ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Chodounská** Jméno: **Marie** Osobní číslo: **420618**

Fakulta/ústav: **Fakulta informačních technologií**

Zadávající katedra/ústav: **Katedra softwarového inženýrství**

Studijní program: **Informatika**

Studijní obor: **Webové a softwarové inženýrství**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Analýza využití návrhových vzorů a principů v aplikaci Home Assistant pomocí metod reverzního inženýrství

Název bakalářské práce anglicky:

Design Patterns and Principles Analysis for Home Assistant Application using Reverse Engineering Methods

Pokyny pro vypracování:

Návrhové vzory a principy v softwarovém inženýrství umožňují aplikovat obecná a ověřená řešení, čímž zkvalitňují výslednou implementaci a napomáhají předcházet překážkám v dalším vývoji. Open-source projekty často vznikají agilně a bez návrhu. Cílem práce je analyzovat v tomto ohledu vybranou open-source aplikaci pomocí reverzního inženýrství (RI), analýzu vyhodnotit a navrhnout zlepšení.
- Seznamte se s metodami RI v oblasti open-source software. Stručně popište různé RI přístupy i návrhové vzory a principy pro účely analýzy. - Proveďte základní rešerši Home Assistant (https://github.com/home-assistant/core) a zjistěte, které metody RI jsou vhodné pro tuto analýzu návrhu této aplikace.- Analyzujte aplikaci s využitím metod RI s důrazem na celkovou architekturu, návrhové vzory a principy DRY, KISS a základních principů dle teorie Normalizovaných Systémů.- Zhodnoťte výsledky analýzy a navrhněte možná zlepšení včetně odhadů pracnosti a přínosů do budoucna.

Seznam doporučené literatury:

Dodá vedoucí práce.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Marek Suchánek, katedra softwarového inženýrství FIT

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **21.10.2020** Termín odevzdání bakalářské práce: _____

Platnost zadání bakalářské práce: _____

Ing. Marek Suchánek podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	doc. RNDr. Ing. Marcel Jiřina, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Studentka bere na vědomí, že je povinna vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

. _____ _____

Datum převzetí zadání Podpis studentky

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Analýza využití návrhových vzorů a principů v aplikaci Home Assistant pomocí metod reverzního inženýrství

Marie Chodounská

Katedra softwarového inženýrství
Vedoucí práce: Ing. Marek Suchánek

27. června 2021

Poděkování

Ráda poděkovala vedoucímu mé bakalářské práce Ing. Marku Suchánkovi za trpělivost a cenné konzultace, které mi při psaní bakalářské práce poskytl. Dále pak příteli a rodině za soustavnou podporu a důvěru.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při 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. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, 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 27. června 2021 . . . . . .

Odkaz na tuto práci

Chodounská, Marie. *Analýza využití návrhových vzorů a principů v aplikaci Home Assistant pomocí metod reverzního inženýrství.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2021.

Abstrakt

Práce se zaměřuje na analýzu návrhu open-source aplikace Home Assistant. Home Assistant je systém pro správu chytrých domácností. Propojuje jednotlivé prvky chytré domácnosti a umožňuje jejich konfiguraci a správu. Aplikace je svým rozsahem netriviální a je stále aktivně vyvíjena. Mezi její hlavní výhody se řadí důraz na bezpečnost a soukromí. Systém řízení chytré domácnosti musí být stabilní a spolehlivý, na což má vhodný návrh přímý dopad. Návrhové vzory a principy jsou šablonami kvalitního návrhu, ověřené časem. Analýzou jejich využití v aplikaci, lze tedy získat představu o kvalitě jejího návrhu a určit, kde by návrh systému šel vylepšit.

K analýze jsou využity vybrané metody reverzního inženýrství. Reverzní inženýrství se zabývá rekonstrukcí nedostupných, neexistujících, nebo neúplných informací, z informací dostupných. Metody byly vybrány s ohledem na doménu Home Assistant a na skutečnost, že se jedná o open-source aplikaci. Analýza odhalila, že při návrhu aplikace byly návrhové vzory a principy brány ve velké míře v potaz. Její návrh lze tedy označit jako kvalitní. Existují však i místa, která principy porušují, nebo která by byla vhodná k dalšímu využití návrhových vzorů.

Klíčová slova Reverzní inženýrství, Rekonstrukce návrhu, Návrhové vzory, Návrhové principy, Analýza kódu, Home Assistant

Abstract

This thesis aims to analyze the design of the open-source application Home Assistant. Home Assistant is a system for smart home control. It connects elements of the smart home and allows their configuration and administration. Application has a non trivial scope and it is still being actively developed. Among its biggest advantages is focus on security and privacy. System which manages a smart home must be stable and reliable, which is directly impacted by proper design. Design principles and patterns are templates of good quality design proven by time. By analysing their use in an application, it is possible to get an idea about the quality of its design and to determine, where it would be possible to make improvements.

Reverse engineering methods are used for this analysis. Reverse engineering focuses on reconstructing unavailable, non existing, or incomplete information, from the information that is available. Used methods were chosen with regards to the domain of Home Assistant Assistant and the fact that it is an open-source project. Analysis has detected that design patterns and principles were considered during the design of Home Assistant. Therefore, it is possible to consider the design of this application to be of good quality. Regardless, there are sections of the application that break the principles and also sections that would benefit from further use of desing patterns.

Keywords Reverse Engineering, Design Recovery, Design Patterns, Desing Principles, Code Analysis, Home Assistant

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Tradiční vodopádový proces vývoje software se skládá ze specifikace domény problému, návrhu, vývoji, vytvoření dokumentace, testování a případná návazná oprava chyb, či specifikace nových požadavků.

I v komerčních projektech je však často z finančních a časových důvodů, fáze návrhu uspěchána, či zcela přeskočena. Především při agilním přístupu vývoje, kdy je kladen důraz na rychlou reakci na změnu v požadavcích. V open-source projektech je tento jev vzhledem k velkému množství programátorů ještě prominentnější. Přitom kvalitní návrh s sebou přináší mnoho výhod. Ať už se jedná o efektivní využití zdrojů, snadnější zorientování pro nově příchozí programátory, lepší rozšiřitelnost či stabilitu.

Právě tématem rozšiřitelnosti a stability se zabývá teorie Normalizovaných systémů fakulty aplikované ekonomiky univerzity v Antverpách, která shlukuje základní principy, jejichž dodržením lze snadné rozšiřitelnosti a stability dosáhnout. Principy obsažené v teorii Normalizovaných systémů byly definovány již dříve, na základě mnoholetých zkušeností a odborných znalostí, stejně jako další návrhové principy a vzory, které jsou v této práci popsány.

Práce je zaměřená na analýzu konkrétní open-source aplikace z pohledu využití návrhových vzorů a principů a doporučení na vylepšení jejich využití. Pro analýzu byla vybrána Python open-source aplikace Home Assistant. Jedná se o software pro vytvoření centrálního kontrolního systému chytré domácnosti.

První motivací pro výběr této aplikace byla aktuálnost téma chytrých domácností. Oblast chytrých domácností je v posledních letech velice probíraná a aktivně rozvíjena. Na trhu přibývá mnoho zařízení, které lze do chytré domácnosti integrovat. Aplikace Home Assistant zařízení vyhledá v síti a umožní jejich správu a vzájemné propojení z jednoho místa. Existuje i celá řada komerčních centrálních kontrolních systémů. Open-source projekty oproti nim mají však řadu výhod, mezi něž mimo jiné patří bezpečnost a ochrana proti sběru dat uživatele. Především však nabízejí možnost rychlého rozšíření

o podporu nových prvků.

Což vede k druhé motivaci výběru této aplikace. Komponenty systému jsou přímo napojeny na funkční prvky domácnosti a jejich výpadek má tedy přímý dopad na kvalitu života uživatele. Je tedy důležité, aby byl systém stabilní a spolehlivý. Z tohoto důvodu a vzhledem k rychlosti vývoje by aplikace z kvalitního návrhu a jeho dodržování mohla těžit.

K analýze využití návrhových vzorů jsou v této práci použity metody reverzního inženýrství. Reverzní inženýrství je proces opačný tradičnímu procesu vývoje nebo výroby. Metody reverzního inženýrství umožňují z koncového produktu a dostupných informací zrekonstruovat vyšší úrovně abstrakce.

V oblasti software se většinou využívá při testování bezpečnosti, etického hackování, zkoumání konkurenčních produktů, či při rekonstrukci návrhu zastaralých systémů, tzv. *legacy software*. Právě rekonstrukce *legacy software* a návrh open-source aplikace se často potýkají s nekompletní, či zcela chybějící dokumentací návrhu. Metody zabývající se analýzou *legacy software* lze tedy použít i pro analýzu open-source projektu.

Kapitola 1

Cíle a struktura práce

Práce má dle zadání poskytnout přehled návrhových vzorů a principů včetně základních principů teorie Normalizovaných systémů. Dále má popsat metody reverzního inženýrství, se zaměřením na takové metody, kterých lze využít při analýze návrhu open-source projektu.

Cílem práce je z popsaných metod reverzního inženýrství, vybrat takové metody, které jsou vhodné pro analýzu open-source aplikace Home Assistant. Následně tyto metody použít a identifikovat jednotlivé komponenty aplikace a jejich vzájemné vazby.

Výsledek analýzy poté vyhodnotit a doporučit vhodná místa k inovaci za účelem vylepšení návrhu dle popsaných návrhových vzorů a principů. Součástí vyhodnocení je odhad pracnosti navrhovaných změn a popsaní jejich přínosu.

Teoretická část této práce je uvedena stručným přehledem základních návrhových vzorů a principů. Následně jsou shrnuty hlavní principy teorie Normalizovaných systémů. Poté jsou rozebrány metody reverzního inženýrství, které lze k analýze open-source aplikace využít. V poslední sekci teoretické části jsou shrnuty dostupné informace o aplikaci Home Assistant.

V praktické části této práce je analyzován návrh aplikace Home Assistant za pomocí existující dokumentace a aplikovaných metod reverzního inženýrství. Jsou vyzdvihnuty místa současného stavu, které návrhové vzory a principy správně využívají. Dále je pak upozorněno na slabé stránky současného návrhu a možnosti vylepšení.

Kapitola 2

Návrhové vzory a principy

Problémy, se kterými se programátoři při vývoji aplikací potýkají se často opakují. Na základě zkušeností a odborných znalostí byly vytvořeny šablony elegantních a správných řešení opakujících se problémů. Těmto šablonám se říká návrhové vzory a principy. Rozdíl mezi návrhovými vzory a principy je v úrovni abstrakce. Návrhové principy popisují celkový tvar a strukturu softwarové aplikace a jsou tedy na vyšší úrovni abstrakce než návrhové vzory, které se soustředí na architekturu jednotlivých modulů a komunikaci mezi nimi. Návrhové vzory zachycují řešení, které byly vyvinuty v průběhu času vylepšováním existujícího kódu. [1] [2]

Jsou tedy řádně otestovány a ověřeny, z čehož vyplývá spolehlivost. Mezi další výhody jejich použití paří, že se problémům předchází již na úrovni architektonického návrhu. Při správném dodržení návrhových vzorů tedy určité problémy vůbec nenastanou. Vzhledem k tomu, že návrhové principy a vzory nejsou implementací závislou na programovacím jazyce, nýbrž šablonou, dají se znovu používat napříč projekty. [3]

2.1 Návrhové principy

Návrhové principy jsou standardy správného návrhu. Jedná se o základní myšlenky formulované odborníky na základě zkušeností. Řeší problémy nekvalitního návrhu, kterými jsou složitost provedení změn, nestabilita systému, nemožnost znovu použití již napsaného kódu a složitost dodržování zavedeného návrhu. Následující sekce shrnují základní návrhové principy. [1]

2.1.1 Principy SOLID

SOLID zkratka je složená z prvních písmen obsažených principů.

- **S**ingle-Responsibility Principle
- **O**pen-Closed Principle

- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

Jedná se o pět základních principů soustředících se na flexibilitu, údržbu systému a řešení závislostí. [1]

Princip jedné odpovědnosti (Single-Responsibility Principle) říká, že třída by měla mít pouze jeden důvod ke změně. Třídě by měla být udělena pouze jedna funkcionalita, pokud by třída měla mít funkcionality dvě, měla by se rozdělit do dvou tříd. [1] [3]

Princip otevřenosti a uzavřenosti (Open-Closed Principle) říká, že moduly by měli být otevřeny pro rozšíření, ale uzavřeny pro modifikaci. Jinak řečeno, pokud je potřeba funkci modulu rozšířit, nemělo by být nutné pozměnit existující logiku, ale měla by být možnost stávající logiku pouze rozšířit. [3]

Liskovové princip zaměnitelnosti (Liskov Substitution Principle) říká, že odvozené třídy by měli být zaměnitelné za jejich bázové třídy. Tedy uživatel bázové třídy by měl být schopen pracovat správně i pokud mu bude poslána odvozená třída místo bázové. [1]

Princip odděleného rozhraní (Interface Segregation Principle) říká, že mnoho specifických rozhraní je lepší než jedno univerzální rozhraní. Pokud je třída, která má několik klientů, je lepší vytvořit specifické rozhraní pro každého klienta, s metodami, které konkrétní klient využije, než načítat třídu se všemi metodami. [1]

Princip obrácení závislostí (Dependency Inversion Principle) říká, že závislosti by měli být na abstrakcích, nikoli na konkrétních implementacích. Každá závislost by v návrhu měla být nasměrována na rozhraní, nebo na abstraktní třídu. [1]

2.1.2 Principy GRASP

GRASP (General Responsibility Assignment Software Patterns) je devět principů, které se soustředí na objektový návrh a přiřazení odpovědnosti. [4]

Nízká/slabá provázanost (Low Coupling) říká, že je potřeba přiřadit zodpovědnost tak, aby provázanost zůstala nízká. Provázanost ukazuje, jak jsou dvě části kódu navzájem na sobě závislé. Vysoká provázanost znamená, že pokud potřebujeme pozměnit kód, budeme ho muset pozměnit na více místech. Snaha je udržet nízkou provázanost. Provázanost je ukazatel, který odkazuje nakolik objekt potřebuje znát informace o jiném objektu se kterým interaguje. Slabě provázaný návrh dovoluje stavět flexibilní objektově orientované systémy, které si dokážou poradit se změnami, protože je minimalizována závislost mezi objekty a tím i riziko, že provedená změna může mít dopad

na více dalších objektů. Nízká provázanost zjednodušuje testování, údržbu a úpravy systému. Systém může být snadno rozpadnut na menší celky. [3]

Vysoká soudržnost (High Cohesion) říká, jak spolu dvě části kódu souvisí. Nízká soudržnost znamená, že funkce obsahuje spoustu částí s různou funkcionalitou, které k sobě logicky nepatří. Vysoká soudržnost je situace, kde funkce má pouze jednu funkcionalitu. Snaha je soudržnost udržet vysokou, je poté snadné kódu rozumět, udržovat ho a znovu ho použít. [3]

Informační expert (Information Expert) říká, že je potřeba přiřadit zodpovědnost takové třídě, která má informace nutné ke splnění požadavku zodpovědnosti. Mezi výhody používání tohoto principu dle [4] patří následující.

- Informační zapouzdření, jelikož objekty používají svoje dostupné informace k plnění úkolů.
- Podpora nízké provázanosti, vzhledem k tomu, že objekty mají informace již k dispozici, což vede k lépe udržitelným a robustním systémům.
- Podpora vysoké soudržnosti, protože zodpovědnost je rozhozená mezi třídy s dostupnými informacemi, jednotlivé třídy jsou tedy jednodušší a tím snazší na údržbu.

Tvůrce (Creator) říká, že je vhodné třídě B přiřadit zodpovědnost vytvoření instance třídy A, pokud platí alespoň jeden z následujících bodů.

- B agreguje objekty A
- B obsahuje objekty A
- B zaznamenává instance objektu A
- B využívá objekty A
- B má data potřebné k inicializaci A
- B se nazývá tvůrcem objektů A.

Pokud bodům odpovídá více tříd B, je vhodné preferovat takovou třídu B, která agreguje nebo obsahuje třídu A Provázanost se použitím tohoto principu nezvýší, i když je třída A provázaná s B, protože vytvořená třída A již v původním návrhu provázaná s tvůrcem byla, jinak by dle definice neexistovala motivace vybrat třídu B jako tvůrce. [4]

Ovladač (Controller) říká, že je potřeba přiřadit zodpovědnost za přijímání, nebo vypořádání se se systémovými událostmi třídě, která reprezentuje systém, zařízení, nebo podsystém. Případně která reprezentuje use case scénář, v rámci něhož událost nastává. Výhodou tohoto návrhového vzoru je znovupoužití a snadné zapojení rozhraní. [4]

Nepřímé vazby (Indirection) říká, že je potřeba přiřadit zodpovědnost objektu, který dělá prostředníka mezi jinými komponenty nebo službami tak, aby nebyli přímo provázané. [4]

Polymorfismus (Polymorphism) o polymorfismu hovoříme, pokud objekt poskytuje různé implementace metod v závislosti na vstupních parametrech, nebo pokud jedno rozhraní může být použito objekty různých typů. [3]

Chráněné změny (Protected Variations) říká, je potřeba přiřadit stabilním rozhraním, které jsou postaveny nad nestabilními, nebo často se měnícími konkrétními třídami. [4]

Uměle vytvořená třída (Pure Fabrication) říká, je potřeba přiřadit vysoce soudržnou sadu zodpovědností uměle vytvořené třídě, která reprezentuje koncept domény problému tak, aby podporovala vysokou soudržnost, nízkou provázanost a snadné znovu použití. Třída je imaginární, existuje pouze z architektonických důvodů. [4]

2.1.3 Zákon Déméter (Law of Demeter-LoD)

LoD, nebo také princip minimální znalosti (The principle of least knowledge), vede k redukci interakce mezi objekty. Poukazuje na nutnost při návrhu myslet na počet tříd, které spolu interagují. Říká, že je potřeba se vyhnout vysoké provázanosti mezi třídami. Pokud je mezi třídami mnoho závislostí, systém se hůře udržuje a jakékoli změny mohou mít dopad na mnoho tříd. Podle principu minimální znalosti by každá třída měla mít omezené znalosti o ostatních třídách. Neměli by jí být známy detaily tříd se kterými potřebuje interagovat. [3]

2.1.4 KISS

Zkratka KISS lze aplikovat na více významů.

- "Keep it short and simple", což v překladu znamená: "zachovej kód krátký a jednoduchý".
- "Keep it simple and straightforward", což v překladu znamená: "zachovej kód jednoduchý a přímočarý".
- "Keep it Simple, Stupid", což v překladu znamená: "zachovej kód jednoduchý".

KISS je princip, který klade důraz na psaní jednoduchého a přehledného kódu uspořádaného do logických, ne příliš dlouhých celků. [5]

2.1.5 DRY, WET, AHA

DRY je zkratka, která znamená "Don't repeat yourself"a říká, že každá znalost nebo informace musí mít jedinou, jednoznačnou a směrodatnou reprezentaci v systému. [6]

DRY je princip, který klade důraz na vyvarování se duplikace kódu. Vyzdvihuje znovu použitelnost psaných metod. Porušení tohoto principu znamená použití opačného principu, principu **WET**. Zkratka WET lze stejně jako KISS aplikovat na více významů.

- "Write everything twice", což v překladu znamená: "vše piš dvakrát".
- "Write every time", což v překladu znamená: "piš vše pokaždé".
- "We enjoy typing", což v překladu znamená: "rádi rádi píšeme kód".
- "Waste everyone's time", což v překladu znamená: "plýtvej časem všech ostatních".

V článku [7] je princip definován jako "Zeptejte se sami sebe, jestli jste již tento kód nepsali na jiném místě. Dvakrát, ale nikdy třikrát". Oproti doslovnému překladu WET tato definice nezobrazuje WET jako nutně negativní princip, ale implikuje, že se vyplatí vytvářet abstrakce až při třetí duplikaci kódu.

WET v tomto významu souzní s principem **AHA**, který znamená "Avoid hasty abstractions". Princip AHA říká, že je duplikace kódu menším zlem, než špatná abstrakce, vzhledem k náročnosti vytváření abstrakcí. [8]

2.1.6 Princip Hollywoodu (The Hollywood Principle)

je princip, který říká "Nevolejte nám, my zavoláme vám". Jinými slovy, komponenty na vyšší úrovni rozhodují, kdy jsou komponenty na nižší úrovni systému potřebné. [3]

2.2 Návrhové vzory

Návrhové vzory se soustředí na strukturu aplikací.Zabývají se architekturou, která souvisí s účelem systému. Soustředí se na uskupení modelů a jejich vzájemných interakcí. [1]

V [2] se návrhové vzory dělí do tří kategorií:

- Tvůrčí nebo vytvářející (**Creational**),
- Strukturální (**Structural**) a
- Vzory zabývající se chováním (**Behaviour**).

2.2.1 Creational

Do kategorie creational spadají vzory, které se soustředí na způsob jakým jsou objekty vytvářeny. Mezi creational vzory patří následující návrhové vzory. [3]

Singleton(jedináček) je návrhový vzor, který zajišťuje jednu a pouze jednu instanci objektu konkrétního typu. Poskytuje globální přístupový bod

objektu. Pomocí jedináčka lze například kontrolovat přístup ke sdíleným prostředkům. Mezi hlavní nevýhody jedináčka patří fakt, že je globální. Může tedy být více referencí na jeden objekt, což může při nesprávné implementaci vyústit v problém, kdy se na jednom místě kód změní, aniž by si toho prostředky na jiném místě byly vědomy. [3]

Factory (továrna) je návrhový vzor, kde je dedikovaná třída zodpovědná za vytváření objektů jiných typů. Klient zavolá metodu s určitým typem a je mu vrácen objekt stejného typu továrnou. Přidání další třídy jiného typu do továrny k její výrobě je jednoduché, bez nutnosti upravování kódu klientem. Ten musí pouze poslat jiný parametr. Dle [3] jsou tři základní typy návrhového vzoru továrna:

- **Simple Factory (jednoduchá továrna)** umožňuje rozhraním vytvářet objekty bez nutné znalosti logiky vytváření objektů. Pomáhá vytvořit objekty různých typů spíše než přímou inicializaci objektu.

- **Factory Method (tovární metoda)** umožňuje rozhraním vytvářet objekty, ale odkloňuje rozhodnutí o vytvoření objektu do podtříd. Je definováno rozhraní pro vytváření objektů, ale místo aby továrna byla zodpovědná za tvoření objektů, zodpovědnost předá do podtřídy, která rozhodne, jaká třída bude inicializována. Objekty jsou vytvářeny skrze dědičnost, a ne skrze inicializaci. Vrací se ta samá instance nebo podtřída místo objektu určitého typu. Důsledkem této metody je lepší možnost uzpůsobení novým potřebám. Kód je generický, není vázán ke konkrétní třídě k inicializaci. Klient je vázaný na rozhraní, a ne na konkrétní třídu. Vzhledem k tomu, že kód který vytváří objekty je oddělený od kódu, který je používá, dochází k nízké provázanosti. Klient nemusí přemýšlet jaký argument má poslat a kterou třídu inicializovat. Přidání nových tříd je jednoduché a nejsou velké nároky na údržbu. Tato metoda odhaluje metodu klientovi k vytváření objektů. Tato metoda používá dědičnost a podtřídy k rozhodnutí který objekt vytvořit. [3]

- **Abstract Factory (abstraktní továrna)** je rozhraní k vytváření příbuzných objektů bez odhalení jejich tříd. Vzor poskytuje objekty další továrny, která vnitřně vytváří jiné objekty. Abstraktní továrna vytváří rodiny příbuzných produktů. [3]

Builder (Stavitel) je návrhový vzor, který odděluje konstrukci komplexních objektů od její reprezentace. Je tedy možné, aby ten samý konstrukční proces vytvořil různé reprezentace. Pomocí tohoto návrhového vzoru se dá vyvarovat konstruktorům s velkým množstvím parametrů, nebo velkému množství konstruktorů, které se od sebe příliš neliší. [2]

Dle [2] má stavitel čtyři části:

- *Stavitel*, který specifikuje abstraktní rozhraní pro vytváření částí *Produkt*.
- *Konkrétní stavitel*, který implementuje rozhraní *Stavitel*.
- *Režisér*, který používá rozhraní *Konkrétní stavitel*.
- *Produkt*, který reprezentuje komplexní objekt, který je konstruován.

Prototype (Prototyp) je návrhový vzor, který vytváří objekty klonováním instance prototypu. Dle [2] má prototyp tři části:

- *Prototyp*, který specifikuje abstraktní rozhraní pro klonování prototypu.
- *Konkrétní prototyp*, který implementuje rozhraní prototypu, klonuje sám sebe.
- *Klient*, který žádá prototyp aby se naklonoval, čímž vytváří nové objekty.

2.2.2 Structural

Do kategorie strukturálních vzorů paří takové vzory, které se soustředí na strukturu objektu a tříd a vztahy mezi nimi. Je kladen důraz na dědičnost a kompozici. Strukturální vzory popisují kombinování objektů a tříd do větších struktur, které zjednodušují realizaci a vztahy mezi nimi. [3]

Adapter (Adaptér) je návrhový vzor, který převádí rozhraní jedné třídy na rozhraní druhé třídy tak, aby mezi sebou tyto třídy byli kompatibilní a schopné komunikovat. [3]

Bridge (Most) je návrhový vzor, který odděluje rozhraní a jeho implementaci tak, že oba mohou fungovat nezávisle na sobě. [3]

Decorator (Dekorátor) je návrhový vzor, který dynamicky za běhu přidává rozhraním funkce objektu navíc. [3]

Facade (Fasáda) je návrhový vzor, který poskytuje jednoduché rozhraní komplexním subsystémům na pozadí, čímž se k nim dá přistupovat snadno a přehledně. Subsystémy neuzavírá, ale kombinuje je do subsystémů v pozadí. Důsledkem je snížení provázanosti implementace. Fasáda využívá principle of least knowledge. Dle [3] má fasáda tři části:

- *Fasáda*, která poskytuje rozhraní vnitřním subsystémům.
- *Systém*, který reprezentuje třídy implementující funkcionalitu systému, se kterými se složitě pracuje.
- *Klient*, který interaguje s fasádou, nemusí znát složitost systému.

Proxy je návrhový vzor, kde služba zaobaluje reálný objekt tak, aby bylo možné kontrolovat přístupy k objektu. Může rozšiřovat funkcionalitu objektu který obaluje. Poskytuje zjednodušené rozhraní tomuto objektu, zvyšuje bezpečnost a chová se jako štít proti útokům. Používá se jako vrstva, nebo rozhraní k podpoře distribuovaného přístupu. Mezi hlavní výhody patří zabezpečení reálných objektů a zlepšení výkonnosti aplikace. [3]

Composite (Složenina) je návrhový vzor, který skládá objekty do stromových struktur. Dle [2] má složenina tři části:

- *Komponenta*, která definuje abstraktní rozhraní pro objekty ve složenině.
- *List*, který reprezentuje objekt, jenž nemá žádné potomky.
- *Složenina*, která implementuje operace komponenty související s potomky.
- *Klient*, který přistupuje k objektům složeniny skrze rozhraní komponenty.

2.2.3 Behaviour

Do kategorie behaviour spadají vzory, které se soustředí na interakce mezi objekty a zodpovědnosti jednotlivých objektů. Také se soustředí nato aby spolu objekty mohli interagovat a přesto byly málo provázané.

Observer (Pozorovatel) je návrhový vzor, kde si objekt udržuje seznam na sobě závislých objektů a v případě změny je automaticky notifikuje. Dle [3] má pozorovatel tři části:

- *Subjekt*, což je třída která si je vědoma pozorovatele. Subjekt má metody jako registruj() a odregistruj() které používají pozorovatelé k registrování pozorování třídy.
- *Pozorovatel*, který definuje rozhraní pozorující subjekt. Definuje metody, které musí být implementovány pozorovatelem k tomu aby byl notifikován o změnách v subjektu.
- *Konkrétní pozorovatel*, který uchovává stav, jenž by měl být konzistentní se stavem subjektu. Implementuje rozhraní pozorovatele.

Konkrétní pozorovatel se registruje se subjektem implementací rozhraní poskytnutého pozorovatelem. Pokud je v subjektu změna, notifikuje všechny konkrétní pozorovatele pomocí notifikační metody pozorovatelů. Dle [3] jsou dva různé způsoby notifikací.

1. **Push model**, kde má subjekt dominantní roli. V push modelu posílá změny pozorovateli. Je však potřeba kontrolovat objem poslaných informací. Při poslání detailních informací mohou vyústit v pomalé odpovědi.

Vhodné je posílat pouze takové informace, které pozorovatel následně využije.

2. **Pull model**, kde má pozorovatel aktivní roli. V pull modelu subjekt broadcastem oznámí všem registrovaným pozorovatelům změnu a pozorovatelé si změny stáhnou. Na rozdíl od push modelu tedy tento model vyžaduje dva kroky a z toho důvodu je méně efektivní.

Subjekt a pozorovatel mají nízkou provázanost. Jediná věc, kterou *subjekt* o *pozorovateli* ví je, že implementuje určité rozhraní. Nezná *konkrétního pozorovatele*. *Subjekt* nemusí být modifikovaný pro přidání nového *pozorovatele*. Nejsou na sebe vázaní, každý z nich si může dělat své vlastní změny a nebudou se navzájem ovlivňovat. Mezi výhody pozorovatele patří nízká provázanost, efektivní posílání dat jiným objektům, bez nutné změny v subjektu, nebo pozorovateli a snadné přidání/odebrání pozorovatelům. Mezi nevýhody patří nemožnost kompozice, z důvodu nutnosti implementace rozhraní pozorovatele v konkrétním pozorovateli, který obsahuje dědičnost. Další nevýhodou je možná nespolehlivost notifikací, což může vést k nekonzistencím. [3]

Command (Příkaz) je návrhový vzor, kde je objekt použit k zapouzdření všech informací, které jsou potřebné k provedení akce nebo ke spuštění eventu. Mezi obsaženými informacemi je jméno metody a objektu, který metodu vlastní a hodnoty parametrů metody. Příkaz povoluje parametrizaci klientů s jinými požadavky a ukládání požadavků do fronty. Poskytuje objektově orientovaný callback. Příkaz také vytváří strukturu operací na vysoké úrovni abstrakce, které jsou postaveny nad menšími operacemi.

Dle [3] má příkaz pět částí:

- *Příkaz (Command)*, což je rozhraní pro spuštění operace, v objektu příkazu se ukládají parametry pro metodu příjemce.
- *Konkrétní příkaz*, který definuje propojení mezi objektem příjemce a akcí.
- *Klient*, který vytváří objekt konkrétního příkazu, klient vytvoří objekt příkazu a nastaví jeho příjemce.
- *Invoker (Vyvolávač)*, který poptává se konkrétního příkazu po provedení požadavku. Ví jak se spouští příkaz.
- *Receiver (Příjemce)*, který provádí operace spojené s vykonáváním požadavku.

Klient požádá *příkaz*, aby se vykonal. *Vyvolávač* vezme *příkaz*, zapouzdří ho a uloží do fronty. Třída *Konkrétní příkaz* zodpovídá za provedení příkazu a dotazuje se příjemce, aby provedl příslušnou akci. Mezi výhody tohoto návrhového vzoru patří snížení provázanosti tříd, které vyvolávají operaci

z objektu a které vědí jak spustit operaci. Poskytuje možnost vytvořit sekvenci příkazů, řazením do fronty. Přidávání nových příkazů je jednoduché a může být provedeno bez změny existujícího kódu. Také poskytuje možnost implementace rollback systému. Mezi nevýhody patří velké množství spolupracujících tříd, z čehož vyplývá obezřetnost při vývoji. Každý individuální příkaz je konkrétní příkaz třída, která zvyšuje počet tříd pro implementaci a údržbu.

Interpreter (Tlumočník) je návrhový vzor, který definuje reprezentaci gramatiky konkrétního jazyka. [2]

State (Stav) je návrhový vzor, kde objekt může zapouzdřovat vícero chování založené na vnitřním stavu objektu. Umožňuje měnit chování za běhu. Dle [3] má stav tři části.

- *Stav*, což je rozhraní, které zapouzdřuje chování objektu. Toto chování je spojováno se stavem objektu
- *Konkrétní Stav*, což je podtřída, která implementuje stav rozhraní. Konkrétní stav implementuje skutečné chování spojené s konkrétním stavem objektu
- *Kontext*, který definuje rozhraní klientům, udržuje instance podtřídy konkrétního stavu která vnitřně definuje implementaci objektu v určitém stavu

Strategy (Strategie) je návrhový vzor, který definuje skupinu algoritmů a zapouzdří je, díky čemuž je možné je mezi sebou vyměnit. Algoritmus se tím pádem může lišit nezávisle na klientech, kteří jej využívají. [2]

Chain of Responsibility (Zřetězení zodpovědnosti) je návrhový vzor, který předchází provázanosti příjemce a odesílatele požadavku tím, že určí více objektů, které mohou požadavek přijmout. Řetězí objekty příjemce a posílá požadavek do té doby, dokud jeden z příjemců požadavek nevykoná. [2]

Visitor (Návštěvník) je návrhový vzor, který reprezentuje operaci, která má být vykonána na struktuře objektů. Pomocí návštěvníka je možné definovat novou operaci, která zabrání změně třídy nad kterou je operace spuštěna. [2]

Iterator (Iterátor) je návrhový vzor, který poskytuje sekvenční přístup agregovanému objektu, aniž by bylo nutné znát jeho vnitřní reprezentaci. [2]

Mediator (Prostředník) je návrhový vzor, který zapouzdřuje způsob interakce mezi skupinou objektů. Není tedy nutné, aby na sebe objekty odkazovali přímo, čímž se snižuje provázanost. [2]

Memento je návrhový vzor, který zachycuje vnitřní stav objektu tak, že je možné objekt k tomuto stavu navrátit. [2]

Template (šablona) je návrhový vzor, který definuje kostru programu nebo algoritmu v metodě, které se říká šablona. Pomáhá změnit, nebo upravit určité kroky algoritmu tím, že odkáže implementaci některých těchto kroků do podtřídy. Podtřídy si mohou nadefinovat vlastní chování (overriding). Definuje

kostru algoritmu primitivními operacemi. Redefinuje určité operace podtříd bez změny struktury algoritmu. Tím zvyšuje znovu použitelnost kódu a vyhýbá se duplikaci úsilí. Využívá běžné rozhraní nebo implementace. Používá se, když více algoritmů nebo tříd implementuje podobnou nebo identickou logiku Dle [3] má šablona tři části:

- *Abstraktní třída*, která deklaruje rozhraní pro definici kroků algoritmu. Definuje operace nebo kroky algoritmu s pomocí abstraktních metod. Tyto kroky budou přepsány v konkrétní třídě.
- *Konkrétní třída*, která definuje podtřídu, a v ní specifické kroky algoritmu. Implementuje kroky specifické této třídě definované v abstraktních metodách.
- *Metoda šablony*, která definuje algoritmus voláním kroků metod. Definuje kostru algoritmu a sekvenci kroků algoritmu. Volá kroky definované v abstraktních metodách

Mezi výhody tohoto návrhového vzoru patří možnost znovu použít jeden kód, čímž přispívá k deduplikaci kódu. Při změně musí být přepsáno pouze několik metod. Také poskytuje velkou míru flexibility, kdy si podtřídy mohou upravit jednotlivé kroky algoritmu.

Mezi nevýhody patří složitost údržby. Pro nové programátory může být náročné porozumět sekvenci tříd. Je potřeba mít důkladnou dokumentaci a definovaný postup při opravě chyb. Hlavní nevýhodou je riziko rozbití implementace, při změnách na kterékoli úrovni. [3]

2.3 Normalizované systémy

Používaný informační systém je potřeba pravidelně aktualizovat. Ať už jde o změny *business procesů* v pozadí, reakci na nové technologie, či rozšíření funkcionality. Správně navržený a implementovaný systém je schopen na změny reagovat a zachovat si stabilitu a bezpečnost. Konstrukcí takového návrhu se zabývá teorie Normalizovaných systémů (NS). NS se zabývá zkoumáním chování modulárních struktur v reakci na změny. Vyslovuje čtyři základní principy a definuje, že pokud je nedodržen alespoň jeden z nich, změna generuje tzv. kombinatorický efekt. Kombinatorický efekt znamená, že provedená změna má dopad přímo úměrný velikosti systému. Pokud jsou všechny tyto principy dodrženy, velikost dopadů na systém je omezená. [9]

Nejprve je potřeba definovat základní pojmy. Jako zdroj pro teorii NS je [9]. Pro sjednocení české terminologie je použit první takový překlad, tj. překlad z [10].

- **Datová entita (Data entity)** je softwarová jednotka obsahující různé atributy nebo hodnoty, včetně odkazů na další datové entity. [10] [9]

- **Akční entita (Action entity)** je softwarová jednotka, která reprezentuje operaci. Akční entita obsahuje jeden nebo více úloh (tasks). [10] [9]
- **Úloha (task)** je součást akční entity a jedná se o množinu instrukcí, které provádějí určitou funkcionalitu. [10] [9]

Principy teorie NS jsou následující.

1. **Oddělení odpovědnosti (Separation Of Concerns)**

   Definice: "Akční entita může v normalizovaných systémech obsahovat pouze jednu úlohu (task)." [10] [9]

   Oddělení odpovědnosti je princip, kde je jeden blok kódu zodpovědný pouze za jeden aspekt funkcionality. Princip operuje na vyšší úrovni abstrakce než princip jedné odpovědnosti. Sdružuje třídy do modulů, které mají na starosti jednu funkcionalitu tak, aby se co nejméně překrývali. Cílem tohoto principu je snížit provázanost. Kód je vhodné rozdělit do více logických částí. Příkladem použití tohoto principu je více vrstvá architektura aplikace, TCP/IP protokol nebo ESB. [9]

2. **Oddělení/Udržování stavu (Separation Of States)**

   Definice: "Volání akční entity jinou akční entitou si musí v normalizovaných systémech udržovat stav." [10] [9]

   Oddělení stavu je princip, který je zaměřený na způsob volání mezi jednotlivými funkcemi. Pro dosažení stability je potřeba, aby volající funkce udržovala stav. To vede k odstranění provázanosti mezi moduly, čímž se zlepší stabilita systému. Příkladem použití tohoto principu jsou *workflow* systémy, asynchronní zpracování a komunikace mezi systémy. [9]

3. **Transparentnost akční verze (Action Version Transparency)**

   Definice: "Akční entita, která je volána jinou akční entitou, musí v normalizovaných systémech vystavovat transparentní verzi." [10] [9]

   Transparentnost akční verze je princip, který se zabývá transparentností akčních entit. Tedy tím, jak je funkce volána jinou funkcí. Mělo by být možné akční entitu snadno vylepšit. Příkladem použití tohoto principu je polymorfismus a zapouzdření. [9]

4. **Transparentnost datové verze (Data Version Transparency)**

   Definice: "Datová entita, která je přijímána jako vstup nebo produkována jako výstup některou akční entitou, musí mít v normalizovaných systémech transparentní verzi." [10] [9]

   Tento princip se zaměřuje na transparentnost datových entit. Zabývá se tím, jak jsou datové struktury posílány akčním entitám. Mělo by být

možné datovou entitu snadno vylepšit bez nutnosti úpravy akčních entit. Příkladem použití tohoto principu jsou webové služby a zapouzdření informací. [9]

Kapitola 3

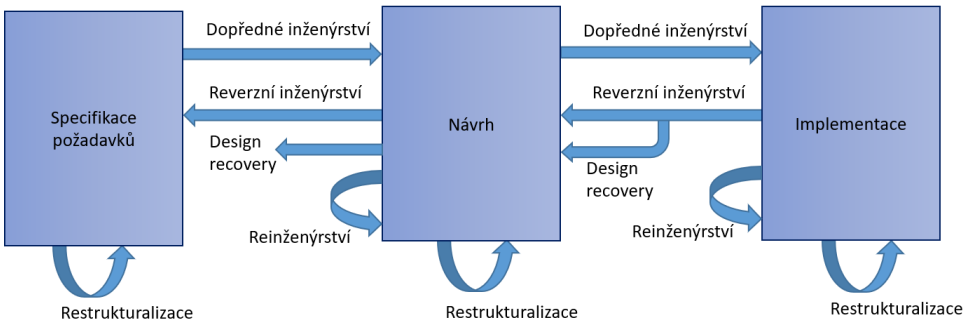
Metody reverzního inženýrství

Reverzní inženýrství je proces analýzy systému za účelem identifikování jeho komponent a vztahů mezi nimi. Přistupuje se k němu ve chvíli, kdy je potřeba doplnit chybějící znalosti o zkoumaném systému. Důvodem chybějících znalostí může nedostatečně zpracovaná, neudržovaná, nebo ztracená dokumentace. V jiných případech jsou znalosti neposkytnuty záměrně, například pokud je cílem aplikace reverzního inženýrství odhalit skryté informace o konkurenčním produktu.

Formální definice reverzního inženýrství byla vyslovena pro hardwarové systémy. V softwarových systémech byla terminologie sjednocena v [11].

Definice: "**Reverzní inženýrství** je proces vývoje skupiny specifikací pro komplexní hardwarový systém, skrze řádné zkoumání vzorku systému. Tento proces provádí někdo jiný než vývojář, bez možnosti nahlédnutí do originálních nákresů, s cílem replikovat původní hardwarový systém. Cílem reverzního inženýrství softwarového systému je zvýšit celkovou srozumitelnost pro údržbu a nový vývoj." [11]

Jednotlivé procesy spadající do oblasti reverzního inženýrství a vztahy mezi nimi jsou zobrazené na obrázku 3.1.



Obrázek 3.1: Reverzní inženýrství, vlastní zpracování dle [11]

Výstupem reverzního inženýrství je vytvoření reprezentace systému v jiné formě, nebo na vyšší úrovni abstrakce. Pokud je v rámci procesu zahrnuta i úprava systému do nové formy, nazývá se proces reinženýrstvím. Tento proces je také znám pod pojmem reklamace, nebo renovace. Restrukturalizace je transformace reprezentace systému v rámci jedné úrovně abstrakce do reprezentace jiné. Dopředným inženýrsvím (Foward Engineering) se v této oblasti rozumí standardní proces vývoje. [11]

Dle [11] má proces reverzního inženýrsví mnoho podprocesů, nejznámějšími však jsou redokumentace a *design recovery*.

Redokumentace je proces, jehož cílem je vytvoření nebo přezkoumání semanticky ekvivalentní reprezentace v rámci stejné úrovně abstrakce. Používá se k vizualizaci vztahů mezi jednotlivými komponenty systému. [11]

Design recovery je proces, který znovu vytváří ztracené návrhové abstrakce. *Design recovery* je kombinace analýzy kódu a existující dokumentace návrhu, osobních zkušeností a obecných znalostí problému a aplikační domény. Cílem je vytvoření struktury, které pomohou softwarovému inženýrovi porozumět programu nebo systému. *Design recovery* musí obnovit všechny informace, které inženýr potřebuje k porozumění toho co program, dělá, jak to dělá a proč to dělá. [12]

V *design recovery* metodách se mimo jiné využívá strukturální, dynamická a statická analýza kódu. [13]

3.1 Statická analýza

Statická analýza (Static Analysis) se zabývá vyhodnocením kódu bez jeho spuštění. Zkoumá tedy vlastnosti programu, které platí pro všechna spuštění. Spadá pod ní například *control flow* analýza a *data flow* analýza. [13] [14]

Control flow analýza se zabývá zkoumáním posloupnosti úloh programu. Jejím výstupem je *control flow* graf, kde uzly pro každou úlohu kódu a hrany určují posloupnost jejich spuštění. [15]

Data flow analýza se zabývá tokem dat v programu. Pro každý uzel control flow grafu studuje stav programu a jeho dat. Jejím výstupem je *data flow* graf, který zobrazuje závislost mezi daty a operacemi a určuje jejich dostupnost. [15]

3.1.1 Strukturální analýza

Strukturální analýza (Structural Analysis) se zabývá analyzováním struktury zdrojového kódu a jejích reakcí na externí změny. Vyhodnocuje kód bez jeho spuštění a je tedy typem statické analýzy. Spadá pod ní například analýza závislostí a dopadová analýza. [13]

Dopadová analýza (Impact Analysis) se zabývá určením potenciálních následků požadované změny systému. Kromě následků je dalším výstupem analýzy identifikace potřebných úprav systému, k docílení zadané změny.

Určuje tím rozsah implementace, díky čemuž se dá zhodnotit, zda je výhodné vývoj zadat. Dependency Impact Analysis je analýza, která určuje hloupku dopadu na systém [16]

Analýza závislostí se zabývá zkoumáním závislostí objektu. Díky tomu určuje místa, na které bude případná změna objektu mít dopad.

Dle [17] jsou její kroky následující.

1. Identifikace artefaktů, na nichž bude analýza prováděna.

2. Trasování vztahů artefaktu a jejich cílů dopadovou analýzou.

3. Rekurze trasování vztahů, dokud nenalezne žádné další cíle.

3.2 Dynamická analýza

Dynamická analýza (Dynamic Analysis) se zabývá spouštěním a zkoumáním chování programu a jeho vlastností při spuštění. Zkoumá vlastnosti, které platí pro jedno, nebo více spuštění programu. Nemůže dokázat, že program splňuje konkrétní vlastnost, může však nalézt porušení těchto vlastností a poskytnout užitečné informace o chování programu a jeho slabých místech. Zkoumané vlastnosti se mohou týkat například bezpečnosti, výkonu a práce s pamětí. Mezi metody dynamické analýzy patří například program slicing, analýza frekvenčního spektra, nebo analýza konceptu pokrytí. [14] [13]

Program slicing je metoda dynamické analýzy, pro automatickou dekompozici programu analýzou *data flow* a *control flow*. *Slicing* redukuje kód s určitou funkcionalitou na minimální formu, která stále stejnou funkcionalitu. [18]

Jinak řečeno, *program slicing* je úloha, která vypočítává *program slices*. *Program slices* jsou části programu, které ovlivňují, nebo mohou ovlivňovat hodnoty počítané v bodu zájmu. Tomuto bodu se říká slicing kritérium. Jde o dvojici (bod programu, skupina proměnných). Takové části programu, které mohou ovlivňovat hodnoty určěné *slicing kritériem* jsou *slicem* programu s respektováním kritéria. [19]

- **Static slicing** je takový *slicing*, kde k výpočtu *slice* stačí pouze staticky dostupné informace. [19] První přístup [18] slice vypočítává na základě *data flow* a *control flow* závislostí. Další přístup [20] představuje problém statického slicování jako problém dostupnosti v grafu závislostí programu (Program Dependency Graph). Jedná se o orientovaný graf reprezentující data a řídící závislosti mezi úlohami, kde hrany jsou *data flow* závislosti, nebo *control flow* závislosti a uzly jsou úlohy a predikátový výrok. [19]

- **Dynamic slicing** je takový *slicing*, kde je zájem zjistit, jak informace putují skrz program k získání konkrétní hodnoty. Prochází se při něm graf závislostí, které nastávají při specifickém běhu programu mezi úlohami programu. Dynamické *slicing* kritérium specifikuje vstup a rozlišuje mezi různými výskyty úloh v exekuční historii. Jedná se o trojici (vstup, výsyt úlohy, proměnná). Oproti statickému *slicing* tedy sleduje závislosti konkrétního vstupu. [19]

- **Thin slicing** je statický *slicing*, který řeší problém *slicing* ve velkých systémech. *Thin slicing* je technika porozumění kódu, která zahrnuje pouze takové úlohy, které jsou užitečné pro vývojový úkon. Tedy zahrnuje pouze takové úlohy, které pomáhají spočítat sledovanou hodnotu a ne takové hodnoty, vysvětlují důvod hodnoty (například podmínky). [21]

Analýza frekvenčního spektra je metoda dynamické analýzy, která pomáhá rozčlenit program, identifikovat výpočty, které souvisí se specifickým vstupem a výstupem programu. [14]

Analýza konceptu pokrytí je metoda dynamické analýzy, která počítá dynamické analogy na statické *control flow* vztahy. Porovnání těchto vztahů s jejími statickými protějšky může navést na oblasti kódu, které je potřeba více protestovat a jak spolu jednotlivé testy souvisí. [14]

3.3 Metody design recovery

Metody *design recovery* popisují způsob získání návrhových abstrakcí.

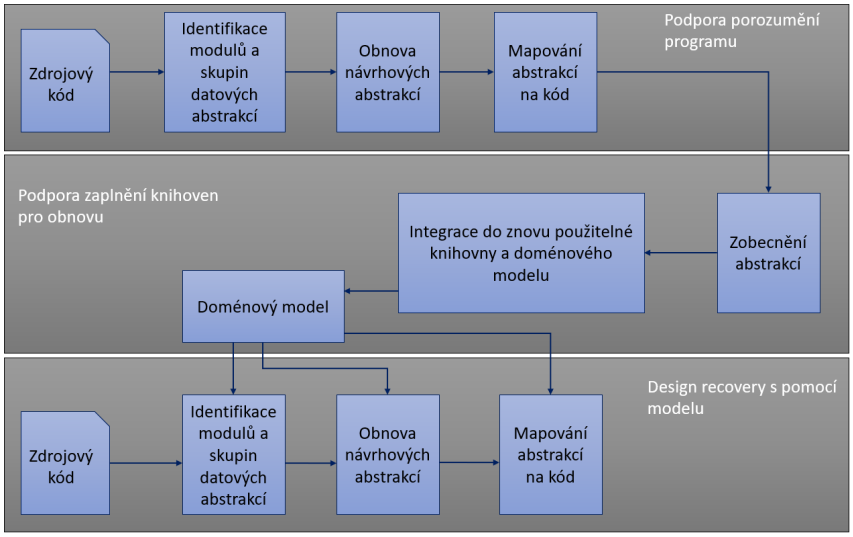
Přehled základních informací o vybraných metodách poskytuje tabulka 3.1. Základ tabulky byl převzat z [13]. Z původní tabulky byly vybrány pouze metody vhodné k analýze open-source aplikace Home Assistant. Tedy takové metody, které jako vstup berou zdrojový kód a jejichž doména je v souladu s doménou aplikace. Byl také posuzován výstup metody, který by měl napomáhat k analýze využití návrhových vzorů. Tabulka byla dále doplněna o další vhodné metody, které se v [13] nevyskytovaly. Metody s nejvyšší mírou relevance jsou dále popsány ve větším detailu.

3.3.1 Metoda Biggerstaff (1989): Vytvoření znovu použitelné knihovny

Logika metody Biggerstaff (1989) je postavená nad faktem, že nejjednodušší cestou *design recovery* je mít experta, který systém zná. Takový expert dokáže poskytnout informace, které pouze z kódu nelze vyčíst a které vedou k porozumění systému. Snaha je takového experta vytvořit v podobě automatizovaného systému. Systém musí mít stejnou bázi znalostí, jako by měl expert. Biggerstaff takovou bázi znalostí spatřuje v doménovém modelu. [12]

Publikace	Metoda	Doména	Vstup	Výstup
K. Keller et al. (1999) [22]	Strukturální analýza	Obecná	Zdrojový kód	Třídní hierarchie
Lothar (2005) [23]	Statická a dynamická analýza	Obecná	Zdrojový kód, strukturní vzory	Fuzzy hodnoty
Arun Lakhotia (1993) [24]	Hierarchical clusters	Obecná	Zdrojový kód	*Flow* orientovaný návrh
Nija Shi et al. (2005) [25]	Strukturální + statická analýza	Obecná	Zdrojový kód	Ověření, že vzor odpovídá sémantice
G. Antonoil et al. (1998) [26]	Strukturální analýza	Objektově orientované systémy	Zdrojový kód	Kandidáti na návrhové vzory
Harald et al. (1995) [27]	Znalost domény	Legacy systémy	Zdrojový kód	Objektově orientovaná architektura, kandidáti na návrhové vzory
Robitaille et al. (2002) [**?**]	Browsing, hledání	Obecná	Model zdrojového kódu	Informace o návrhu
Biggerstaff (1989) [12]	Strukturální analýza	Obecná	Zdrojový kód	Znovu použitelná knihovna
Kontogiannis et al. (1993) [29]	Strukturální + dynamická analýza	Legacy systémy	Zdrojový kód	Schémata programu, porovnávací techniky, rozhraní pro integraci nástrojů
Maurice et al. (2007) [30]	Datová analýza	Obecná	Zdrojový kód, objektově orientované míry	Klasifikované třídy se vztahy
Niere et al. (2001) [31]	Fuzzy logika	Obecná	Zdrojový kód	Návrhové vzory
Heuzeroth et al. (2003) [32]	Statická a dynamická analýza	Obecná	Zdrojový kód	Kandidáti na návrhové vzory

Tabulka 3.1: Přehled relevantních metod design recovery



Obrázek 3.2: Metoda Biggerstaff (1989), vlastní zpracování dle [12]

Vytvoření knihovny pro *design recovery* je dle [12] složený ze tří kroků:

1. **Podpora porozumění programu**. Pro splnění tohoto bodu je postup analytika následující.

   a) *Identifikace modulů a seskupení datových abstrakcí.* V tomto kroku analytik hledá organizační struktury vysoké úrovně.

   b) *Obnovení abstrakcí návrhu*, kterými jsou například neformální diagramy, neformální koncepty a vztahy, logický základ návrhu, moduly a *flow*.

   c) *Mapování abstrakcí na kód.* Pro splnění tohoto kroku si při plnění předchozího kroku analytik udržuje informace o vztazích mezi abstrakcemi a segmenty kódu, které implentují

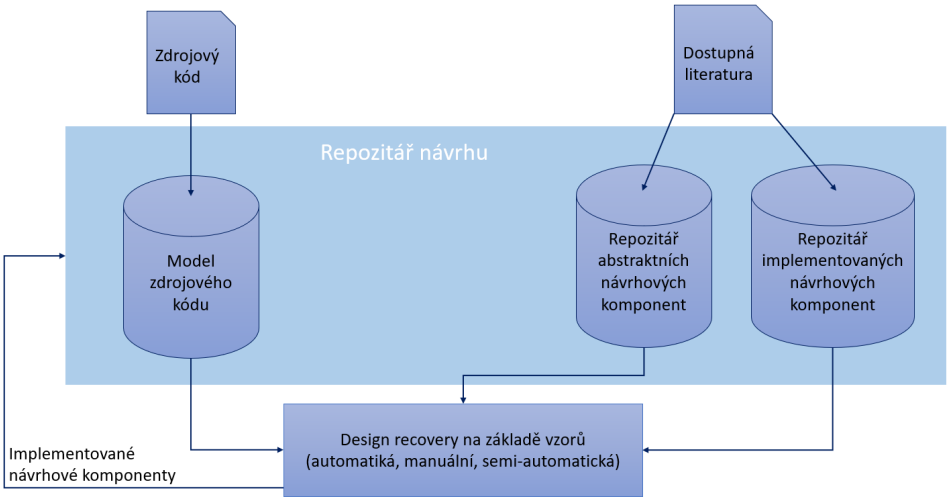
2. **Podpora zaplnění a znovu použití knihoven pro obnovu**. Z mapování abstrakcí na kód analytik abstrakce generalizuje a integruje je do znovu použitelné knihovny a doménového modelu

3. **Design recovery s pomocí modelu**. V tomto kroku je aplikován doménový model při dalším zkoumání zdrojového kódu.

Graficky je princip metody zobrazen na obrázku 3.2.

3.3.2 Metoda K. Keller et al. (1999): Reverzní inženýrsví návrhových komponent pomocí vzorů

Metoda K. Keller et al. (1999) byla představena jako logika za prostředím SPOOL. Myšlenkou této metody je využití strukturálních popisů návrhových vzorů, k nalezení jejich použití v systému, na které je reverzní inženýrství aplikováno.



Obrázek 3.3: Metoda K. Keller et al. (1999), vlastní zpracování dle [22]

Postup metody je následující. Nejprve je vytvořen repozitář, ve kterém je zdrojový kód reprezentován diagramy tříd. Poté je vytvořen repozitář šablon návrhových vzorů reprezentovanou diagramy tříd Posledním krokem této metody je snaha návrhové prvky zkoumaného systému uskupit do struktur, které by odpovídali šablonám návrhových vzorů z vytvořeného repozitáře. [22]

Dle [22] je poslední krok možné řešit třemi následujícími způsoby.

1. **Manuální design recovery**, kde jsou návrhové prvky seskupovány manuálně. Prostředí SPOOL umožňuje prvkům přiřadit role, které figurují v návrhových vzorech.

2. **Automatická design recovery**, kde jsou prvky seskupovány automaticky. Prostředí SPOOL obsahuje mechanismus dotazování, který dovede rozpoznat návrhové vzory dle jejich popisu.

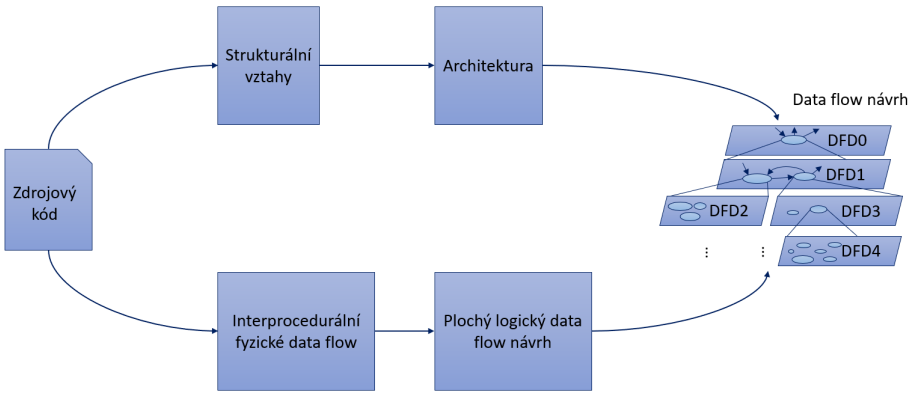
3. **Semi-automatická design recovery**, což je kombinace předchozích dvou způsobů.

Graficky je princip metody zobrazen na obrázku 3.3.

3.3.3 Metoda Arun Lakhotia (1993): Obnova *data flow* návrhu softwarového systému

Výsledkem metody Arun Lakhotia (1993) je *data flow* návrh obsahující hierarchii *data flow* diagramů. Jedná se o hierarchické uskupení funkcí a procedur tak, aby bylo možné identifikovat jednotlivé prvky (tzv. *bublin*) na různých úrovních hierarchie *data flow diagramu*. Na nejvyšší úrovni je jedna *bublina* reprezentující celý systém. Nižší úrovně hierarchie vznikají rozložením jedné *bubliny* výše na samostatný *data flow diagram*, dokud *bubliny* nelze již dále rozdělit. [24].

Problém obnovy *data flow* návrhu softwarového systému rozděluje na dva problémy. Vytvoření hierarchie *bublin* s funkcemi a procedurami a nalezení *flow* mezi *bublinami* na stejné úrovni tak, že výsledný *data flow* návrh je správnou abstrakcí skutečného systému. [24].



Obrázek 3.4: Metoda Arun Lakhotia (1993), vlastní zpracování dle [24]

Graficky je princip metody zobrazen na obrázku 3.4.

Kapitola 4

Home Assistant

4.1 Základní informace

Home Assistant je platforma pro kontrolu a správu domácí automatizace. Je centrálním bodem, který poskytuje uživatelsky přívětivé grafické prostředí pro ovládání a správu chytré domácnosti. Umožňuje vzdálenou konfiguraci koncových zařízení chytré domácnosti a zajišťuje jejich propojení. Pomocí Home Assistant je také možné definováním automatizačních pravidel stanovit vzájemnou závislost funkcí koncových zařízení a tím automatizovat procesy v chytré domácnosti. Dovoluje integrovat velké množství chytrých zařízení různých výrobců a jejich počet se stále zvyšuje. Systém běží v lokální síti a nevyužívá cloudových serverů pro zpracování dat, čímž zajišťuje vysokou míru bezpečnosti a soukromí.

Autorem Home Assistant je Paulus Schoutsen. Poprvé byla aplikace nasazena v roce 2013 a v současné době vycházejí nové verze měsíčně. Seznam dalších vývojářů a kontributorů, společně s jejich příspěvky do aplikace, je dostupný z [36].

Aplikace má uživatelskou dokumentaci, [34], která poskytuje návody k instalaci, konfiguraci a informace potřebné k používání Home Assistant. Kromě uživatelské dokumentace existuje také vývojářská dokumentace [33]. V té je popsána architektura aplikace společně s instrukcemi a návody týkající se vývoje.

Home Assistant je open-source aplikace vedená pod *Apache licencí 2.0*, což je licence s minimem restrikcí. Pro verzování je použit *Git*. Zdrojový kód aplikace je dostupný z [37]. Nad novým kódem je prováděna *Code Review*. Kód musí splnit všechny unit testy a projít skrz *linting* nástroje, které provádějí automatickou kontrolu zdrojového kódu. Pro Home Assistant je požadováno dodržení *PEP8 style* [39] a *PEP 257 Docstrings* [40] konvencí. K formátování kódu je použitý nástroj *Black* [38].

4.2 Architektura aplikace

Celá tato sekce je shrnutím informací poskytnutých v [33], týkajících se architektury. Home Assistant složený ze dvou částí:

- **Frontend** a
- **Backend**.

Ke komunikaci Home Assistant využívá *WebSocket API* a *REST API*.

4.2.1 Frontend

Frontend je postavený pomocí webových komponent. Je napsaný v *TypeScript* a používá web framework *Lit*. Pro testování je použit *Mocha*, což je testovací framework pro *JavaScript*.

Frontend Home Assistant má čtyři části:

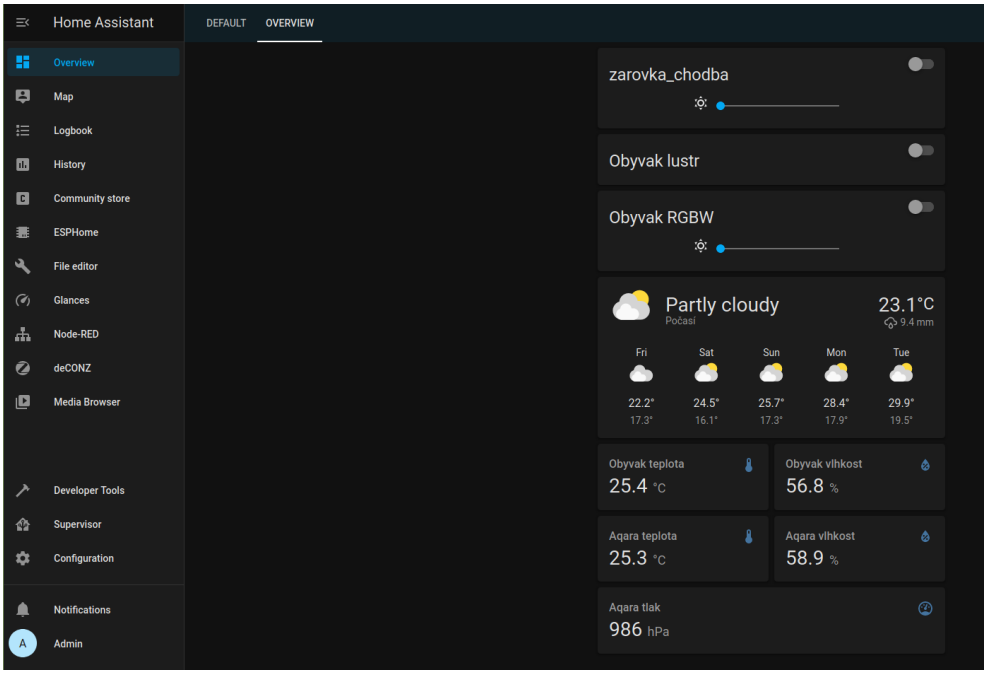
1. **Bootstrap** , což je skript, který se jako první nahraje na stránku. Je zodpovědný za kontrolu přihlašovacích údajů a nastavení *WebSocket* spojení s backendem.
2. **App shell** , což je skript, ve kterém je vše potřebné ke správnému načtení bočního panelu a směrování.
3. **Panels** , což je složka obsahující panely. Jeden panel je jedna stránka v Home Assistant.
4. **Dialogs** , což jsou informace a datové vstupy prezentovány uživateli.

Grafické uživatelské rozhraní Home Assistant se nazývá **Lovelace**. Je díky němu možné vytvářet nástěnky, které lze snadno pomocí karet uzpůsobit vlastním potřebám. Příklad nástěnky lze vidět na obrázku 4.1.

4.2.2 Backend

Backend je napsaný v *Python* a pro testování je použitý nástroj *Tox*. Pro správný běh aplikace jsou zapotřebí následující části, jejichž architektura je zobrazená na obrázku 4.2.

- **Operační systém**, který zajišťuje prostředí k běhu Core a Supervizora.
- **Home Assistant Core**, což je platforma pro automatizaci domácnosti. Interaguje s uživatelem, Supervizorem a IoT zařízení a služeb.
- **Supervizor**, který části výše spravuje a udržuje je aktualizované. S operačním systémem komunikuje pomocí D-BUS.



Obrázek 4.1: Home Assistant Frontend

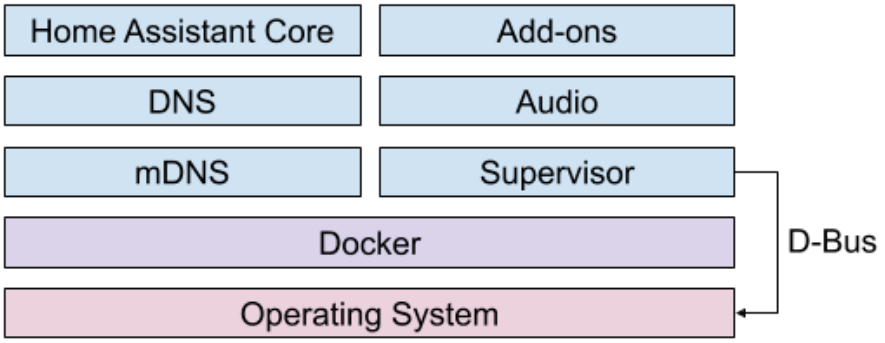
- **Addons**, což jsou aplikace, které běží na serveru. Umožňují rozšířit funkcionality Home Assistant.
- **DNS**, které umožňuje komunikaci Core a Addons.
- **Multicast DNS**, které objevuje a připojuje zařízení a služby v síti.
- **Audio**, které umožňuje Core a Addons využívat zvukových zařízení.
- **Operační Systém**, který zajišťuje prostředí k běhu Home Assistant.
- **Docker kontejner**, na kterém běží aplikace.

4.2.2.1 Operační systém

Pro Home Assistant jsou možné dva operační systémy (OS). **Home Assistant OS**, který se dříve nazýval **HassOS** a **Debian 10 (Bustler)**. Oba tyto OS jsou testovány a udržovány a optimalizovány pro zajištění bezproblémového běhu Home Assistant.

Běh obou těchto OS je podporovaný na následujících hardware zařízení.

- *Raspberry Pi*
- *Hardkernel ODROID*



Obrázek 4.2: Architektura Home Assistant Backend [33]

- *Intel NUC*
- *Asus Tinker Board*
- *Virtual appliances*

Home Assistant OS je operační systém optimalizovaný pro Home Assistant a jeho Addons. Nasazuje Supervizora jako docker kontejner. OS prioritizuje efektivní využívání paměti a úsporu místa. Home Assistant OS je složený z následujících částí.

- **Bootloader**, který se skládá z *Barebox* pro zařízení, která podporují *EFI* a *U-Boot* pro zařízení, která *EFI* nepodporují.
- **OS**, který je vytvořený pomocí *buildroot*, což je nástroj pro sestavení Linuxových systémů.
- **Souborové systémy**
  - *SquashFS*, pro read-only souborový systém pomocí *LZ4* komprese.
  - *ZRAM* pro /tmp, /var a swap za pomoci *LZ4* komprese.
- **Kontejnerová platforma**, což je *Docker Engine* vytvořený pro běh komponent Home Assistant v docker kontejnerech.
- **Aktualizace**, tedy *RAUC* pro *Over The Air (OTA)* a USB aktualizace.
- **Zabezpečení**, kterého je docíleno pomocí *AppArmor*.

4.2.2.2 Supervizor

Pomocí supervizora může uživatel spravovat instalaci Home Assistant, skrze Frontend. Mezi zodpovědnosti supervizora patří spuštění a aktualizace Core a aktualizace OS. Při neúspěšné aktualizaci spustí rollback a uvede Core i OS do původního stavu. Supervizor vytváří zálohy a v případě nutnosti provádí restore. Supervizor má následující části:

- **supervisord**, což je serverová část supervizora. Mezi jeho zodpovědnosti patří spouštění programů, odpovídání na příkazy klienta, restartování podprocesů, logování podprocesů a generování eventů odpovídajících činnosti podprocesů.
- **supervisorctl**, což je příkazová řádka pro klienta. Poskytuje rozhraní podobné shell pro funkce poskytované supervizord.
- **Web Server**, což je webové uživatelské rozhraní, které je alternativou k supervizorctl.
- **XML-RPC rozhraní**, což je webové uživatelské rozhraní, které může být použito k ovládání supervizora a programů, které na něm běží.

4.2.2.3 Core

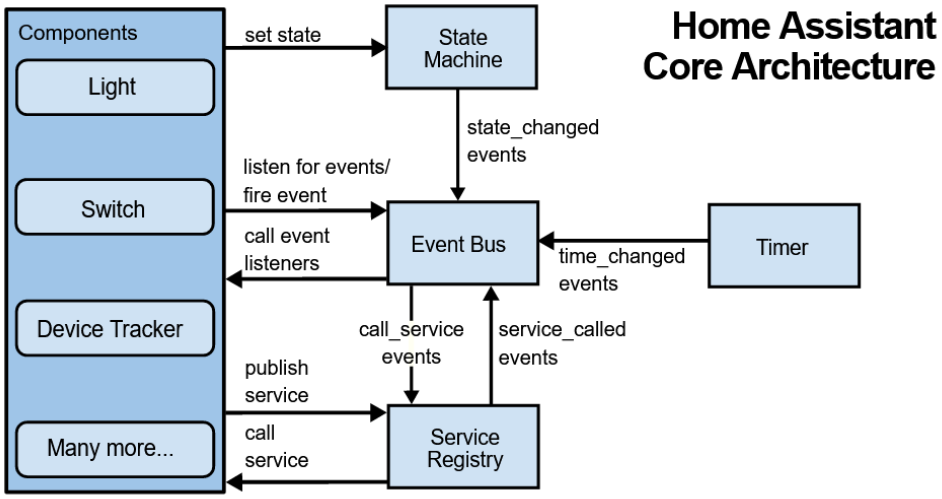
Home Assistant Core zajišťuje interakci mezi uživatelem, supervizorem a IOT zařízení a služeb. Skládá se ze čtyř hlavních částí, jejichž architektura je zobrazena na obrázku 4.3.

- **Event Bus**, tato část má na starostí vysílání a poslouchá eventům. Jedná se o jádro Home Assistant.
- **State Machine**, tato část monitoruje stavy všech komponent a hlásí state_changed event, když se některý stav změní
- **Service Registry**, tato část naslouchá event bus a zavolá službu, pokud přijde call_service event. Také umožňuje registraci služeb nových.
- **Timer**, tato část každou sekundu na event bus zašle event time_changed

Kromě těchto čtyřech částí obsahuje i pomocné třídy, které mají na starost zpracování častých scénářů z prostředí automatizace domácnosti.

4.2.2.4 Integrace

Home Assistant architektura byla navržena s ohledem na snadnou rozšiřitelnost. Rozšiřitelnosti je docíleno využitím **integrací**. Integrace je složená ze dvou částí, **komponenty** a **platformy**. Komponenta obsahuje



Obrázek 4.3: Architektura Home Assistant Core [33]

základní logiku prvku a platforma propojuje integraci s ostatními integracemi. Každá integrace je zodpovědná za specifickou doménu v Home Assistant. Mohou poslouchat nebo spouštět eventy, poskytovat služby a udržovat stavy.

Jsou čtyři typy integrací:

- **Integrace, které definují IoT doménu**, což jsou integrace, které definují konkrétní kategorii zařízení IoT v Home Assistant. Poskytují kontrolu nad zařízením a definují jaká zařízení jsou v Home Assistant dostupná a v jaké podobě. Příkladem takovéto integrace je *Light*.

- **Integrace, které interagují s externími zařízeními a službami**, což jsou integrace, které umožňují přidání externích zařízení a služeb do Home Assistant a interagují s nimi. Externí služby, nebo zařízení jsou reprezentovány skrze integraci předchozího typu. Příkladem takovéto integrace je *Philips Hue*, která je dostupná jako Light entita.

- **Integrace, které reprezentují virtuální/spočítané datové typy**, což jsou integrace, které reprezentují datové typy entit, buď na základě virtuálních dat, nebo na základě jiných dat v Home Assistant. Příkladem takovéto integrace je *input_boolean* pro první podtyp a *utility_meter* pro druhý podtyp. Pomocí *input_boolean* integrace je možné definovat boolean hodnoty, které mohou být použity v podmínkách automatizace. Utility_meter integrace poskytuje možnost trasování spotřeby různých utilit jako energie, plyn, voda a topení.

- **Integrace, které mohou být spuštěny uživatelem, nebo jako odpověď na eventy**, což jsou integrace, které poskytují části logiky automatizace domácnosti. Příkladem takovéto integrace je *automation*, která umožňuje uživatelům automatizaci skrze konfiguraci pro první podtyp a *flux*, která kontroluje světla na základě pohybů slunce pro druhý podtyp.

Pokud integrace reprezentuje zařízení, nazývá se **entitou**. Jsou různé strategie, jak udržovat Home Assistant synchronizovaný se stavem zařízení.

- **Polling (dotazování)**. Při této strategii se Home Assistant v daných intervalech ptá entity na její poslední stav. Home Assistant používá tuto strategii, pokud je **should_poll** vlastnost nastavena na *true* (výchozí nastavení). Jde implementovat vlastní aktualizační logiku pomocí metody **update()**, nebo pomocí asynchronní metody **async_update()**.

- **Odběr update**. Při této strategii je entita zodpovědná za informování Home Assistant o aktualizaci stavu. Home Assistant používá tuto strategii, pokud je **should_poll** vlastnost nastavena na *false*. Pokud entita dostane informace o novém stavu, zavolá metodu **schedule_update_ha_state()**, nebo asynchronní metodu **callback_async_schedule_update_ha_state()**, čímž informuje Home Assistant o změně.

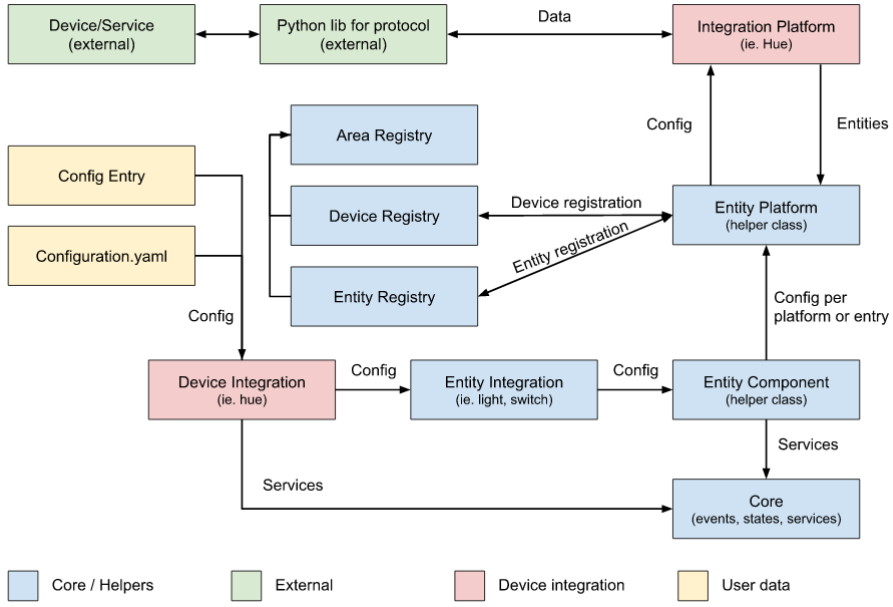
U entit jsou používané tzv. **Lifecycle hooks**, což jsou asynchronní metody, které spouští kód při určitých událostech. Jedná se o následující metody.

- **async_added_to_hass()**. Tato metoda je zavolána, pokud má entita přiřazené entity_id a hass objekt předtím, než je poprvé zapsána do state machine. Tato metoda je použita například pro obnovu stavu, odběr aktualizací a nastavení callback/dispatch funkce/posluchače.

- **async will remove from hass()**. Tato metoda je zavolána předtím, než je entita odebrána z Home Assistant. Mezi příklady použití patří odpojení od serveru, nebo zrušení odběru aktualizací.

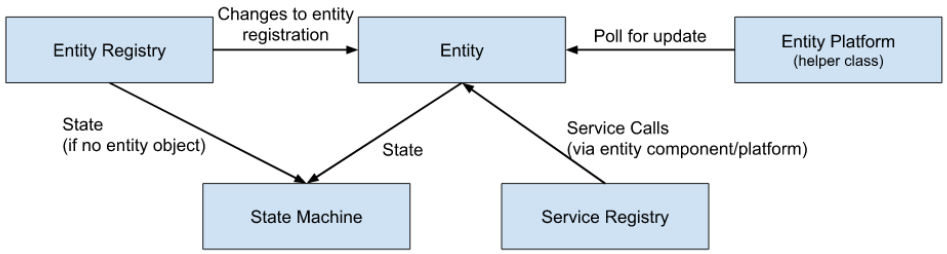
Na obrázku 4.4 je vidět, že pro rozšíření aplikace není potřeba se zabývat tím, jak funguje jádro Home Assistant. Místo toho stačí aby nově vytvořená entita, dědila od třídy **Entity** a implementovala nezbytné vlastnosti a metody pro typ zařízení, které je přidává. **Entity** je abstraktní třída entit a služeb, které entity ovládají.

Třída **Entity** je zodpovědná za formátování dat a jejich posílání **state machine**. Interakce třídy **Entity** je možné vidět na obrázku 4.5. Entita, od této třídy dědí, je zodpovědná za získávání dat, zpracování volání služeb a za informování Home Assistant o nových datech, pokud je **polling** nastaveno na *false*. Díky **Registru entit (Entity registry)** a **Registru**



Obrázek 4.4: Integrace entit do Home Assistant [33]

zařízení (Device registry), které si udržují informace o všech entitách a zařízení budou při smazání, zakázání, nebo naopak povolení jakéhokoli objektu entity, nebo zařízení, upraveny všechny objekty, které jsou na něj navázány. **Registr oblasti (Area Registry)** udržuje informace o oblastech. Oblast reprezentuje fyzickou lokaci pro Home Assistant. Mohou být použity k umístění zařízení do různých oblastí domácnosti.



Obrázek 4.5: Třída Entity a její interakce s Home Assistant Core [33]

Integrací je v Home Assistant v době psaní této práce 1800. Kromě integrace zařízení a služeb je možné pomocí integrací například i přidávat panely do Lovelace (integrace iframe Panel), sledovat stavy Supervizord (integrace Supervizord), nebo exportovat informace z Home Assistant pro možnost jejich zobrazení na zařízení používajících RSS reader (integrace RSS

Feed Template).

4.2.3 Data

Data jsou posílána dvěma způsoby. Jedním způsobem je posílání dat skrze objekt **hass**. Hass objekt je v aplikaci pouze jeden a obsahuje informace o posledním stavu aplikace. Umožňuje také posílat příkazy zpátky serveru. Hass objekt se posílá všem komponentám. Pokud se změní stav a aktualizuje se objekt hass, jsou vytvořené nové verze objektů, které byly změněny.

Druhým způsobem posílání dat je odběr skrze *WebSocket API*. Pro získání dat které nelze vyčíst z objektu hass, se mohou komponenty zaregistrovat k odběru informací u *WebSocket API*.

Datový vstup **config entry** na obrázku 4.4 je vytvořen pomocí famework **Data Entry Flow**. **Config entry** jsou konfigurační data, která jsou v Home Assistant ukládána perzistentně.

Ve výchozím nastavení je pro ukládání dat použita databáze *SQLite*. Pomocí integrace **Recorder** je možné databázi změnit. Recorder používá ORM (Object Relational Mapper) *SQLAlchemy* a je tedy možné nastavit jakoukoli databázi, která je SQLAlchemy podporována,jako například *MySQL*, *MariaDB*, *PostgreSQL*, nebo *Microsoft SQL Server*.

4.2.4 Automatizace zařízení

Home Assistant poskytuje možnost definovat automatizační pravidla, kterými se dají nastavit procesy chytré domácnosti a propojit funkcionality zařízení. Automatizace poskytuje uživateli vrstvu nad základem Home Assistant se zaměřením na zařízení. Při vytváření automatizací se uživatel nemusí zabývat stavy a eventy. Místo toho si vybere zařízení a poté ze seznamu možných, předdefinovaných **spouštěčů (Trigger)**, **podmínek** a **akcí** stanoví, co se jak má zařízení na tyto prvky reagovat.

Spouštěče (trigger) zařízení jsou vázané na konkrétní zařízení a event, nebo změnu stavu. Například zapnutí světla. Pokud integrace podporuje spouštěče zařízení, má ve své složce soubor **device_trigger.py**

Podmínky zařízení umožnují uživateli zkontrolovat, jestli jsou splněny. Například zkontrolovat stav zařízení. Pokud integrace podporuje podmínky zařízení, má ve své složce soubor **device_condition.py**

Akce zařízení umožňují uživateli provést akci se zařízením. Například změnit stav zařízení. Pokud integrace podporuje akce zařízení, má ve své složce soubor **device_action.py**

4.2.5 Autentizace

Home Assistant má autentizační systém, který umožňuje přihlášení a nastavení práv uživatelům. Metodu autentizace a její provedení má na starosti

Autentizační poskytovatel. Ve výchozím nastavení je použitý poskytovatel Home Assistant. V souboru **configuration.yaml** lze poskytovatele změnit.

Uživatel je člověk v systému. K přihlášení do Home Assistant je potřeba, aby se uživatel autentizoval u poskytovatele. Uživatel, který je vytvořen při procesu *onboarding* je označen jako **Vlastník**. Ten spravuje ostatní uživatele a má vždy všechna přístupová práva. Autentizace uživatele je uložena pomocí **přístupových údajů**. Je možné mít více přístupových údajů, ale vždy pouze jedny přístupové údaje pro jednoho poskytovatele. Uživatelé jsou členy **skupin**. Skupinám jsou přiřazena práva pomocí **pravidel**. Pravidla určují, ke kterým zdrojům mají mít konkrétní skupiny přístup.

Uživatel může poskytnout přístup do Home Assistant aplikacím třetích stran skrze **Home Assistant autentizačního API**. Přístup je zajištěn pomocí **přístupové token (Access Token)**. Tento token je platný pouze po určitou dobu a musí být obnoven. Za obnovení přístupového tokenu je zodpovědný **obnovovací token (Refresh Token)**, který je tří typů:

- *Normální*, což jsou tokeny, které jsou generovány při autorizaci aplikace třetí strany.
- *Dlouhodobý*, což jsou tokeny, které udržují dlouhodobé přístupové tokeny. Ty jsou validní po dobu deseti let a jsou používány pro integraci API třetích stran.
- *Systémový*, což jsou tokeny, které jsou určené systémovým uživatelům jako Home Assistant OS a Supervizor.

Kapitola 5

Analýza

Vzhledem k cíli této práce, kterým je zhodnocení návrhu aplikace, jsem se soustředila na metody postavené nad strukturální analýzou.

Vybrala jsem si metodu K. Keller et al. (1999) [22]. Motivací k výběru této metody byla existence vývojářské dokumentace, která poskytuje popis architektury na vysoké úrovni abstrakce. Je tedy možné získat znalosti, které pomáhají k vytvoření modelu zdrojového kódu. Druhým důvodem výběru byla možnost část procesu automatizovat pomocí dostupných nástrojů.

5.1 Průběh analýzy

Dle vybrané metody bylo potřeba si vytvořit model zdrojového kódu, repozitář abstraktních návrhových komponent a repozitář implementovaných návrhových komponent.

Pro vytvoření **modelu zdrojového kódu** jsem nejprve analyzovala existující dokumentaci a vytvořila si představu o jednotlivých komponentách systému a jejich vzájemné komunikaci. Strategii jsem zvolila podobou jako v [24] a postupovala jsem po jednotlivých úrovních abstrakce směrem ze shora dolu. Pomocí nástroje enterprise architect [41] jsem vygenerovala UML Structural: class diagram. Dále jsem pomocí nástroje PyCharm [42] vygenerovala třídní hierarchii pro třídu Entity. Aby se v třídní hierarchii lépe vyhledávalo, extrahovala jsem data vygenerovaná nástrojem PyCharm do csv, upravila jeho strukturu a postavila nad nimi jednoduchý PowerBI [43] report. Pro statickou analýzu kódu jsem opět zvolila nástroj PyCharm. Vygenerovaný class diagram doplněný o chybějící informace s pomocí vývojářské dokumentace a třídní hierarchie jsem použila jako model zdrojového kódu.

Pro vytvoření **repozitáře abstraktních návrhových komponent** jsem použila [2].

Pro vytvoření **repozitáře implementovaných návrhových komponent** jsem použila [3].

Následně bylo pořeba vykonat samotnou *design recovery*. Pomocí vytvořených repozitářů s abstraktními a implementovanými návrhovými komponentami, vyhledávat kandidáty na návrhové vzory a tyto kandidáty poté seskupit do struktur, které by odpovídali návrhovým vzorům z repozitářů. Tento proces jsem prováděla manuální cestou.

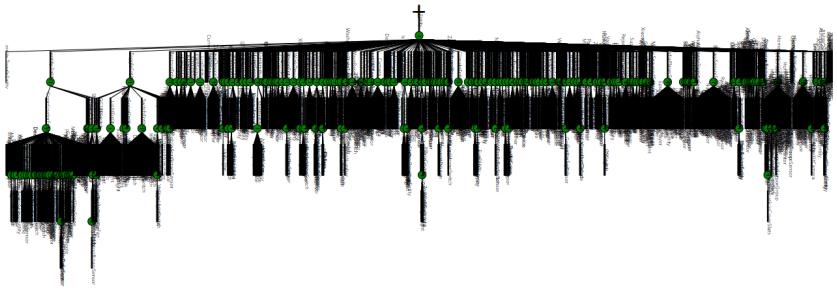
Nejprve jsem se soustředila na analýzu celkové architektury, tak jak je popsána v [33] a její zhodnocení. V této fázi jsem především hledala využití návrhových principů. Už v této fázi ovšem bylo možné identifikovat použité návrhové vzory. V další fázi jsem postupovala dle popsaných komponent hierarchicky směrem dolu, do větších detailů.

Tento postup jsem opakovala pro každý návrhový vzor z vytvořeného repozitáře návrhových vzorů.

K nálezu návrhových vzorů dopomáhal i fakt, že je kód Home Assistant řádně okomentován. Čtením komentářů bylo tedy také možné vytypovat místa použití návrhových vzorů a bylo nutné pouze zkontrolovat, zda jsou informace v komentářích pravdivé a namapovat zdrojový kód na návrhové vzory v repozitáři implementovaných návrhových komponent.

5.2 Výsledky analýzy

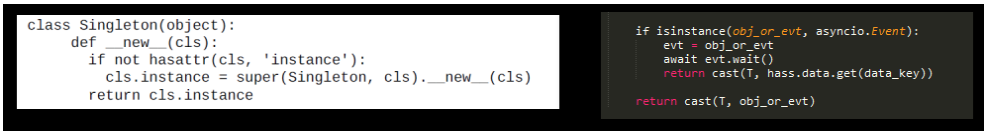
Již při analýze dokumentace a instrukcí pro vývoj bylo patrné, že vývojáři Home Assistant jsou si vědomi výhod, které s sebou implementace návrhových vzorů a principů přinášejí. Systém je navržený jako modulární s důrazem na snadnou rozšiřitelnost, spolehlivost a striktní dodržování zavedených standardů. Vnitřní logika je odstíněná a přidání integrace nového zařízení znamená pouze dědičnost od třídy **Entity** a implementaci vlastností specifických pro přidanou integraci. Na obrázku 5.1 lze vidět graf tříd dědících od třídy **Entity**.



Obrázek 5.1: Třídy, které dědí od třídy Entity

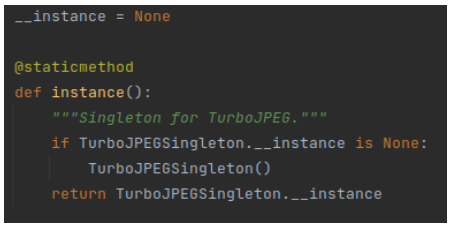
5.2.1 Využití návrhových vzorů

Hass objekt je v aplikaci pouze jeden. Již z celkové architektury tedy bylo patrné, že Home Assistant využívá návrhový vzor **singleton**. Soubor **singleton.py** v modulu helpers tento návrhový vzor implementuje. Porovnání kódu z repozitáře návrhových vzorů (vlevo) a kódu aplikace (vpravo), je možné vidět na obrázku 5.2. Singleton je kromě objektu hass použit i pro instanci



Obrázek 5.2: Porovnání singleton

objektu **Zeroconf**, konktext CoAP komponenty **shelly**, add-on manažera komponenty **zwave_js**, pomocníka při obnově stavu, SNMP engine komponenty **brother**, vzorů pro automatizaci komponenty **automation** a objekt třídy **TurboJPEGSingleton**. Třída **TurboJPEGSingleton** ale nevyužívá **singleton.py** souboru, nýbrž vlastní implementace. Jeho implementace je možná vidět na obrázku 5.3.



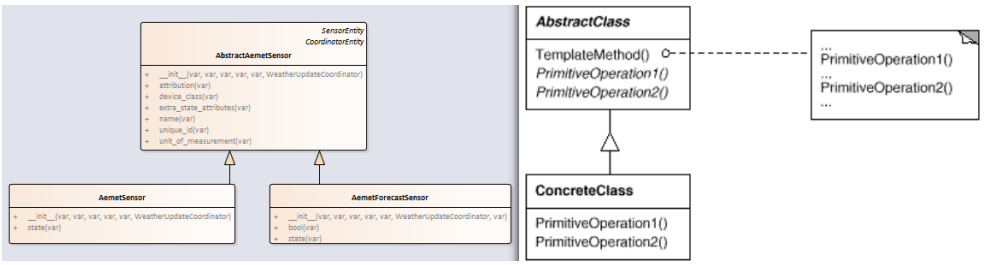
Obrázek 5.3: TurboJPEGSingleton

Home Assistant získává informaci ohledně stavu zařízení pomocí **pozorovatele**. **Should_poll** vlastnost určuje, jestli bude využíván push, nebo pull model. V souboru **core/helpers/collection.py** je třída **ObservableCollection**, která subjektem v tomto návrhovém vzoru, jak je možno vidět z obrázku 5.4.

Obrázek 5.4: Pozorovatel z repozitáře návrhových vzorů

Dále je využit návrhový vzor **dekorátor**. V soboru **helpers.py** komponenty **toon** je funkce **toon_exception_handler**, která implementuje tento návrhový vzor. Dále je tento návrhový vzor využit u entity **MediaPlayer**, kdy se za běhu přidávají příkazy k ovládání zvuku, nebo u entity **abd_decorator**.

Návrhový vzor **šablona** je využíván často. Příkladem takového užití může být **AemetSenzor** na obrázku 5.5, porovnaný s návrhovým vzorem šablony z repozitáře návrhových vzorů.



Obrázek 5.5: Porovnání šablony

5.2.2 Využití principů teorie Normalizovaných systémů

Z analýzy je zřejmé, že Home Assistant již od počátku vývoje klade důraz na snadnou rozšiřitelnost. Home Assistant je vícevrstvá aplikace s jasně definovanými moduly a komponenty. Skrze celou aplikaci využívá principu **oddělení odpovědnosti**. Jediným nalezeným místem, kde byl tento princip porušen byla metoda async_turn_on v komponentě home connect. Tato akční úloha má na starosti zapínání světla, změnu jeho barvy a změnu jeho jasu.

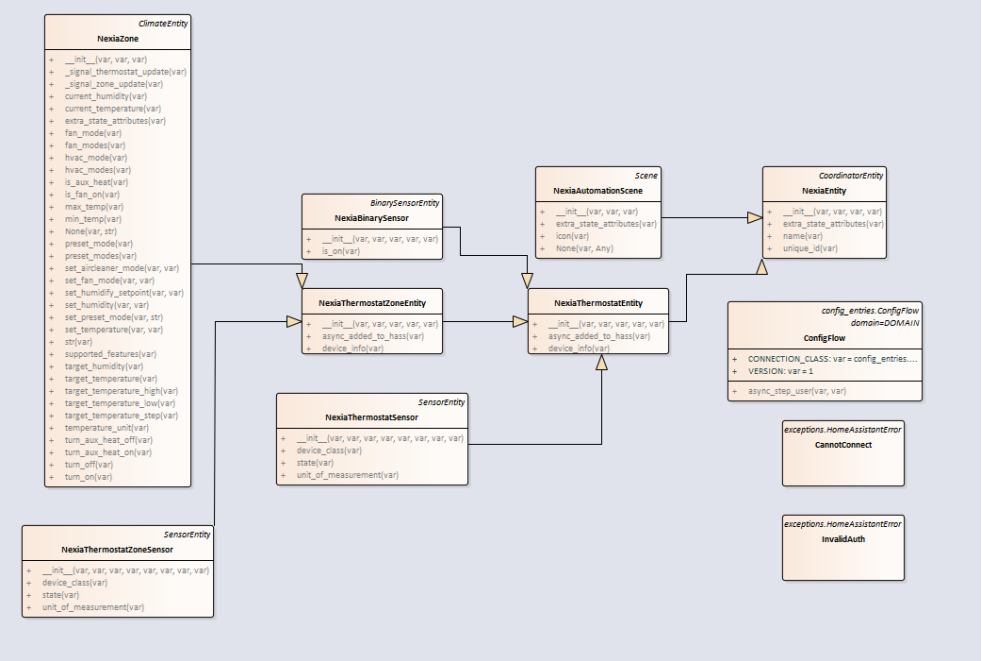
Princip oddělení stavu je dodržen. Hass objekt se posílá všem komponentám. Pokud se změní stav a aktualizuje se objekt hass, jsou vytvořený nové verze objektů, které byli změněny. Krom toho se o oddělení stavů starají registr entit, registr zařízení a registr oblastí. V aplikaci se používá asynchronního volání.

Logování a zpracování chyb je provedeno skrze separátní entity, což napomáhá dodržování principů **transparentnost akční verze** a **Data Version Transparency**.

Princip transparentnost akční verze je navíc dodržován skrze polymorfismus. **Ptransparentnost akční verze** je navíc dodržován pomocí **Config Flow Handlers**, které kontrolují data uložená v **config entry**. Díky nim je možné zmigrovat **config entry** do nového formátu, pokud se změní verze. Příklad lze vidět na obrázku 5.6 komponenty **nexia**.

5.2.3 Využití dalších návrhových principů

Data která nelze vyčíst ze stavu hass mohou komponenty získávat odběrem skrze *WebSocket API*. Je tedy využit **princip Hollywoodu**.



Obrázek 5.6: Config Flow v komponentě nexia

Princip otevřenosti a uzavřenosti je dodržován, vzhledem k tomu, že je Home Assistant navržen tak, aby bylo snadné přidávat integrace nových komponent.

Pro analýzu dodržení principů **DRY, WET a AHA** byla použita statická analýza kódu. Statická analýza odhalila v core 3 758 duplicitních fragmentů kódu, z toho 2964, pokud byla inspekce spuštěna na odhalení duplicitních fragmentů v rámci jednoho souboru. Provedla jsem důkladnější analýzu těchto výsledků, ovšem neodhalila porušení **principu DRY**. Příklad detekované duplikace kódu zobrazuje obrázek 5.7. V plné velikosti je obrázek dostupný v příloze. Velkou skupinou detekovaných duplikací byli testovací soubory, ve kterých se duplicity očekávají. Výsledek analýzy je důsledek toho, že při pull request prochází nový kód důkladnou revizí. Vzhledem k dodržení **principu DRY** jsou porušeny **principy WET a AHA**. Díky navržené architektuře Home Assistant a snadnému přidání nových komponent to však není na závadu. Jediným sporným místem je komponenta **home_connect**, která má úroveň abstrakce navíc v podobě skupin zařízení.

Obrázek 5.7: Detekce duplicitního kódu

Kapitola 6

Návrh

Vzhledem k tomu, že Home Assistant návrhové vzory a principy využívá, jsou navrhované změny tedy pouze drobné.

6.1 Rozdělení async_turn_on entity light komponenty home_connect

Tato metoda světlo nejen zapíná, nýbrž i mění barvu a jas. Nedodržuje tím princip jedné odpovědnosti. Navrhuji metodu rozdělit do tří, z nichž každá bude mít jen jednu z těchto zodpovědností.

- *Přínos:* Nedodržení principu jedné odpovědnosti vede ke kombinatorickým efektům, které vedou k nestabilitě systému.
- *Odhad pracnosti:* 2MD
- *Priorita:* Vysoká

6.2 Přepracování struktury abstrakcí komponenty home_connect

. Tato komponenta porušuje princip AHA a KISS. Princip AHA je porušen protože v komponentě je krom koncových zařízení také skupiny zařízení, do kterých zařízení spadá. Jedno zařízení může spadat do více skupin.

- *Přínos:* Zjednodušení implementace
- *Odhad pracnosti:* 2MD
- *Priorita:* Střední

6.3 Restrukturalizace vývojářská dokumentace

Tato úprava s návrhovými vzory souvisí nepřímo. Vývojářská dokumentace obsahuje většinu důležitých informací. Je však špatně strukturována. Například informace o Supervizoru jsou dostupné pouze z informací o integraci **Supervizord**, nikoli ze sekce supervizora v přehledu architektury. Obdobně informace ohledně databáze je pouze v informacích o integraci **Recorder**. Informace o OS jsou pouze v GitHub. Navrhuji pozměnit strukturu dokumentace, nebo přidat do bočního panelu odkazy na všechny informace. Sekci Home Assistant této práce je možné využít jako vzor pro navrhovanou strukturu.

- *Přínos:* Kontributoři si budou lépe vědomi celkové architektury a využitých návrhových vzorů a principů. Může to přinést více vývojářů s kvalitními vstupy.
- *Odhad pracnosti:* 1MD
- *Priorita:* Nízká

Závěr

Výsledkem práce je zhodnocení návrhu aplikace Home Assistant a doporučení k jeho vylepšení. Aplikace byla zhodnocena jako kvalitně navržena.

Přesto jsou v práci navrženy změny aplikace spojené s využitím návrhových vzorů a principů. Navržené změny byly odůvodněny shrnutím jejich přínosů. Doporučení bylo doplněné o odhad pracnosti.

Analýza byla provedena pomocí metod reverzního inženýrství. Z velkého množství metod v tomto odvětví byly vybrány takové metody, které jsou vhodné k analýze návrhových vzorů a principů. Metody byli prezentovány v přehledné podobě. Metody, které byly použity, nebo byly použity jejich části, či myšlenky, jsou rozvedeny do bližších detailů.

Aplikace Home Assistant je aktivně vyvíjena s měsíčním release. V rámci release je upravována i vývojářská dokumentace, která byla klíčová pro vytvoření doménového modelu. I přes komplikaci v podobě častých změn a nutnosti závěry přepracovat, byli všechny stanovené cíle úspěšně naplněny.

Možností rozšíření této práce, by mohlo být vybrání několika metod z poskytnutého přehledu a porovnání jejich efektivity, nebo dostupných nástrojů pro částečnou, či plnou automatizaci. Případně lze na práci navázat v podobě implementace navrhovaných změn.

Literatura

[1] MARTIN, Robert C. *Design Principles and Design Patterns.* [online]. 2001. Dostupné z: https://fi.ort.edu.uy/innovaportal/file/2032/1/design_principles.pdf

[2] GAMMA, Erich. *Design patterns: elements of reusable object-oriented software*. Reading, Mass.: Addison-Wesley, c1995. ISBN 978-0201633610.

[3] GIRIDHAR, Chetan. *Learning Python Design Patterns Second Edition.* Packt Publishing Ltd, 2016. ISBN 978-1-78588-803-8.

[4] LARMAN, Craig. *Applying UML and patterns: an introduction to object-oriented analysis and design and iterative development*. 3rd ed. Upper Saddle River, N.J.: Prentice Hall PTR, c2005. ISBN 978-0131489066.

[5] Interaction Design Foundation (IxDF). UX Design Courses & Global UX Community — Interaction Design Foundation (IxDF). *KISS (Keep it Simple, Stupid) - A Design Principle* [online]. Dostupné z: https://www.interaction-design.org/literature/article/kiss-keep-it-simple-stupid-a-design-principle

[6] HUNT, Andrew a David THOMAS. *The pragmatic programmer: from journeyman to master*. Reading, Mass: Addison-Wesley, 2000. ISBN 978-0201616224.

[7] DURBIN, Conlin. *Stop trying to be so DRY, instead Write Everything Twice (WET)* [online]. Copyright © [20.12.2018]. Dostupné z: https://dev.to/wuz/stop-trying-to-be-so-dry-instead-write-everything-twice-wet-5g33

[8] DODDS, Kent C. *AHA Programming* [online].[22.6.2020]. Dostupné z: https://kentcdodds.com/blog/aha-programming

[9] MANNAERT, H. a J. VERELST. *Normalized Systems: Re-creating Information Technology based on Laws for Software Evolvability*. Koppa, 2009. ISBN 978-90-77160-00-8-S.

[10] JANEČEK, L. Bc. *Studie volby databázové platformy pro realizaci normalizovaných softwarových systémů*. Diplomová práce, Katedra softwarového inženýrství, obor Webové a softwarové inženýrství. [online]. 2014. Dostupné z: https://dspace.cvut.cz/handle/10467/62995

[11] CHIKOFSKY, E.J. a J.H. CROSS. *Reverse engineering and design recovery: a taxonomy*. IEEE Software [online]. 1990, 7(1), 13-17 [cit. 2021-6-22]. ISSN 0740-7459. Dostupné z: doi:10.1109/52.43044

[12] BIGGERSTAFF, T.J. *Design recovery for maintenance and reuse*. Computer [online]. 1989, 22(7), 36-49 [cit. 2021-6-22]. ISSN 0018-9162. Dostupné z: doi:10.1109/2.30731

[13] SADIQ, Jawaria a Tabinda WAHEED. *Reverse engineering & design recovery: An evaluation of design recovery techniques*. In: International Conference on Computer Networks and Information Technology [online]. IEEE, 2011, 2011, s. 325-332 [cit. 2021-6-22]. ISBN 978-1-61284-940-9. Dostupné z: doi:10.1109/ICCNIT.2011.6020889

[14] BALL, Thomas. *The Concept of Dynamic Analysis*. NIERSTRASZ, Oscar a Michel LEMOINE, ed. Software Engineering — ESEC/FSE '99 [online]. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, 1999-8-27, s. 216-234 [cit. 2021-6-22]. Lecture Notes in Computer Science. ISBN 978-3-540-66538-0. Dostupné z: doi:10.1007/3-540-48166-4 14

[15] Semantic Designs: Control and Data Flow Analysis. Semantic Designs [online].[cit. 24.06.2021]. Dostupné z: http://www.semdesigns.com/products/DMS/FlowAnalysis.html

[16] ARNOLD, R.S. a S.A. BOHNER. *Impact analysis-Towards a framework for comparison*. In: 1993 Conference on Software Maintenance [online]. IEEE Comput. Soc. Press, 1993, s. 292-301 [cit. 2021-6-24]. ISBN 0-8186-4600-4. Dostupné z: doi:10.1109/ICSM.1993.366933

[17] IBM Docs. [online]. Copyright © Copyright IBM Corporation 2019. [cit. 24.06.2021]. Dostupné z: https://www.ibm.com/docs/de/wsr-and-r/8.5.6?topic=analysis-dependency

[18] WEISER, Mark. *Program Slicing*. IEEE Transactions on Software Engineering [online]. 1984, SE-10(4), 352-357 [cit. 2021-6-22]. ISSN 0098-5589. Dostupné z: doi:10.1109/TSE.1984.5010248

[19] TIP, Frank. *A survey of program slicing techniques*. Journal of Programming Languages, vol.3, no.3, pp.121–189. 1995. [online]. Dostupné z: https://www.franktip.org/pubs/jpl1995.pdf

[20] OTTENSTEIN, Karl J. a Linda M. OTTENSTEIN. *The program dependence graph in a software development environment*. In: Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments - SDE 1 [online]. New York, New York, USA: ACM Press, 1984, 1984, s. 177-184 [cit. 2021-6-22]. ISBN 0897911318. Dostupné z: doi:10.1145/800020.808263

[21] SRIDHARAN, Manu, Stephen J. FINK a Rastislav BODIK. *Thin slicing*. In: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation - PLDI '07 [online]. New York, New York, USA: ACM Press, 2007, 2007, s. 112- [cit. 2021-6-22]. ISBN 9781595936332. Dostupné z: doi:10.1145/1250734.1250748

[22] KELLER, Rudolf K., Reinhard SCHAUER, Sébastien ROBITAILLE a Patrick PAGÉ. *Pattern-based reverse-engineering of design components*. In: Proceedings of the 21st international conference on Software engineering - ICSE '99 [online]. New York, New York, USA: ACM Press, 1999, 1999, s. 226-235 [cit. 2021-6-22]. ISBN 1581130740. Dostupné z: doi:10.1145/302405.302622

[23] WENDEHALS, Lothar. *Dynamic Design Pattern Recognition*. Dissertation proposal. Software Engineering Group, Department of Computer Science,University of Paderbom Warburger StraBe 100, 33098 Paderborn, Germany. 2005. Dostupné z: https://www.yumpu.com/en/document/view/23207945/dynamic-design-pattern-recognition

[24] LAKHOTIA, Arun. *An Approach to Recovering Data Flow Oriented design of a software system*, The Center for Advanced Computer Studies, University of Southwest em Louisiana, 1993. Dostupné z: https://people.cmix.louisiana.edu/arun/papers/TR-92-5-9.pdf.

[25] SHI, Nija a Ronald A. OLSON. *Reverse Engineering of Design Patterns for High Performance Computing*, Department of Computer Science University of California, Davis, California 95616-8562. 2005. Dostupné z: https://www.cs.ucdavis.edu/ olsson/pubs/2005/shi.pdf

[26] ANTONIOL, G., R. FIUTEM a L. CRISTOFORETTI. *Design pattern recovery in object-oriented software*. In: Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No.98TB100242) [online]. IEEE Comput. Soc, 1998, s. 153-160 [cit. 2021-6-22]. ISBN 0-8186-8560-3. Dostupné z: doi:10.1109/WPC.1998.693342

[27] GALL, Harald, René KLÖSCH a Roland MITTERMEIR. *Object-oriented re-architecturing*. SCHÄFER, Wilhelm a Pere BOTELLA, ed. Software Engineering — ESEC '95 [online]. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, 1995-8-18, s. 499-519 [cit. 2021-6-22]. Lecture Notes in Computer Science. ISBN 978-3-540-60406-8. Dostupné z: doi:10.1007/3-540-60406-5_32

[28] ROBITAILLE, SCHAUER a KELLER. *Bridging program comprehension tools by design navigation*. In: Proceedings International Conference on Software Maintenance ICSM-94 [online]. IEEE Comput. Soc. Press, 2000, 2000, s. 22-32 [cit. 2021-6-22]. ISBN 0-8186-6330-8. Dostupné z: doi:10.1109/ICSM.2000.882972

[29] KONTOGIANNIS , K., M. BERNSTEIN, E. MERLO, R. DE MORI. *The Development of a Partial Design Recovery Environment for Legacy Systems*. In Proceedings of the 1993 CAS Converence, pages 206-216, Toronto, Ont., Canada. 1993. IBM Canada Ltd. Laboratory, Centre for Advanced Studies. Dostupné z: doi: 10.1145/962308

[30] CAREY, M.M. a G.C. GANNOD. *Recovering Concepts from Source Code with Automated Concept Identification*. In: 15th IEEE International Conference on Program Comprehension (ICPC '07) [online]. IEEE, 2007, 2007, s. 27-36 [cit. 2021-6-22]. ISBN 0-7695-2860-0. Dostupné z: doi:10.1109/ICPC.2007.31

[31] NIERE, J., J. WADSACK a L. WENDEHALS, 2001. *Design Pattern Recovery Based on Source Code Analysis with Fuzzy Logic*.Department of Mathematics and Computer Science, University of Padernorn, Paderborn, Germany. 2001. Dostupné z: https://www.hni.uni-paderborn.de/publikationen/publikationen/?tx_hnippview_pi1%5bpublikation%5d=7204

[32] HEUZEROTH, D., T. HOLL, G. HOGSTROM a W. LOWE. *Automatic design pattern detection*. In: MHS2003. Proceedings of 2003 International Symposium on Micromechatronics and Human Science (IEEE Cat. No.03TH8717) [online]. IEEE Comput. Soc, 2003, s. 94-103 [cit. 2021-6-22]. ISBN 0-7695-1883-4. Dostupné z: doi:10.1109/WPC.2003.1199193

[33] Home Assistant Developer Documentation [online]. 2021 [cit. 22.06.2021]. Dostupné z: https://developers.home-assistant.io

[34] Home Assistant User Documentation [online]. 2021 [cit. 22.06.2021]. Dostupné z: https://www.home-assistant.io/getting-started/

[35] Home Assistant Developer Documentation. *Licence* [online]. 2021 [cit. 22.06.2021]. Dostupné z: https://www.home-assistant.io/developers/license/

[36] Home Assistant Developer Documentation. *Credits* [online]. 2021 [cit. 22.06.2021]. Dostupné z: https://www.home-assistant.io/developers/credits/

[37] Home Assistant Code [online]. 2021 [cit. 22.06.2021]. Dostupné z: https://github.com/home-assistant

[38] PyPi, Black code formatter [online]. 2021 [cit. 22.06.2021]. Dostupné z:https://github.com/psf/black

[39] Python.org, PEP 8 Style Guide for Python Code [online]. 2021 [cit. 22.06.2021]. Dostupné z: https://www.python.org/dev/peps/pep-0008/

[40] Python.org, PEP 257 Docstring Conventions [online]. 2021 [cit. 22.06.2021]. Dostupné z: https://www.python.org/dev/peps/pep-0257/

[41] SparxSystems, Enterprise Architect [online]. 2021 [cit. 22.06.2021]. Dostupné z: https://sparxsystems.com/

[42] JetBrains, PyCharm Professional [online]. 2021 [cit. 22.06.2021]. Dostupné z: https://www.jetbrains.com/pycharm/

[43] Microsoft, PowerBI Desktop [online]. 2021 [cit. 22.06.2021]. Dostupné z: https://powerbi.microsoft.com/en-us/downloads/

Příloha A

Seznam použitých zkratek

AHA Avoid Hasty Abstractions

API Application Programming Interface

DFD Data Flow Diagram

DNS Domain Name System

DRY Don't Repeat Yourself

EFI Extensible Firmware Interface

ESB Enterprise service bus

GUI Graphical user interface

IoT Internet of things

ORM Object Relational Mapping

OS Operating System

OTA Over The Air

RAUC Robust Auto-Update Controller

RSS Rich Site Summary

USB Universal Serial Bus

WET Write Everything Twice

Příloha B

Obsah přiloženého CD

- `README.txt` ..... stručný popis obsahu CD
- `chodomar_Thesis` ..... adresář se zdrojovou formou práce
- `chodomar_Thesis.pdf` ..... text práce ve formátu PDF
- `duplicity_example.png` .. příklad nalezených duplicit statickou analýzou
- `models` ..... adresář s modely použitými v praktické části práce
  - `core-master` ..... vygenerovaný model nad kódem Core
  - `supervisor-main` ..... vygenerovaný model nad kódem Supervizor