Slovenská technická univerzita v Bratislave

Fakulta informatiky a informačných technológií

3D UML

1. kontrolný bod - Inžinierske dielo

Tím č. 15

Vedúci tímu: Ing. Ivan Polášek, Phd.

Členovia tímu: Bc. Matej Víťaz, Bc. Ľubomír Jesze, Bc. Lukáš Skala, Bc. Peter Zajac, Bc. Dominik Žilka, Bc. Matúš Gáspár, Bc. Jakub Minárik

Akademický rok: 2016/2017

Obsah

1 Úvod

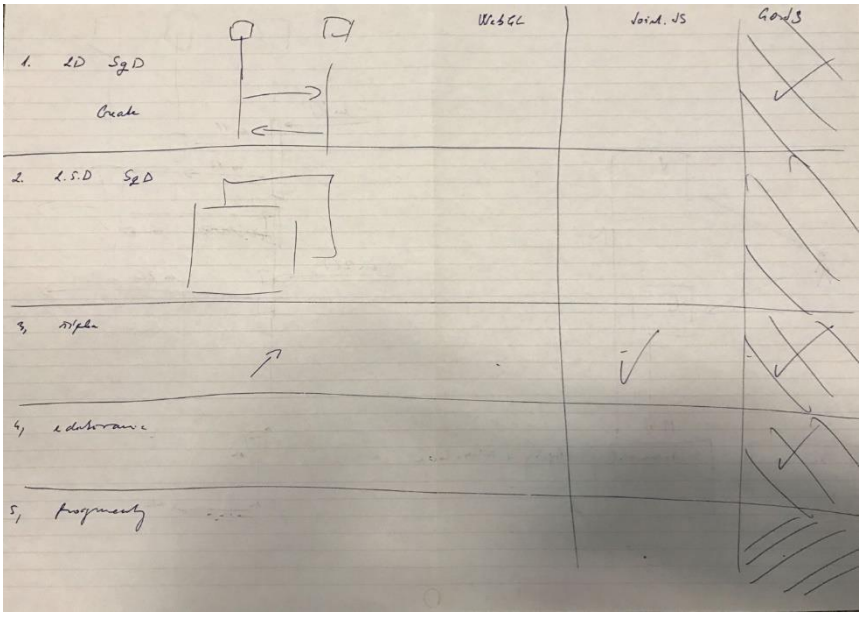
Dokument inžinierskeho diela bol vytvorený z dôvodu zdokumentovania našej tímovej práce v akademickom roku 2016/2017 v predmete Tímový projekt. Náš tím, číslo 15 s názvom Next Dimension Team, sa podieľa na projekte s názvom 3D UML, a teda vytvorenie 3D prostredia na modelovanie UML diagramov. Konkrétne sa v našom projekte špecializujeme na vytvorenie tohto prostredia pre sekvenčný diagram. Dokument je rozdelený na niekoľko častí. V úvode sa čitateľ oboznámi so všeobecnou problematikou, ktorú riešime. V prvej časti sa dozvie náš cieľ, ktorý chceme dosiahnuť v rámci zimného semestra a získa prehľad o tom, čo plánujeme v rámci projektu urobiť a aký je celkový, všeobecný, pohľad na aplikáciu. V druhej časti sú ozrejmené všetky moduly systému, ktoré používame. Postupne prejdeme každým modulom aplikácie a ozrejmíme jeho časti. Moduly ktoré sa nachádzajú v našej aplikácii sú - vykresľovanie diagramov, serverová aplikácia a jadro klientskej aplikácie. Pre prehľadnosť sme rozdelili každý opis modulu na štyri časti - analýza, návrh, implementácia a testovanie.

1.1 Globálne ciele projektu

V tejto kapitole uvádzame ciele, ktoré sme si stanovili po konzultácii s pedagogickým vedúcim projektu na jednotlivé semestre. Obsahuje popis častí projektu, ktoré by sme chceli implementovať.

Aplikáciu sme na stretnutí s vedúcim si rozdelili na niekoľko častí:

1. Vytvorenie 2D sekvenčného diagramu
2. Plátna diagramu – 2.5D
3. Prepojenie jednotlivých plátien (3D šípka) - vytvorené ako diplomová práca a bude potrebné integrovať do nášho projektu
4. Editovanie diagramu
5. Fragmenty sekvenčného diagramu – bude vytvorené ako bakalárska práca a integrované do projektu



Obrázok: Plán projektu, tak ako bol načrtnutý vedúcim projektu

1.1.1 Ciele pre zimný semester

V zimnom semestri je naším cieľom implementovať perzistenciu diagramu do databázy a jeho načítavanie. Navyše sa zameriame na implementáciu prvých dvoch bodov, ktoré sme uviedli v predchádzajúcej kapitole. Konkrétne ide o nasledujúce časti:

1. Integrovanie knižnice, ktorá vykresľuje 2D diagramy – ide o použitie už existujúcej knižnice, ktorú integrujeme do nášho projektu, aby bolo možné pracovať s diagramom na jednotlivých vrstvách
2. Možnosť v aplikácií pridávať plátna - plátna predstavujú 3D aspekt, kde jednotlivé diagramy sú v priestore za sebou a na jednotlivých vrstvách sú 2D diagramy (ide vlastne o 2.5D)

Po ukončení viacerých šprintov sa nám podarilo zistiť, že výhodnejšie bude namiesto použitia knižnice na vytváranie 2D diagramov, naprogramovať svoje vlastné jadro na vykresľovanie diagramu. Preto našim hlavným cieľom bolo, aby sme takúto funkcionalitu naprogramovali.

1.2 Celkový pohľad na 3D UML aplikáciu

Náš systém sa skladá z dvoch častí aplikácie, jedna je serverová, druhá je klientská. Serverová je tá, ktorá obstaráva komunikáciu s databázou na vzdialenom zariadení, zatiaľ čo klientská časť aplikácie sa zobrazuje používateľovi v prehliadači. Klientská časť aplikácie je tá, s ktorou používateľ komunikuje, čiže v prípade UML modelovania je to tá, kde kreslí, posúva, maže, alebo upravuje diagramy, resp. jednotlivé súčasti diagramov.

Serverová a klientská časť spolu komunikujú prostredníctvom JSON API.

Na obrázku dole je znázornená základná architektúra nášho systému.

Obrázok: Architektúra aplikácie

2 Moduly systému

Nasledujúca kapitola obsahuje popis vykonanej analýzy, návrhu a implementácie jednotlivých modulov aplikácie, ktoré boli realizované počas prvých troch šprintov tímového projektu. Tieto moduly boli následne testované ako u vádzajú podkapitoly jednotlivých modulov.

2.1 Vykresľovanie diagramov

Táto kapitola popisuje modul aplikácie, ktorého úlohou je vykresľovanie 2D UML diagramov, ktoré budú integrované do 3D prostredia pomocou ďalších modulov aplikácie. Kapitola sa zaoberá analýzou použiteľných technológií a príslušných knižníc, zakončenou zhodnotením výhod a nevýhod jednotlivých možností. Taktiež obsahuje vysvetlenie návrhu a implementácie tohto modulu aplikácie.

2.1.1 Analýza

2.1.1.1 Porovnanie vykresľovacích technológií

Dôležitým krokom pri tvorbe 3D prostredia je výber vykresľovacej technológie, ktorá najviac vyhovuje naším potrebám na tvorbu 3D UML diagramov. V našom prípade prichádzajú do úvahy dve alternatívy, ktorými sú WebGL a CSS 3D.

WebGL je JavaScriptové API, ktoré umožňuje vykresľovanie 3D grafiky vo webovom prehliadači. Grafika aplikácie vytvorená prostredníctvom WebGL je renderovaná do HTML elementu canvas a je spustiteľná na grafickej karte s podporou shader renderingu, vďaka čomu je WebGL úplne hardvérovo urýchľované priamo v prehliadači. Hlavnou nevýhodou WebGL je obtiažnejšia práca s textom a zatiaľ neexistujúce knižnice pre tvorbu diagramov z ktorých by sme mohli vychádzať.

CSS 3D transformácie taktiež podobne ako WebGL umožňujú vykresľovanie 3D grafiky vo webovom prehliadači avšak iným spôsobom, konkrétne prostredníctvom 3D efektov nad bežnými HTML (DOM) elementami. V porovnaní s WebGL sú elementy ľahšie štýlovateľné, resp. v CSS3D je ich vzhľad definovaný prostredníctvom kaskádových štýlov (CSS), no vo WebGL musí mať každý objekt definovaný materiál a prípadne aj svoju textúru.

WebGL je v porovnaní s CSS 3D "silnejší" z pohľadu rozsiahlosti možností pri tvorení 3D prostredia, avšak pre tvorbu UML diagramov je postačujúce CSS 3D. Veľkou výhodou CSS 3D sú existujúce knižnice pre tvorbu 2D diagramov, ktoré je možné zužitkovať taktiež v 3D prostredí a jednoduchosť spracovania HTML elementov do XMI formátu, prostredníctvom ktorého je možné importovať vytvorené diagramy do nástroja EA, ale taktiež ich aj exportovať z EA a následne spracovať do týchto HTML elementov. Naopak veľkou nevýhodou CSS 3D je obtiažna tvorba 3D objektov (konkrétne šípky) medzi jednotlivými plátnami diagramov.

2.1.1.2 Analýza knižníc na tvorbu 2D diagramov

JointJS

JointJS je knižnica na tvorbu a vykresľovanie rôznych druhov diagramov. Funguje na základe práce s SVG elementami, ktoré ponúka HTML5 jazyk. Knižnica poskytuje MVC (alebo skôr MV) architektúru oddeľujúcu modely grafu, vzťahov a elementov od ich vykresľovania. To uľahčuje pripojiť JointJS k backendu aplikácie. Teda pri práci s jednotlivými elementami, pri ich špecifikácií sa upravuje model nie view. JointJS využíva knižnice JQuery, Underscore, Backbone, a SVG elementy.

Model - View štruktúra je definovaná nasledovne. Diagram (Graph) v JointJS pozostáva z elementov (Element) spojených vzťahmi (Link). Vykresľovacia plocha (Paper) pozostáva z ElementView a LinkView a je v nej definovaný model diagramu (Graph), ktorý upravujeme. V praxi to znamená, že ak chceme na vykresľovaciu plochu vykresliť diagram a jeho elementy, musíme si definovať Paper, určiť mu rozmery, element v HTML, na ktorý sa ma plocha vykresliť (najčastejšie) a hlavne model (Graph), v ktorom definujeme vzťahy a elementy.

JointJS ponúka široké možnosti definovania vlastných typov elementov a vzťahov a taktiež možnosť ukladania modelov elementov na backend. Chýbajú ale vytvorené elementy pre sekvenčný diagram a taktiež overenie diagramu na základe metamodelu. Keďže sa JointJS používa skôr na diagramy podobné diagramom aktivít, diagramu tried a pre sekvenčný diagram platia špecifické pravidlá pre vzťahy medzi lifelinami by sme sa mohli pri použití JointJS stretnúť s množstvom problémov, či už pri vykresľovaní alebo definovaní jednotlivých elementov pre sekvenčný diagram.

GoJS

GoJS je prispôsobiteľná knižnica pre tvorbu interaktívnych diagramov, ktoré sú následne vykresľované prostredníctvom webových prehliadačov a platforiem. GoJS vytvára JavaScriptové diagramy na základe komplexných uzlov (nodes), odkazov (links) a skupín elementov (groups), ktoré sú prispôsobiteľné prostredníctvom šablón a špecifikácií. GoJS používa čistý JavaScript a grafiku renderuje do HTML canvas alebo SVG elementov.

GoJS ponúka rozšírenú funkcionalitu pre používateľskú interakciu, ako je napríklad drag and drop, kopírovanie a vkladanie elementov, priama editácia textu, kontextové menu, automatické rozloženie elementov, šablóny a spracovanie udalostí. Výhodou je, že je táto funkcionalita z veľkej miery prispôsobiteľná taktiež nie je závislá na žiadnych JavaScriptových knižniciach a rámcoch. Ďalšou veľkou výhodou je, že GoJS už obsahuje implementáciu šablón pre základné elementy sekvenčného diagramu, konkrétne čiary života (lifeline), správy (message) a blok akcie (action block). Tieto elementy sú už síce implementované no vytvorenie nových elementov a ich dodatočné prispôsobenie môže byť problematické. Nevýhodou GoJS môže byť taktiež jeho použitie v TypeScripte, keďže základné elementy sekvenčného diagramu sú implementované v čistom JavaScripte, je potrebné kód týchto elementov transformovať do TypeScriptu.

2.1.1.3 Zhodnotenie

Prostredníctvom analýzy vykresľovacích technológií sme dospeli k poznatkom, na základe ktorých sme vytvorili porovnanie WebGL a CSS 3D. Toto porovnanie je zobrazené v tabuľke č.1.

WebGL		**CSS 3D**	
Výhody	Nevýhody	Výhody	Nevýhody
Hardvérovo urýchľované	Problém v práci s textom	Jednoduché selektovanie elementov	Problém s vytvorením šípky medzi plátnami
Možnosť použitia shaderov	Neotvorený zdrojový kód	Zapojenie JS, Jquery	
Možnosť použitia osvetľovania scény	Nižšia podpora ako v prípade CSS 3D	Uloženie do XML (cez html dom)	

Silnejšie ako CSS 3D		Možnosť otáčať celé vrstvy	
Natívne pre Three.js rendering		Existujúce 2D UML knižnice	
Možná podpora pre VR			

Tabuľka: Porovnanie WebGL a CSS 3D

Na základe analýzy knižníc na tvorbu 2D diagramov sme vytvorili nasledujúce provnanie GoJS a JointJS knižníc.

GoJS

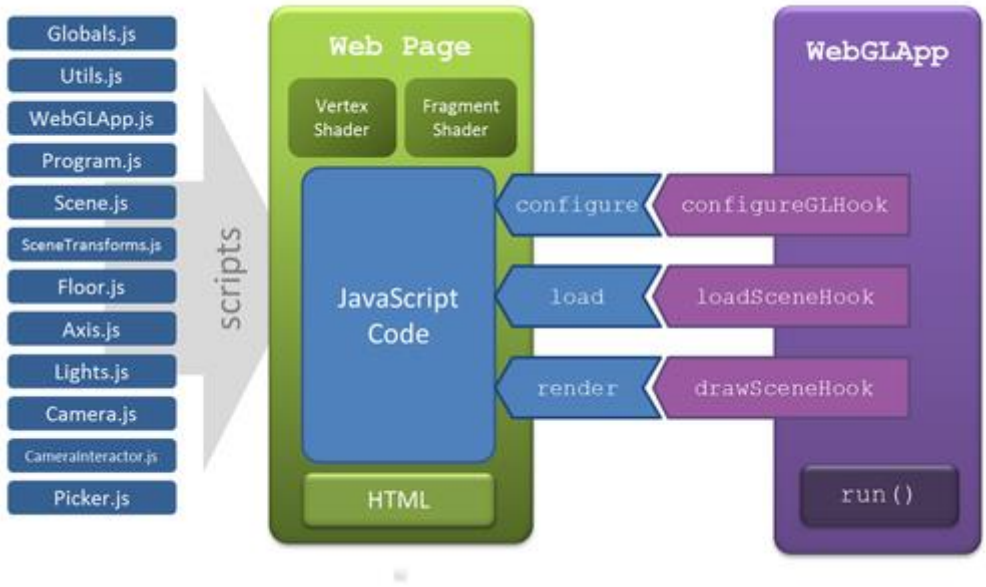
- Výhody:
  - Vytvorené šablóny pre základné elementy sekvenčného diagramu
  - Možnosť prispôsobovať funkcionalitu knižnice
  - Možnosť definovať vlastné elementy a rozširovať existujúce
- Nevýhody
  - Chýba overenie na základe metamodelu sekvenčného diagramu
  - Možné problémy pri prispôsobovaní funkcionality a vytváraní šablón

JointJS

- Výhody:
  - Možnosť definovať vlastné elementy a rozširovať existujúce
  - Jednoduché prepojenie modelov elementov s backendom
  - Prehľadná Model-View štruktúra (pracuje sa s modelmi, view len renderuje)
- Nevýhody
  - Nedefinované elementy sekvenčného diagramu (potreba definovať vlastné)
  - Chýba overenie na základe metamodelu sekvenčného diagramu
  - Možné problémy pri práci s niektorými špecifickými elementami (action block, lifeline) a ich renderovaní

2.1.2 Návrh

Prototyp aplikácie bude pozostávať iba z klientskej časti aplikácie. Tá sa bude implementovať pomocou jazyka Javascript a bude vykresľovaná na technológii WebGL s knižnicou Three.js. Prototyp teda nebude obsahovať žiadne serverové komponenty, žiadnu komunikáciu so serverovou časťou a žiadnu databázu. Na obrázku nižšie vidíme spôsob komunikácie WebGL a samotnej aplikácie.



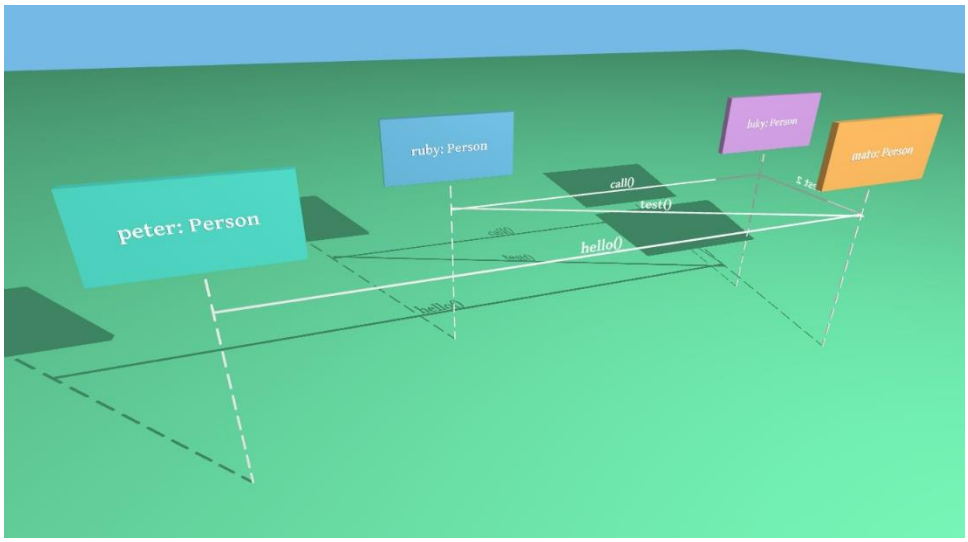
Obrázok: Prepojenie WebGL a stránky

Pre potreby overenia funkčnosti nie je potrebné zaoberať sa správnou a udržateľnou štruktúrou dát, prototyp bude slúžiť iba na overenie funkcionality vykresľovania diagramu jednotlivými knižnicami. Dáta do diagramu budú napevno určené v kóde aplikácie a nebude možné ich meniť z klientskej strany.

2.1.3 Implementácia

2.1.3.1 Prototyp WebGL technológie

Ako rozšírenie analýzy vykresľovacích technológií a knižníc sme sa rozhodli implementovať prototyp aplikácie založenej na technológii WebGL. Cieľom tohto prototypu bolo preukázať možnosti technológie WebGL a knižnice Three.js. Po následnej diskusii a po zvážení všetkých kladov a záporov s ohľadom na udržateľnosť a rozšíriteľnosť aplikácie sme sa však nakoniec rozhodli ďalej aplikáciu s využitím technológie WebGL nevyvíjať. Namiesto nej sme zvolili alternatívnu technológiu CSS3 3D transformácií. Nasledujúci obrázok ilustruje vytvorený prototyp aplikácie pomocou technológie WebGL.

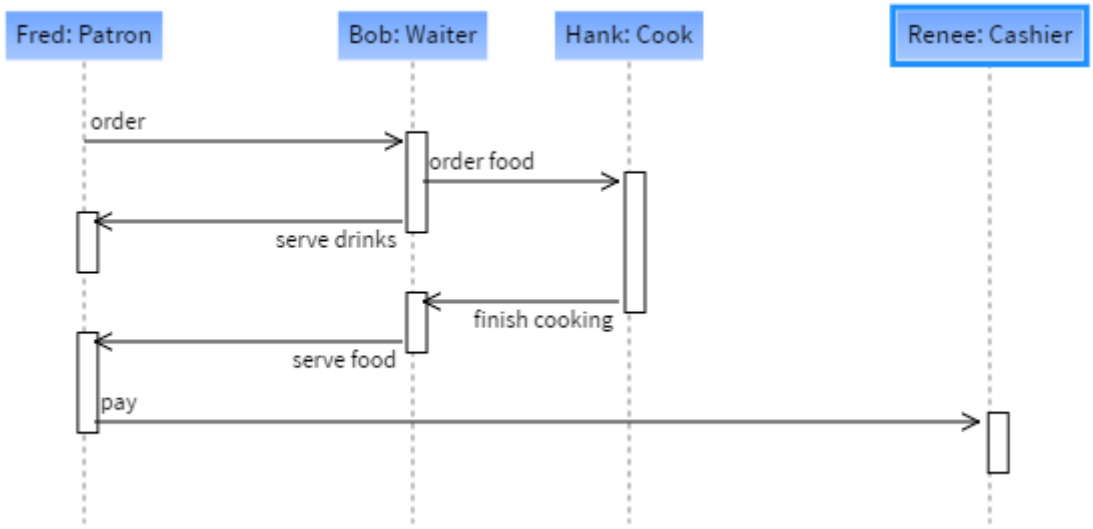


Obrázok: Ukážka WebGL prototypu aplikácie

2.1.3.2 Knižnica na tvorbu 2D diagramov

Na základe vedomostí, ktoré sme nadobudli počas analýzy vykresľovacích technológií sme sa rozhodli do nami vyvíjanej aplikácie integrovať knižnicu GoJS, kvôli jej vhodným vlastnostiam. Knižnica GoJS nám umožňuje vykresľovať UML diagramy do HTML elementu "canvas". Na docielenie 3D efektu UML diagramov je následne možné aplikovať CSS3 3D transformácie. Vďaka nim sú UML diagramy vizuálne zobrazené ako 3D prvky, medzi ktorými je následne možné vykresľovať spojenia.

Tento spôsob práce v kombinácii s viacerými knižnicami a technikami nám umožňuje využiť pozitívne vlastnosti oboch technológií, ako HTML s využitím 2D grafiky v elemente "canvas", tak aj jazyka CSS3 na zavedenie 3D aspektu do aplikácie. Dekomponovanie 3D scény na 2D plátna prináša výhodu z hľadiska možnosti použitia už existujúcich knižníc na tvorbu UML diagramov na jednotlivých plátnach. Na obrázku nižšie môžeme vidieť ako vyzeral sekvenčný diagram v knižnice GoJS.



Obrázok: Príklad diagramu z knižnice GoJS

2.1.4 Testovanie

Po implementovaní GoJS sekvenčných elementov do AngularJS sme dospeli k záverom, že niektorá funkcionalita elementov je správna, ale niektoré elementy sekvenčného diagramu (napr. kombinovaný fragment, action bloky) je potrebné vytvoriť a prispôsobiť.

2.2 Serverová aplikácia

V tejto kapitole opisujeme časť aplikácie, ktorej hlavnou úlohou je perzistencia dát a ich následné poskytovanie pri požiadavke. Ide o vytvorenie aplikačného rozhrania na základe protokolu, ktoré následne poskytuje dáta klientovi.

2.2.1 Analýza

2.2.1.1 Analýza serverových prostredí

PHP

PHP je interpretovaný jazyk na serverovej strane, ktorý pôvodne vznikol na vyvíjanie webových aplikácii, neskôr sa však začal používať aj ako všeobecne použiteľným programovací jazyk.

Tento objektovo orientovaný programovací jazyk je možné použiť spolu s HTML kódom, alebo napríklad aj vrámci frameworku, alebo web content management systému. Funkcie v PHP sú blokované, až do ukončenia, čiže nejaký príkaz sa vykoná až keď bol predchádzajúci príkaz splnený.

NodeJS

Open source, multi platformové Javascriptové prostredie na vytváranie aplikácii. Nie je to Javascript framework, aj napriek tomu, že viaceré moduly sú napísané práve v Javascripte. Má event driven architektúru schopnú asynchrónnych vstupno-výstupných operácii. Tieto vlastnosti majú za úlohu optimalizovať priepustnosť a škálovateľnosť webových aplikácii, zameraných najmä na obrovské množsto I/O operácii, ako sú napríklad hry v prehliadači, alebo komunikácia v reálnom čase.

Aplikácie NodeJS podporujú Mac OS X, Microsoft Windows, NonStop a Unix servery. Alternatívne vedia byť tieto aplikácie vytvorené pomocou CoffeeScript, TypeScript, alebo ľubovoľného iného jazyka skompilovateľného do Javascriptu.

Využitie má podobné ako PHP, teda na tvorbu webových serverov. Hlavný rozdiel je však vo vykonávaní inštrukcií. Funkcie v NodeJS nie sú blokujúce, a teda vedia byť vykonané paralelne, pričom na signalizáciu dokončenia, alebo zlyhania použijú callback.

NodeJS používa Event driven programovanie na vytváranie rýchlych webových serverov. Vďaka tomuto programovaciemu vzoru sú servery v NodeJS vysoko škálovateľne aj bez použitia vlákien.

Zend vs Laravel

Porovnanie v dnešnej dobe populárnych frameworkov na pracú s programovacím jazykom PHP.

Vlastnosť	**Laravel**	**Zend**
licencia	MIT	BSD
databázový model	relačný	objektovo orientovaný
vývojové princípy	Test Driven Development	
	žiadne opakovania	
	konfigurácia pred pohodlnosťou	konfigurácia pred pohodlnosťou
programovacie paradigmy	Event driven	Event driven
	objektovo orientované	objektovo orientované
	funkcionálne	
podpora skriptovacích jazykov	PHP, Javascript	PHP
znoupoužitelnosť kódu	Model View Controller	Model View Controller
	webové služby	
podpora Maven	nie	áno
pripojená Javascript knižnica	nie	áno
čiastočné triedy	áno	nie
návrhové vzory	Active record	Active record
	Model View Controller	Model View Controller
	Dependency injection	Dependency injection
	Event driven	Event driven
	Observer	
	Facade	

		Data mapper
databázy	Sqlite, Mysql, Postgresql, Redis, Microsoft Bi	Sqlite, Mysql, Oracle, Postrgesql, Microsoft Bi, Mongodb, Mariadb

2.2.1.2 Analýza komunikačných protokolov

JSON API

Zadefinováva štruktúru, akou bude JSON formátovaný. Tým, že je presne zadefinovaná forma súboru JSON sa zvyšuje produktivita a prehľadnosť kódu.

Okrem štruktúry dokumentu definuje aj správanie klienta a serveru. Napríklad klient musí posielať všetky JSON API dáta v request dokumente s hlavičkou :

```
Content-Type: application/vnd.api+json
```

a neobsahujú žiadne parametre typu média. Pri odpovedi zase musí ignorovať všetky parametre typu média, ktoré sú v hlavičke súboru. Server má obdobne zadefinované správanie pri komunikácii s klientom.

Jednoduchý príklad :

```
{ "data": { "type": "articles", "id": "1", "attributes": { // ... this article's attributes }, "relationships": { // ... this article's relationships } } }
```

JSON RPC

V JSON formáte zakódovaný protokol na vzdialené volanie procedúr (podobný ako XML-RPC pre formát štandardu XML). Jednoduchý protokol, ktorý definuje iba základné dátové typy a príkazy. Podporuje posielanie viacerých dopytov na server, ktoré nemusia byť spracované v poradí, v akom dané dopyty odišli. Okrem toho ponúka aj možnosť jednosmernej komunikácie formou notifikácii, kedy serveru klient odošle dáta, na ktoré nepotrebuje odpoveď.

Funguje na princípe posielania dopytov na server, kde je tento protokol implementovaný. Podporuje možnosť viacerých vstupov ako parameter volania, kedy sú tieto vstupy uložené ako pole, alebo ako objekt. Metóda takisto môže vrátiť viacero hodnôt na výstupe.

Všetky transférové typy sú objekty serializované pomocou JSON. Každé volanie musí mať následne tri vlastnosti :

- Method – meno metódy, ktorú voláme
- Params – objekt alebo pole hodnôt, ktoré sa majú predať metóde ako parameter
- Id – hodnota určená na spárovanie dopytu a odpovede

Server zase musí odpovedať valídnymi odpoveďami, ktoré obsahujú tieto vlastnosti:

- Result – vrátené dáta ako výstup metódy, null ak prišlo ku chybe vo vykonávaní metódy
- Error – číslo chybového hlásenia, ak ku chybe nedošlo nadobúda hodnotu null
- Id – identifikačné číslo dopytu na ktorý server odpovedá

Notifikácie majú podobnú štruktúru, akurát je vyhodené, alebo nastavené na hodnotu null pole id, ktoré nie je potrebné, keďže neposielame odpoveď.

XML

Extensible Markup Language, ktorý definuje súbor pravidiel pre štruktúru dokumentu tak, aby boli zrozumiteľné aj pre ľudí, aj pre počítače. Obdobne ako JSON je to teda ideálny formát dát na výmenu medzi serverom a klientom.

Svojou syntaktickou skladbou pripomína HTML, kde sú tiež všetky informácie obkolesené takzvanými tagmi. Tieto tagy určujú informáciu akého typu sa jedná a samotnú informáciu. Existujú tri druhy tagov:

- Štartovacie - <section>
- Ukončovacie - </section>
- S prázdnym elementom -

Element je potom informácia uchovávaná medzi štartovacím a ukončovacím tagom. V týchto tagoch ďalej môžu byť vnorené atribúty, ktorými bližšie definujeme daný element. Môže ísť o doplňujúcu informáciu, akými sú napríklad vek, farba, ...

Štandard ďalej definuje, akú hlavičku musí obsahovať XML súbor, aby bol valídny, špecifikuje aké znaky sa môžu vyskytovať pri menách elementov a oveľa viac.

Porovnanie XML a JSON

JSON

Výhody:

- Jednoduchá syntax
- Jednoduché na použitie s Javascriptom, keďže obsahuje rovnaké základné dátové typy
- Vieme použiť schému aj JSON API na presné dodefinovanie štruktúry
- JsonPath na extrahovanie informácii v hlboko vnorených štruktúrach

Nevýhody:

- Jednoduchá syntax, iba základné dátové typy sú podporované

XML

Výhody:

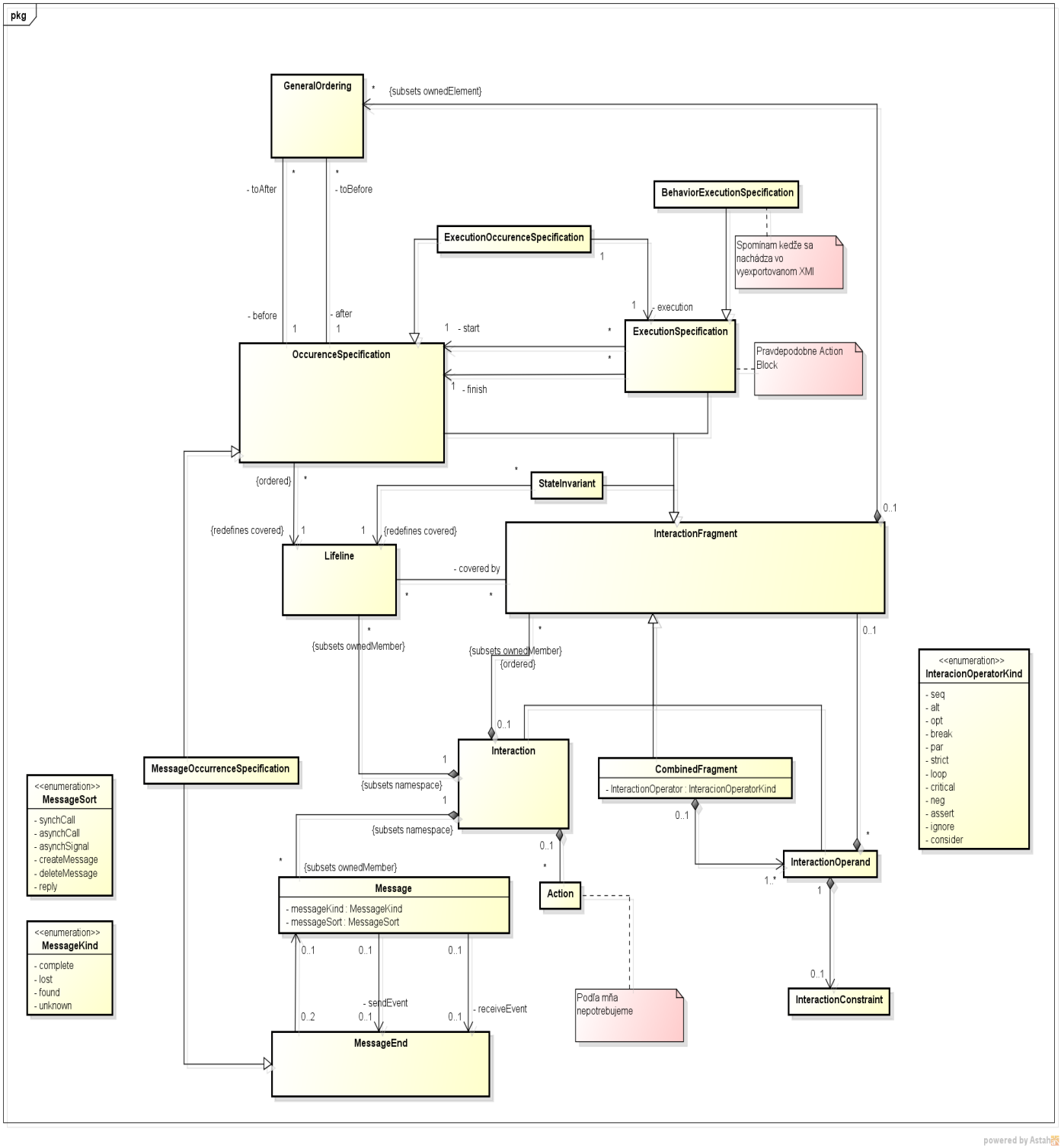
- XML schéma na validáciu štruktúry, alebo tvorbu vlastných dátových typov
- Možnosť transformácie na rozdielne výstupné formáty pomocou XSLT
- XPath na extrahovanie informácii v hlboko vnorených štruktúrach
- Vstavaná podpora pre namespace

Nevýhody:

- Obsahuje viac pomocných informácii, čím sa stáva súbor väčší, a teda potrebujeme preniesť viac dát, aby sme dostali rovnaké množstvo informácii

2.2.1.3 Metamodel sekvenčného diagramu

Metamodel sekvenčného diagramu v UML predstavuje štruktúru prepojení všetkých elementov v sekvenčnom diagrame. Práve z tohto metamodelu sme vychádzali, a to preto, aby sme dokázali správne určiť vzťahy medzi elementami a následne ich použiť pri práci a manipulácii s nimi na scéne.



Obrázok: Metamodel sekvenčného diagramu

Dôležité časti metamodelu sú:

- **Message** - správa medzi lifeline-ami, ktorá môže byť v stave complete, lost, found alebo unknown. Táto správa má taktiež aj typ a tieto typy sú: synchrónne volanie, asynchrónne volanie, vytvorenie, vymazanie, odpovedanie
- **Interaction** – predstavuje akoby celý sekvenčný diagram (napr. Diagram s tromi lifeline a medzi nimi 6 message tvorí jednu interakciu)
- **Lifeline –** predstavuje čiaru života v sekvenčnom diagrame. Obsahuje rolu alebo objektové inštancie, ktoré sa zúčastňujú v sekvenčnom diagrame
- **CombinedFragment** - kombinovaný fragment, ktorý môže byť napr.: alt - alternatíva, loop - cyklus... Takýto fragment môže obsahovať inštanciu samého seba (to znamená, že sú v ňom vnorené ďalšie iné combined fragmenty)
- **InteractionOperand –** je operand interakcie (napr. A + B , tak 'A', 'B' sú operandy a '+' je operátor)
- **InteractionConstraint –** je podmienka, ktorá ak platí, tak sa vykoná daný typ operátora
- **GeneralOrdering** – poradie výskytov daných elementov. Navrhnutý ako spájaný zoznam výskytov. To znamená že každý prvok v rámci lifeliney vie, ktorý prvok ide za ním.
- **ExecutionOccurrence** – predstavuje action block
- **OccurenceSpecification** – táto trieda značí výskyt. Hovorí o tom kde sa element začína a kde sa končí. Dedia od nej ďalšie dve triedy, ktoré konkretizujú výskyt správy (MessageOccurrenceSpecification) a action bloku (ExecutionOccurrenceSpecification).

2.2.1.4 Zhodnotenie

Pri výbere serverového programovacieho jazyka boli analyzované hlavne PHP a Node.js. PHP je samostatný programovací jazyk, zatiaľ čo Node.js je postavené na Javascripte a poskytuje funkcionalitu Javascriptu aj na strane servera. PHP je starší, zaužívaný jazyk, ktorý podporuje všetky dostupné platformy operačných systémov. Za dlhé obdobie jeho používania vzniklo viacero frameworkov, ktoré pridávajú, prípadne upravujú funkcionalitu čistého PHP jazyka. Node.js má hlavnú výhodu, že nemusí byť postavený iba na Javascripte, ale aj nad CoffeeScriptom, Typescriptom, alebo ľubovoľným iným jazykom, ktorý je možné skompilovať do Javascriptu.

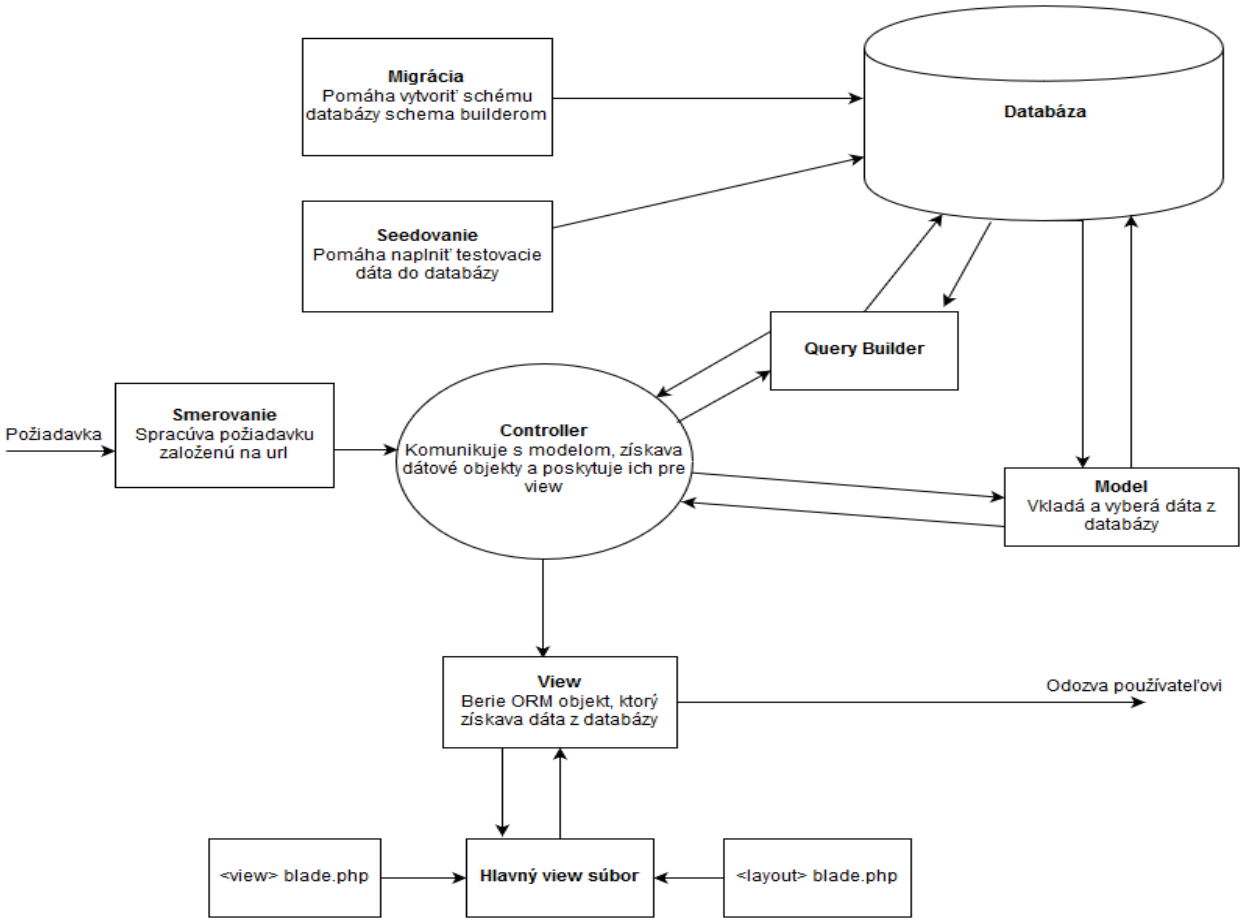
Dostupné frameworky na PHP sú napríklad Laravel, alebo Zend. Oba frameworky poskutujú Model View Controller na implementovanie používateľských rozhraní. Laravel podporuje okrem jazyka PHP aj Javascript a je teda univerzálnejším. Líšia sa možnými použiteľnými databázami, alebo aj podporou Maven.

Metamodel tak, ako je vyššie zadefinovaný bude použitý v ďalších fázach vývoja aplikácie, kedy z neho bude vychádzať napríklad fyzický model databázy, ale aj jednotlivé triedy, ktoré potrebujeme vytvoriť na správnu prácu s dátami.

Komunikačné protokoly sú dva hlavné (XML a JSON). XML definuje štruktúru predávania informácii v zrozumiteľnom tvare aj pre stroje, aj pre človeka. Štruktúra sa skladá z tagov, v ktorých sú vložené informácie. Tento jazyk však obsahuje viacero pomocných informácii, ktoré zvyšujú potrebné množstvo dát na prenesenie. Oproti tomu je JSON API, ktoré presne definuje štruktúru a konvencie pri vytváraní JSON súboru. Je jednoduchší, ľahší a vďaka štandardizácii aj dobre čitateľný a použiteľný.

2.2.2 Návrh

2.2.2.1 Architektúra serverovej aplikácie



Obrázok: Dekompozícia aplikácie backendu

Na obrázku vyššie je znázornená architektúra, ktorú používame pri vývoji naše aplikácie. Odvíja sa od návrhového vzoru MVC (model-view-controller). Takto sme schopní oddeliť aplikačnú logiku od zobrazovania komponentov. Ktokoľvek bude teda schopný zmeniť stratégiu vykresľovania diagramu a nebude nutné meniť dátový model. Popri architektonickom vzore na obrázku vyššie využívame aj iné, už vopred vyvinuté a overené programátorské prístupy.

ORM

Využívame mapovaciu techniku ORM, ktorou zaistíme synchronizáciu medzi použitými objektami v aplikácii tak, aby bola zaistená perzistencia dát. Taktiež je použitá aj z dôvodu, že často redukuje množstvo kódu, ktorý je potrebný písať.

Lazy Loading

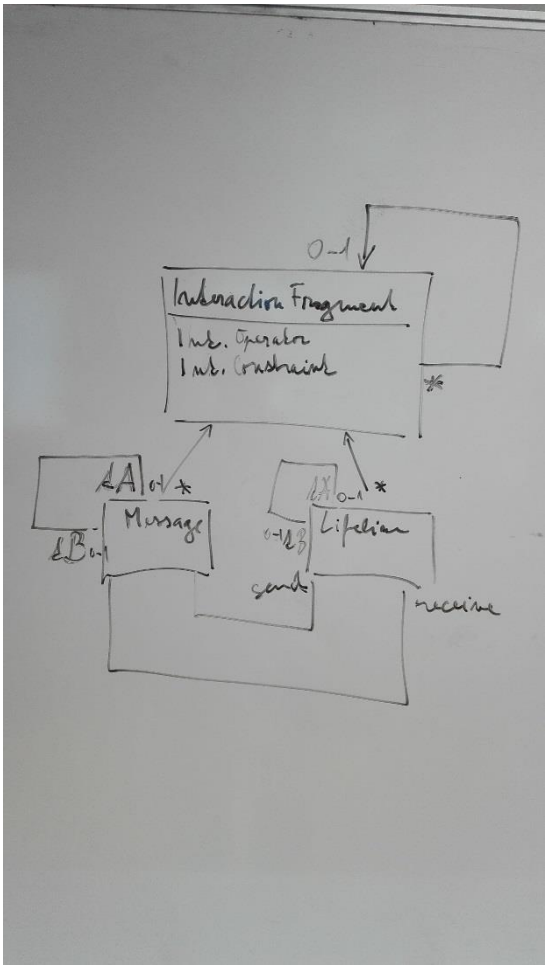
Jeho využitie je dôležité z toho hľadiska, že nemusíme načítať do pamäte všetky dáta z databázy, ale vždy len práve tie, ktoré potrebujeme. Tento prístup je vhodný, keď používame viacero tabuliek v databáze a potrebujeme sa dopytovať do nich naraz a nie vtedy ak máme všetky údaje v jednej tabuľke.

Dependency injection

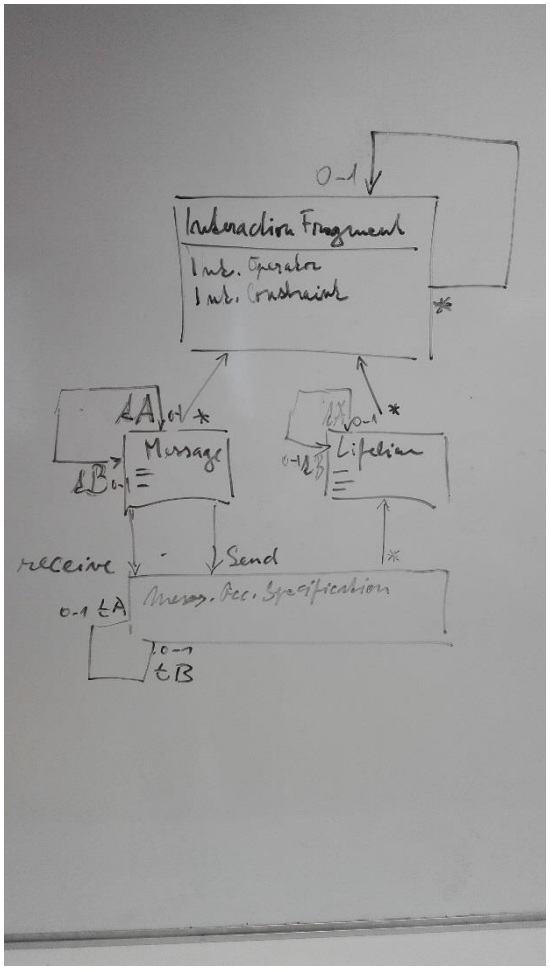
Aby sme udržali kód čo najčistejší a zbavený závislostí, využívame taktiež dependency injection. Zaručíme tým teda dekompozíciu aplikácie a taktiež zjednodušíme testovanie.

2.2.2.2 Fyzický dátový model

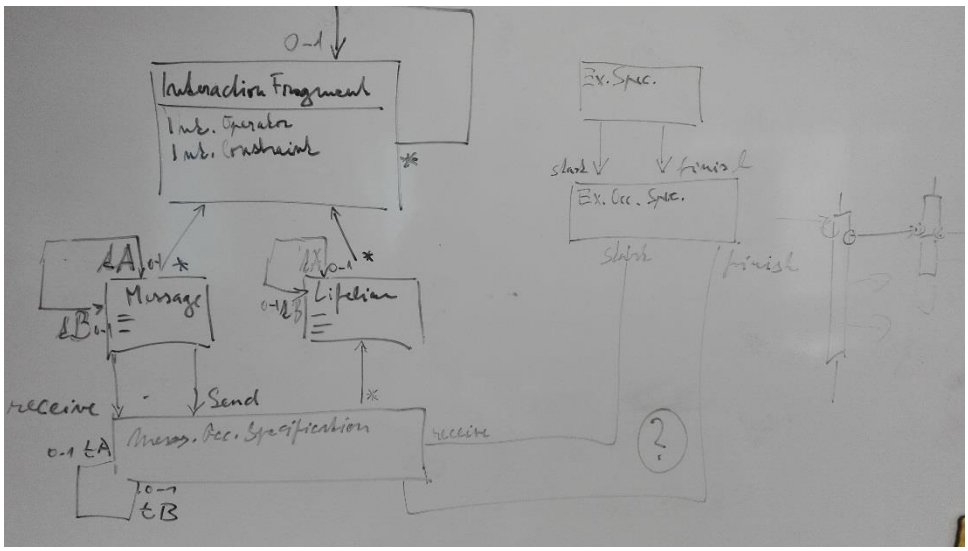
V nižšie uvedenom diagrame sa nachádza fyzický dátový model. Na obrázkoch nad diagramom sa nachádza postup ako sme ho vypracovávali v spolupráci s naším vedúcim. Pri jeho vytváraní sme sa držali metamodelu z UML superstructure. Jednotlivé triedy reprezentujú dané elementy v superstructure a vzťahy medzi nimi.



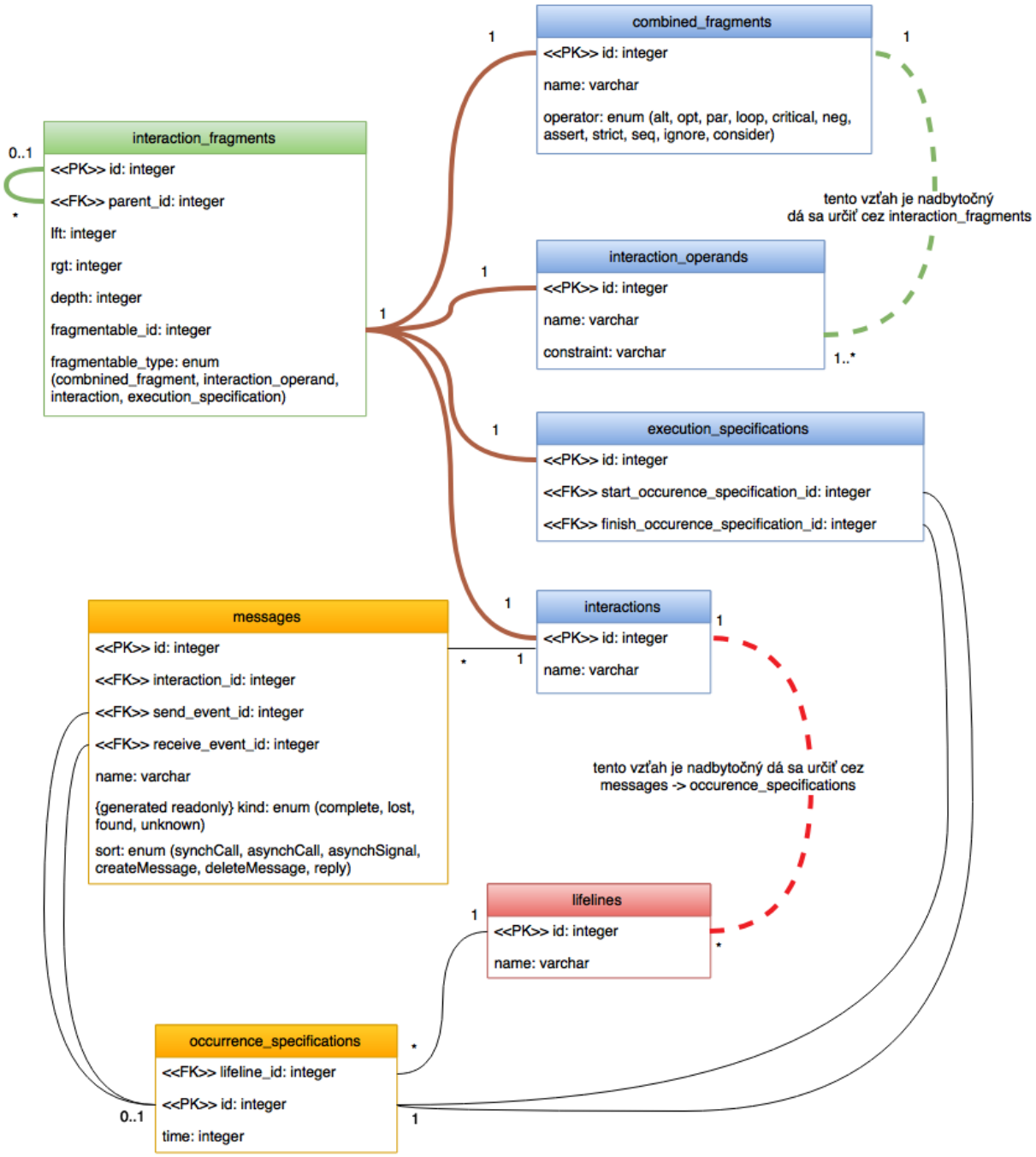
1.krok - identifikácia tried



2. krok – doplnenie vzťahov a entity



Obr.: Postup vytvárania fyzického dátového modelu



Obrázok: Fyzický model

- **interaction_fragments** - abstraktná trieda reprezentujúca fragment v sekvenčnom diagrame. Vďaka nej sme schopní vnímať sekvenčný diagram ako kompozitnú štruktúru. Všetky fragmenty diagramu majú vzťah s touto tabuľkou 1 k 1. Následne teda každý fragment môže mať vzťah samého so sebou a diagram sa teda chová ako kompozit.
- **combined_fragments** - kombinovaný fragment, ktorý v sebe môže niesť ďalšie kombinované fragmenty. Vo fyzickom modely riešime túto skutočnosť tak, že sa odkážeme cudzím kľúčom v interaction_fragmente na samého seba. Tým pádom sa vieme rekurzívne vnárať do podfragmentov. Kombinované fragmenty sú typu alt, opt. par a pod.
- **interaction_operands** – predstavuje operand vo fragmente. Jeden operand môže v sebe obsahovať aj iné operandy, a taktiež viacero kombinovaných fragmentov.

- **interactions –** interakcia, ktorá predstavuje celý sekvenčný diagram
- **messages –** predstavujú správy medzi jednotlivými lifeline-ami. Obsahuje cudzí kľúč na interakciu, keďže správa existuje len v rámci interakcii a v jednej interakcii môže byť viacero správ. Správa má niekoľko typov ako: complete, lost, found, unknown a taktiež niekoľko rozdelení: synchrónna, asynchrónna, vytvorená, vymazaná, alebo ako odpoveď. Viacero správ prislúcha iba jednej lifeline-ne. Cudzie kľúče send_event_id a receive_event_id predstavujú správu, ktorá je odoslaná a správu, ktorá je prijatá.
- **lifelines -** znázorňuje čiaru života. Každá čiara života má id interakcie, do ktorej patrí ako cudzí kľúč. Má teda vzťah s interakciou viac ku 1. Taktiež má vzťah so správami, ktoré na čiare života začínajú alebo končia. Teda vzťah je vyjadrený ako 1 ku viac. Je to dvojnásobne zastúpený vzťah keďže sa zvlášť zameriava na začiatok správy a zvlášť na jej koniec.

2.2.3 Implementácia

V tejto kapitole popisujeme programové prostredie, databázu a protokol komunikačného rozhrania, ktoré sme sa rozhodli využiť pri realizovaní implementácie aplikácie.

2.2.3.1 Programové prostredie

Ako programovací jazyk serverovej aplikácie sme si zvolili jazyk PHP. Tento jazyk nám v kombinácii s frameworkom Laravel poskytol všetky požadované vlastnosti na serverové prostredie nami vyvíjanej aplikácie. Architektúra frameworku Laravel je založená na kombinácií viacerých návrhových a architektonických vzorov. Vďaka tomu je možné aplikáciu jednoduchým a prehľadným spôsobom dekomponovať a organizovať na samostatné moduly.

Základným stavebným vzorom je MVC (Model-View-Controller). Ten nám poskytuje možnosť oddelenia biznis logiky od zobrazení a vrstvy dát. Vrstva dát obsahuje dátové modely. Framework Laravel tiež obsahuje implementovaný návrhový vzor Dependency Injection, ktorý tiež napomáha pri dekomponovaní a testovaní aplikácie. Aplikácia samotná je implementovaná ako IoC (Inverse of Control) kontajner. Injekcia závislostí jednotlivých modulov je následne realizovaná naprieč všetkými časťami frameworku.

Framework Laravel naviac obsahuje funkcionalitu, ktorá nám uľahčií prácu s databázou. Ide napríklad o systém databázových migrácií spolu s naplnením databázy testovacími údajmi, ktoré nezávisia na použitom type databázy ani na príslušnom SQL dialekte. Rovnako je obsiahnutý aj objektovo relačný mapovač Eloquent, ktorý umožňuje mapovanie dát na dátové modely aplikácie.

2.2.3.2 Perzistencia údajov

Je potrebné, aby nami vytvorená aplikácia bola schopná dlhodobej perzistencie UML diagramov. S ohľadom na budúcu možnosť importovania a exportovania UML diagramov do zaužívaných formátov sme sa rozhodli údaje o UML diagramoch taktiež ukladať aj do relačnej databázy.

Zvolili sme si databázu PostgreSQL kvôli jej kvalite, rozšírenosti a otvorenosti. Ukladanie UML diagramov do databázy vo forme fyzického dátového modelu nám tiež umožňuje automatické overovanie ukladaných dát voči metamodelu.

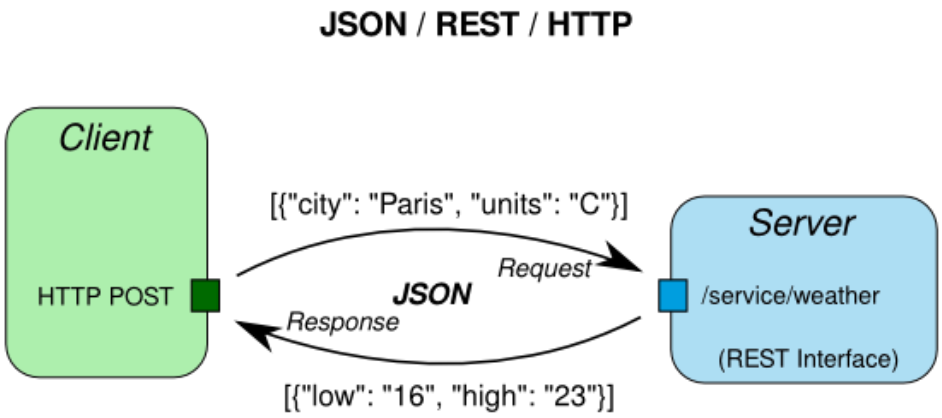
Prepojenie medzi databázou PostgreSQL a frameworkom Laravel je realizované pomocou ORM mapovača Eloquent. Tento mapovač obsahuje aj implementáciu užitočných návrhových vzorov ako napríklad lazy loading, čím je dosiahnutý vyšší výkon pri prístupe k údajom uloženým v databáze.

V ostatných častiach aplikácie je možné pracovať s dátovými entitami ako s objektami bežných tried, ktoré sa nazývajú dátové modely. Dátové modely obsahujú relácie v podobe metód, ktoré umožňujú prácu s príslušnými väzbami na iné dátové modely.

2.2.3.3 Protokol komunikačného rozhrania

Aby sme vedeli zabezpečiť výmenu informácií medzi serverovou časťou aplikácie a klientami potrebujeme presne stanovený formát a pravidlá. Z analýzy možných formátov sme si ako formát na výmenu dát vybrali JSON, najmä pre jeho rozšírenosť, jednoduchosť použitia a interoperabilitu.

Špecifikácia JSON API je sada generických konvencií formátovania požiadaviek a odpovedí na zdroje z angl. resources. Jeho hlavnou výhodou je definovanie všeobecných pravidiel, ktoré možno použiť na akúkoľvek doménu riešenia. Špecifikácia neobsahuje len pravidlá na dopytovanie na zdroje ale taktiež aj na ich aktualizáciu, či mazanie. Obsahuje nielen atribúty zdroja ale taktiež aj jeho vzťahy na iné zdroje spolu s odkazmi na dané zdroje. Umožňuje aj filtrovanie výsledkov či stránkovanie. Ďalšou významnou výhodou je jeho štandardizácia a teda možnosť použitia knižnice do zvoleného jazyka, ktorá spĺňa špecifikácu. To nám umožňuje jednoducho implementovať aplikačné rozhranie a toto rozhranie konzumovať klientami opäť použitím knižnice na strane klienta pre transparentnú prácu nad zdrojmi. Nasledujúci obrázok zobrazuje komunikáciu medzi klientskou aplikáciou so serverom pomocou špecifikácie JSON API.



Obrázok: Komunikácie klientskej aplikácie so serverom pomocou JSON API

2.2.4 Testovanie

Testovanie funkcionality migrácií v Laraveli prebehlo úspešne. Dané tabuľky boli v databáze vytvorené. Taktiež prepojenie JSON API medzi frontendom a backendom prebehlo úspešne na základe manuálnych testov porovnania požadovaných výsledkov.

2.3 Jadro klientskej aplikácie

Nasledujúca kapitola opisuje aplikáciu, ktorá beží v prehliadači a zabezpečuje celkovú interakciu s používateľom aplikácie.

2.3.1 Analýza

2.3.1.1 Analýza klientských prostredí

Typescript

Je striktným supersetom Javascriptu, ktorý pridáva statické písanie a objektovo orientované programovanie do jazyka. Vieme ho použiť na vytváranie klientských aplikácii, ale aj servorových založených na NodeJS.

Typescript je navrhnutý na vývoj veľkých aplikácii a ich následnej kompilácie do Javascriptu. Keďže je Typescript supersetom Javascriptu, všetky existujúce Javascript programy sú kompatibilné aj s Typescript programami.

Typescript obohacuje Javascript napríklad pridaním anotácii, interface, enum typov, namespace,...

Výhody:

- Superset Javascriptu, čiže vie pracovať aj s budúcimi vlastnosťami pridanými do Javascriptu
- Odhalenie chybových hlásení priamo pri kompilácii do Javascriptu
- Dedenie, interface a ďalšie OOP štandardy
- Syntax podobná pre Java, alebo C# vývojárov
- Využívané napríklad aj Angular frameworkmi

Nevýhody:

- Náročnejšie pre začiatočníkov objektovo orientovaného programovania
- Nutnosť použiť Typescript definície aj pre knižnice, ktoré nie sú v Typescripte
- Vyžaduje najskôr skompilovanie kódu, nevieme teda kód priamo spustiť
- Náročnejšie vytváranie, buildovanie, testovanie projektu, hlavne keď viaceré automatizačné nástroje ešte nepodporujú úplne Typescript

Javascript

Vysoko úrovňový, dynamický a interpretovaný programovací jazyk, ktorý bol štandardizovaný vrámci ECMAScript špecifikácie. Javascript nie je úplne procedurálny jazyk, má toho veľa podobného aj s funkcionálnymi jazykmi, ako napr. Lisp. Obsahuje aj viacero funkcionalít, ktoré pripomínajú objektovo orientované programovanie, ale v skutočnosti neobsahuje triedy.

Výhody:

- Jednoduchý jazyk
- Vykonáva sa na klientovej strane, čím šetrí záťaž servera
- Rozšírená funkcionalita na webových stránkach, vrátane širokej možnosti podporovaných prehliadačov
- Rýchle vykonávanie kódu
- Má prístup k DOM štruktúre

Nevýhody

- Bezpečnostné problémy, kedy takýto kód sa vykonáva u klienta a môže obsahovať škodlivý kód
- Vykreslenie závisí od jednotlivých prehliadačov

Porovnanie Angular 1 a Angular 2

Angular 2 je nová verzia frameworku Angular, ktorá sa od predchádzajúcej líši najmä tým, že už nepracuje s Javascriptom, ale Typescriptom. V prípade, že projekt bude postavený na Typescripte, Angular 1 nie je použiteľný a je treba prejsť na Angular 2. Hlavné zmeny, ktoré nastali vo frameworku s príchodom verzie 2:

- Využíva Hierarchical Dependency Injection systém na zvýšenie výkonu
- Širšia podpora jazykov, vrátanie ES5, ES6, Typescript, Dart
- Webové štandardy sú implementované ako komponenty
- Komplikovanejšie nasadenie Angular 2
- Controller je nahradený komponentom
- Rozdielny spôsob definície lokálnych premenných
- Použitie camelCase na vstavané príkazy
- Priamo používa valídne HTML DOM vlastnosti a udalosti elementov
- Zmena syntaxe pri Dependency Injection

Knockout.js

Samostatná Javascript implementácia Model-View-View model patternu. Je to teda webový framework, v ktorom sa ale ťažšie definuje infraštruktúra, resp. musí si ju používateľ zadefinovať sám. To môže tvoriť problémy v neskorších fázach vývoja aplikácie hlavne v prípade, bola štruktúra zle vytvorená. Využíva sa hlavne na kontrolu reprezentácie UI rozhrania pri menej komplexných aplikáciach. Nepodporuje natívne routing, ale dá sa externe pripojiť pomocou ďalších webových frameworkov.

Ďalšie vlastnosti:

- Dependency tracking
- Jednoduché asociovanie DOM elementov s modelovými dátami
- Pracuje na čistom Javascripte
- Deklaratívne väzby
- Jasné oddelenie doménových dát, viewov a dát na zobrazenie
- Data binding

2.3.1.2 Analýza komunikačných knižníc

Na implementáciu komunikácie klientskej aplikácie so serverom, kde rozhranie je implementované pomocou špecifikácie JSON API, potrebujeme knižnicu, ktorá dokáže na základe tejto špecifikácie vymieňať informácie so serverovou časťou. Do úvahy vzhľadom na výber frameworku Angular2 pripadajú dve alternatívy:

angular2-jsonapi

Knižnica jednoducho poskytuje volania na server pričom odpovede priamo mapuje na modely (DAO) triedy vo frameworku Angular 2. Medzi výhody patrí najmä jednoduché použitie v projekte. Ďalej poskytuje expresívnu definíciu JSON API modelov. Implementuje všetky základné časti špecifikácie a teda je možné nielen dopytovanie, ale aj aktualizácia a mazanie.

ngrx-json-api

Knižnica je rozšírením knižnice ngrx (Reactive Extensions for Angular), ktorá slúži na komunikáciu s aplikačnými rozhraniami a je inšpirovaná frameworkom React. Implementuje taktiež špecifikáciu JSON API, no neposkytuje tak expresívne a jednoduché rozhranie. Definícia modelov je len na úrovni opisu zdroja.

2.3.1.3 Analýza 3D grafických knižníc

Three.js

Open source projekt Javascriptovej knižnice zameraný na vytvorenie WebGL frameworku, ktorý je v súčasnosti aj jedným z najpoužívanejších. Vytvára sa ako ďalšia vrstva nad WebGL, ale zároveň umožňuje aj vykreslenie priamo na canvas pri zariadeniach, ktoré nepodporujú WebGL. Knižnica je dostatočne nízko úrovňová, aby ponúkala možnosti zmeny a úpravy kódu, ale zase nie je potrebné starať sa o komunikáciu s prehliadačom. Ponúka možnosť vytvorenia 3D animácii, ktoré sú akcelerované pomocou GPU.

Obsahuje obsiahlu dokumentáciu a príklady, ale jeho nevýhodou môže byť väčšia spotreba prostriedkov. Preto je vhodná pre menšie projekty, kde nároky na prostriedky nebudú až také vysoké. Knižnica má problémy so spätnou kompatibilitou. V prípade, že vyjde novšia verzia knižnice, staré demá, návody a kusy kódu môžu prestať fungovať.

Vlastnosti:

- Nízko úrovňový API
- Časovo náročnejší na vytvorenie základných vecí
- Možnosť vytvorenia vlastnej logiky a vlastných komponentov
- Nemožnosť exportu do 3D modelovacieho programu
- Dodávané ako minimalistická verzia, všetko ostatné je potrebné doprogramovať

Scene.js

Jednoduchá knižnica, ktorá sa špecializuje na rýchle vykresľovanie obrovského množstva objektov, bez herných efektov, ako sú odrazy alebo tiene. Ponúka menej flexibility, nakoľko je zameraná hlavne na CAD, medicínsku anatómiu, inžinierske vizualizácie a iné.

Babylon.js

Rovnako ako knižnica Three.js, aj Babylon.js je knižnica založená na jazyku Javascript a ponúka rozšírenie WebGL. zameraná hlavne na menšie projekty. Podporuje možnosti práce s grafikou, ako sú napríklad osvetlenie, kamery, materiály, alebo fyzikálne prostredie, ktoré má svoje využitie hlavne pri vyvíjaní hier. Pri jednoduchej grafickej aplikácii, akou je 3D UML nie je potrebné riešiť textúry, alebo fyziku.

Vlastnosti:

- Vyššie úrovňové API
- Menšia flexibilita pri vytváraní vlastných modulov
- Export do viacerých 3D programov
- Obsahuje integrovanú logiku pre hry
- Náročná dekompozícia

2.3.1.4 Zhodnotenie

Pri výbere technológie na klientské prostredia prichádzali do úvahy technológie Javascript, alebo Typescript. Typescript je novší, ako Javascript, čo znamená, že je pre neho menej návodov a dostupného kódu, ale obsahuje takisto vyladené chyby z Javascriptu. Hlavný rozdiel je, že Typescript sa musí pred spustením skompilovať na Javascript. Ponúka však lepšiu prácu s objektami, prehľadnejšiu štruktúru a lepšiu prácu s objektami.

Poďla toho, či zvolíme Typescript alebo Javascript sú k dispozícii viaceré frameworky na prácu s daným programovacím jazykom. V prípade, že by sme zvolili Typescipt je najvhodnejšie použiť Angular 2, ktorý bol vytvorený priamo pre Typescript. Narozdiel od neho, Angular 1, alebo Knockout.js pracujú nad čistým Javascriptom. Hlavná nevýhoda Knockout.js je nedefinovaná štruktúra projektu a preto sa využíva hlavne pri menej zložitých aplikáciách a na kontrolu reprezentácie UI rozhrania.

V prípade 3D knižníc bolo rozdielne ich zameranie. Scene.js bol vhodnejší na CAD projekty, alebo inžiniersku vizualizáciu, preto neobsahoval zložitejšie možnosti. Babylon.js zase na druhej strane toho obsahoval príliš veľa, od tieňov až po fyziku, ktoré sú vhodné najmä pri vytváraní hier. Menšie úpravy a zásahy do preddefinovaných metód boli náročné. Three.js je nízko úrovňová knižnica, ktorá slúži na všeobecné účely, keďže je možné všetko potrebné do nej doimplementovať.

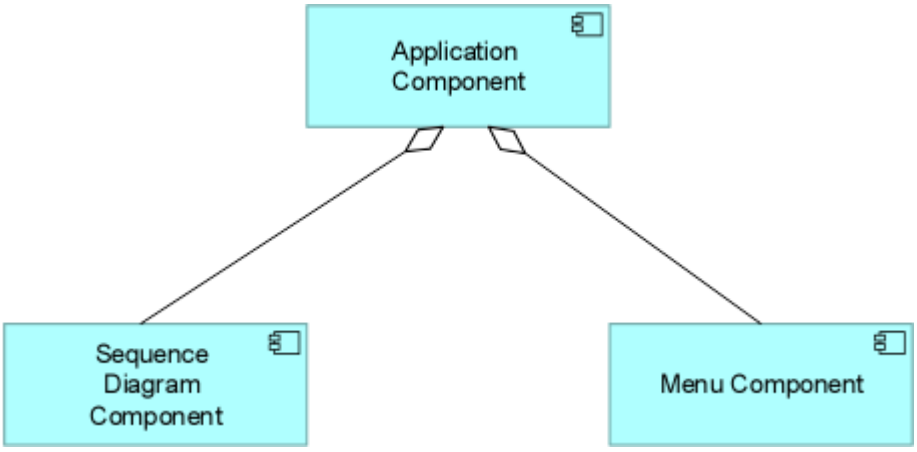
2.3.2 Návrh

2.3.2.1 Architektúra klientskej aplikácie

Aby správne fungovala celá aplikácia je potrebné, aby spolu komunikovali klientská a serverová časť aplikácie. Táto komunikácia prebieha pomocou JSON API, ktoré spája Javascript, resp. Typescript v podobe na klientskej strane s frameworkom na serverovej strane. Existuje viacero knižníc, ktoré implementujú JSON API do Angularu, napríklad Angular 2 JSON API.

Angular 2 JSON API je jednoduchá knižnica, ktorá interaguje priamo s triedami a modelmi, nie čistým JSONom. Umožňuje tak mapovať všetky dáta do modelov a vzťahov. V prvom kroku musíme pridať JsonApiModule do importov, aby sme boli schopní knižnicu používať. Keď je knižnica pripravená, vytvoríme si vlastné modely, v ktorých zadefinujeme štruktúru dát, ktoré budeme pomocou JSON posielať.

Na obrázku nižšie vidíme dekompozíciu Angular knižnice na jednotlivé komponenty. Celá aplikácia je Application Component, ktorá v sebe obsahuje dva ďalšie komponenty. Sequence Diagram Component, ktorého úlohou je zabezpečiť chod klientskej strany aplikácie a Menu Component, ktorý bude obstarávať menu položky, kde bude mat používateľ možnosti práce s aplikáciou.



Obrázok Diagram komponentov v architektúre

2.3.2.2 Komunikácia so serverom

Na komunikáciu so serverom využívame triedu datastore, na ktorej vieme volať metódy. Tieto metódy na základe vopred vytvorených modelov vedia, aké typy informácii majú aplikácii poskytnúť ako odpoveď na dopyt.

Použijeme DAO (Data access object) objekty, ktoré nám poskytnú interface ku databáze, resp. ku dátam. Aplikačné volania sú namapované na perzistentnú vrstvu, kde DAO poskytuje niektoré špecifické dátové operácie bez odhalenia detailov databázy. Je takto zabezpečené oddelenie dát, ktoré potrebuje aplikácia od samotnej implementácie databázovej schémy. Z takto navrhnutej schémy nie je jasné, aké dáta sa v databáze nachádzajú, ani ako databáza vyzerá, vieme iba s akými dátami chceme pracovať. Keby sa zmenila databáza, je potrebné prerobiť modely, nemusíme však meniť metódy na klientskej strane aplikácie.

Príklad použitia triedy datastore na prístup k dátam z databázy, prostredníctvom DAO:

Vytvorenie záznamu:

```
this.datastore.createRecord(Post, {
   title: 'My post',
   content: 'My content'
});
```

Nájdenie záznamu:

```
this.datastore.findRecord(Post, '1').subscribe(
    (post: Post) => {
        post.title = 'New title';
    }
);
```

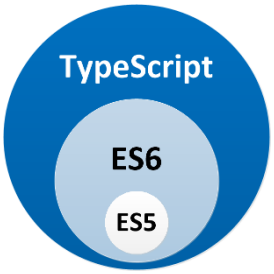
2.3.3 Implementácia

2.3.3.1 Programové prostredie

Vzhľadom na organizovanosť kódu a s ohľadom na jednoduchšiu udržateľnosť vytváranej aplikácie do budúcna sme sa rozhodli ako programovací jazyk pre implementovanie klientskej aplikácie zvoliť jazyk TypeScript. Tento jazyk nám ponúka viac možností ako bežný JavaScript (ECMAScript). Veľmi

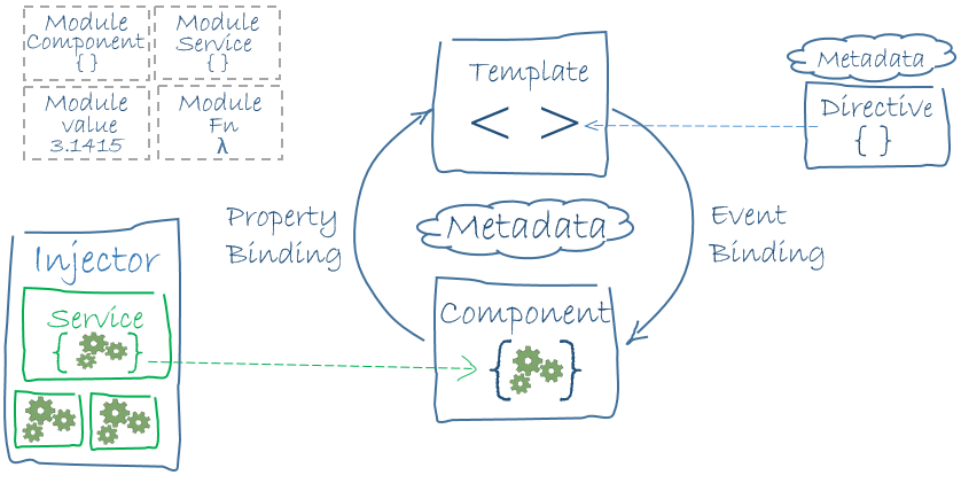
užitočnou funkcionalitou je tiež možnosť dekomponovať jednotlivé časti zdrojového kódu do modulov, vďaka čomu sa zvyšuje prehľadnosť a čitateľnosť kódu.

Z dôvodu nekompatibility TypeScriptu a bežne rozšírených internetových prehliadačov je však potrebné zdrojový kód z jazyka TypeScript skompilovať do jazyka ECMAScript 5. Následne je možné aplikáciu spustiť bežným spôsobom priamo v internetovom prehliadači. Nasledujúci obrázok porovnáva jazyk TypeScript s dvoma verziami jazyka ECMAScript.



Obrázok: Porovnanie TypeScriptu so štandardom ECMAScript

Na prekonanie implementačných prekážok sme sa rozhodli nami vyvíjanú aplikáciu postaviť na frameworku Angular 2 od spoločnosti Google. Angular 2 je kompatibilný s jazykom TypeScript, čo presne splnilo naše požiadavky. Okrem toho framework Angular 2 disponuje viacerými návrhovými vzormi, ktoré nám pomohli aplikáciu implementovať s menšou námahou a robustnejšie. Taktiež komunikáciu so serverovou aplikáciu bolo možné vyriešiť využitím knižnice, ktorá je prepojená s týmto frameworkom. Nasledujúci obrázok zachytáva zjednodušený pohľad na architektúru našej klientskej aplikácie založenej na frameworku Angular 2.



Obrázok: Architektúra Angular 2 aplikácie

2.3.3.2 Zavedenie 3D aspektu diagramov

Nami vyvíjaná aplikácia obsahuje zobrazenie UML diagramov v 3D prostredí. Do tohto 3D prostredia sú umiestnené bežné 2D plátna, medzi ktorými sú vykresľované jednoduché 3D prvky. Napríklad v prípade sekvenčného diagramu sa jedná o interakciu medzi dvoma čiarami života v dvoch 2D plátnach.

Na 2D plátnach sú vykresľované samotné UML diagramy. Vďaka tomuto princípu je možné na vykresľovanie diagramov použiť externé knižnice, ktoré sú prispôsobené na prácu s bežným 2D

prostredím. Celkový 3D efekt je docielený rotovaním 2D plátien s UML diagramami do 3D scény. Na docielenie tejto funkcionality sme využili grafickú knižnicu Three.js.

2.3.4 Testovanie

Testovanie funkčnosti JSON API na frontende (AngularJS/Typescript) prebehlo úspešne a je možné komunikovať s backendom. Taktiež sa podarilo prepojenie GoJS knižnice s AngularJS.

2.4 Grafická reprezentácia častí fyzického modelu

2.4.1 Analýza

Pri programovaní s knižnicou GoJS sme narazali na problém spojený s 3D transformáciou jednotlivých objektov. Po pridaní viacerých plátien a ich následnom zoomovaní alebo rotovaní vznikol problém s následným označovaním objektov v plátne. Taktiež bol problém s pohybom kamery pri jej rotácií. Bolo to spôsobené jadrom samotného GoJS, v ktorom sú elementy implementované v HTML elemente "canvas" (aj plátno je div, atď.). Týmto elementom chýba 3D aspekt, ktorý bol pre nás nevyhnutným (bez neho sa nedalo správne robiť rotácie, atď.).

Dospeli sme k záveru, že je potrebné preprogramovať jadro aplikácie buď zmenou priamo v knižnici GoJS alebo implementáciou vlastných elementov reprezentujúcich objekty vo fyzickom modeli. Tie si môžeme vkladať do HTML tagu "div", pre ktorý už platia 3D transformácie. Na obrázku je možné vidieť sekvenčný diagram vo vrstvách z knižnice GoJS - nebolo možné rotovať jednotlivé objekty v sekvenčnom diagrame ani posúvať (GoJS nepočítal s 3D aspektom).

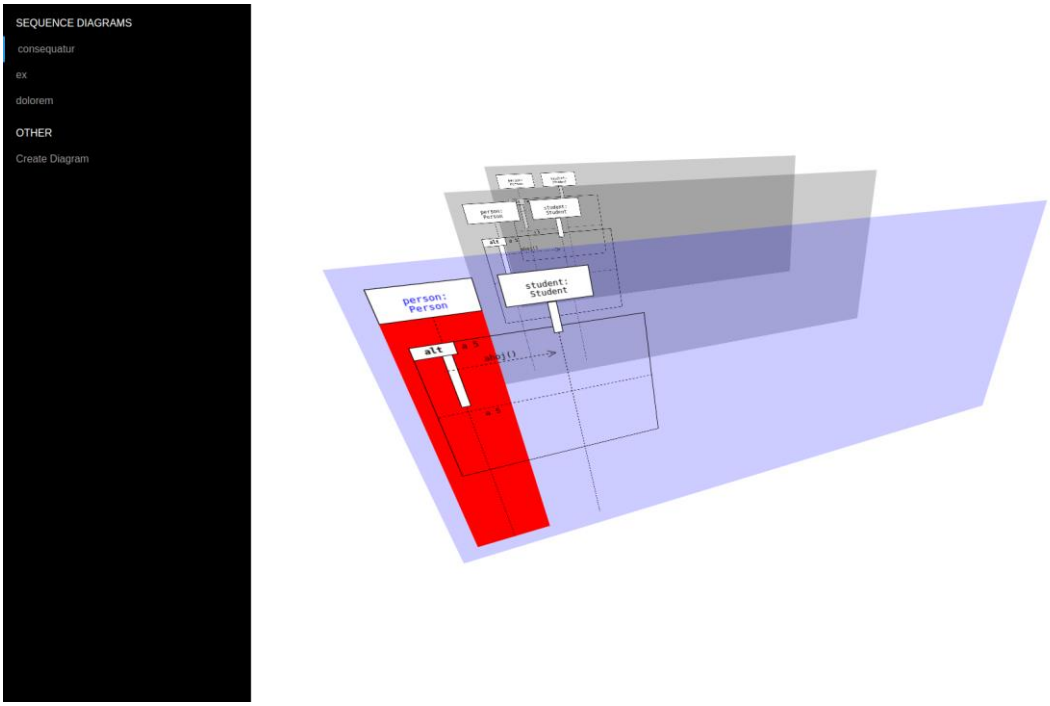
Obrázok: Implementovaná knižnica GoJS vo vrstvách

2.4.2 Návrh

Aplikáciu je potrebné dekomponovať a oddeliť view od modelu, aby sme pri zmenách výzoru aplikácie nemuseli meniť zdrojový kód backendu, keďže nami navrhnutý fyzický model už meniť nebudeme. Usúdili sme, že by bolo vhodné implementovať two-way data binding. To zaručuje, že zmeny vo view sa okamžite prejavia v modeli a naopak.

2.4.3 Implementácia

Aplikáciu sme okrem dekomponovania na MVC dekomponovali taktiež na Angular komponenty. Každý komponent reflektuje objekt z fyzického modelu, taktiež Angular samotný poskytuje two-way data binding. Ako vrchná vrstva nám slúži HTML + CSS a jej 3D transformácie, ktoré vkladáme do tagu "div".



Obrázok: Implementovaná aplikácia vo vrstvách – HTML + CSS