změny v testované aplikaci.
- Zvyšuje čitelnost testů.

Ukázka použití Page obbject ve frameworku Geb:

```
package pages
import geb.Page
class LoginPage extends Page {
static url = "http://localhost:8080/login/auth"
static at = { heading.text() == "Login" }
static content = {
heading { $("h1") }
username { $("input[id=username]") }
password { $("input[id=password]") }
loginButton (to: DashboardPage){ $("input[id=submit]") }
}
}
```

Příkaz *static url* = "*http* : *//localhost* : 8080/*login/auth*" znamená, že url daného page objectu je http://localhost:8080/login/auth. Příkaz *static at* = {*heading.text*() == "*Login*"} slouží ke kontrole, že uživatel se nachází na dané stránce. Sekce *static content* obsahuje všechny elementy nacházející se na stránce *LoginPage*. Dále následuje ukázka testu funkcionality přihlášení se do aplikace s použití *LoginPage* objektu. Tento objekt popisuje stránku na obr. 3.7.

```
@Integration
@Rollback
class LoginTestSpec extends GebSpec {
void "login test"() {
when:"User is on login page"
to LoginPage
assert at(LoginPage)
and:"He enters valid username and password"
username.click()
username << 'me'
password.click()
password << 'password'
and:"He he clicks login button"
loginButton.click()
then:"He is redirected to Dashboard page"
assert at(DashboardPage)
}
}
```



Obrázek 3.7: Stránka popsána pomocí *LoginPage*

Závěr

Cílem této práce bylo provést zejména analýzu informačního systému, který bude sloužit pro údržbářské a úklidové služby hotelu Glacier Park. Jako první je vytvořena vize projektu, která identifikuje hlavní důvody, proč je potřeba v hotelu zavést nový informační systém a proč stávající způsob údržby je nedostačující. K tomu je použita technika SWOT analýzy tzn. určení slabých a silných stránek projektu. Dále jsou identifikovány požadavky na systém rozdělené do dvou kategorií - funkční a nefunkční. Nefunkční požadavky se zaměřují zejména na jednoduchost a spolehlivost celého systému, která je pro něj nejdůležitější. Funkční požadavky jsou dále rozděleny do čtyř podkategorií, kde každá reprezentuje jednu ze tří oblastí, které informační systém pokrývá. Jedná se o řízení stavu hotelu, řízení současného stavu provozu a řízení zaměstnanců. Práce dále popisuje předpokládaný profil typického uživatele, což je důležité zejména při návrhu aplikace a jejího uživatelského rozhraní. Následující část se zabývá analýzou trhu, kde je pro příklad porovnáno několik existujících řešení. Důležitou kapitolou je kapitola zabývající se riziky, jejichž správné řízení umožní dodat projekt ve předpokládaném čase, kvalitě a ceně. Je zde uvedeno několik rizik identifikovaných na základě předchozích zkušeností a předpokládaných problémů, které by mohly při implementaci nastat. Dále jsou představeny Use Case pro jednotlivé role a na závěr také nezbytný analytický model tříd. Následující část práce se zabývá metodikami, podle kterých je možné systém vyvíjet. Na základě jejich popisu je vybrána agilní metodika řízení vývoje SCRUM, která je dále popsána. Poslední kapitola analýzy se zabývá návrhem uživatelského prostředí a nejčastějšími problémy, které se při jeho používání vyskytují. Druhá část práce se zabývá návrhem toho, jak a kým bude v průběhu celého vývojového cyklu aplikace testována. Je zde popsáno několik nejdůležitějších dokumentů, které by při testování měly vzniknout. V poslední části této kapitoly je definována testovací strategie pro implementaci tohoto projektu. Kapitola implementace se zabývá samotným popisem implementace systému navrženého podle předchozích kapitol. V rámci této práce byla implementována základní funkcionalita systému, která zahrnuje scénáře

jako nahlášení závad, přidávání komentářů, rozdělování pracovních úseků zaměstnancům, odesílání objednávek a přířazování vybavení k pokojům. Systém dále umožňuje nahlížení, editaci a mazání všech entit pospaných v kapitole 3.3.1. Celá aplikace pracuje s databázi MongoDB, do které jsou také ukládány všechny změny provedené na doménových třídách. Dále jsou naimplementovány uživatelské role pro aplikaci a login. Velká část této kapitoly je věnována technologiím a frameworkům, které pro tuto práci byly použity. Zároveň jsou uvedeny jednotlivé části aplikace, které tyto technologie využívají. Kromě provedení akceptačních testů, které nebyly provedeny z toho důvodu, že samotný navržený systém nebyl implementován v celém rozsahu, bylo zadání práce splněno a tento bod je tedy možné navrhnout jako případné rozšíření či pokračování práce.

Literatura

[1] Martin, J.: http://mcnickle.org/are-you-testing-your-software-too-late-in-the-process/, verze z 13.12. 2015.

[2] Schwaber, K.; Sutherland, J.: *The Scrum Guide*. 2013, http://www.scrumguides.org/docs/scrumguide/v1/scrum-guide-us.pdf.

[3] Poppendieck, M.; Poppendieck, T.: Lean Software Development: An Agile Toolkit. 2003, http://www.infoq.com/resource/articles/poppendieck-implementing-lean/en/resources/poppendieck_ch02.pdf.

[4] Copeland, L.: Extreme Programming. http://www.computerworld.com/article/2585634/app-development/extreme-programming.html, verze z 13.12. 2015.

[5] Szalvay, V.: Glossary of Scrum Terms. https://www.scrumalliance.org/community/articles/2007/march/glossary-of-scrum-terms, verze z 13.12. 2015.

[6] J. Preece, Y. R.; Sharp, H.: Interaction Design: Beyond Human-Computer Interaction. 2002.

[7] Shneiderman, B.; Plaisant, C.: Designing the User Interface: Strategies for Effective Human-Computer Interaction (4th Edition). 2004.

[8] Nilsson, E. G.: Design guidelines for mobile applications. 2008, http://www.sintef.no/contentassets/311411d4f0734512bc828f6a7501b926/design-guidelines-for-web.pdf.

[9] Dockter, H.; Murdoch, A.: Gradle User Guide. https://docs.gradle.org/current/userguide/userguide.html, verze z 13.12. 2015.

[10] Niederwieser, P.: Spock Framework Reference Documentation. http://spockframework.github.io/spock/docs/1.0/index.html, verze z 13.12. 2015.

[11] Fowler, M.: Page Object. http://martinfowler.com/bliki/PageObject.html, verze z 13.12. 2015.

Příloha **A**

Seznam použitých zkratek

SWOT Strengths, Weaknesses, Opportunities, Threats

FURPS Functionality, Usability, Reliability, Performance, Supportability

DB Database

SW Software

HW Hardware

PC Personal Computer

XML Extensible Markup Language

IT Information technology

SIT System Integration Tests

UAT User Acceptance Tests

SLA Serice Level Agreement

ID Identifier

MD Man-day

RUP Rational Unified Process

UML Unified Modeling Language

QA Quality Assurance

UI User Interface

MVC Model View Controller

ORM Object-Relational Mapping

GORM Grails' Object-Relational Mapping

CRUD Create, Read, Update and Delete

SQL Structured Query Language

DSL Domain Specific Language

IDE Integrated Development Environment

URL Uniform Resource Locator

GUI Graphical User Interface

HTML HyperText Markup Language

CSS Cascading Style Sheets

SMTP Simple Mail Transfer Protocol

JSON JavaScript Object Notation

PSON Binary JSON

NoSQL Not Only SQL database

GSP Groovy Server Pages

JSP Java Server Pages

API Application Programming Interface

REST Representational State Transfer

HTTP Hypertext Transfer Protocol

SOAP Single Object Access Protocol

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF