Univerzita Karlova

Pedagogická fakulta

Katedra informačních technologií a technické výchovy

Návrh JS knihovny pro tvorbu atypických forem didaktických testů

A proposal for a JS library for creating atypical didactic tests

Martin Kozub

Vedoucí bakalářské práce: *PhDr. Josef Procházka, Ph.D.*

Studijní program: *B7507 Specializace v pedagogice*

Studijní obor: *B IT (7507R040)*

2017

Prohlašuji, že jsem bakalářskou práci na téma návrh JS knihovny pro tvorbu atypických forem didaktických testů vypracoval pod vedením vedoucího bakalářské práce samostatně za použití v práci uvedených pramenů a literatury. Dále prohlašuji, že tato bakalářská práce nebyla využita k získání jiného nebo stejného titulu.

V Praze dne 14. dubna 2017

....

podpis

Poděkování

Rád bych poděkoval vedoucímu mé práce PhDr. Josefu Procházkovi, Ph.D. za rady a množství času, které mi během konzultací poskytl, stejně jako rodině a přátelům za jejich trpělivost a podporu.

ANOTACE

V úvodní části tato bakalářská práce srovnává existující druhy didaktických testů a vyhledává atypické formy vhodné pro zpracování na počítači, potažmo v prostředí webu. Dále se práce zabývá JavaScriptovými knihovnami a frameworky, jejich rozdíly a adekvátními postupy pro jejich vývoj. Stejně tak ukazuje některé možnosti moderního JavaScriptu.

Cílem práce je analýza možností vývoje specifických typů dotazů didaktických testů v prostředí www. Praktickým výstupem práce je návrh knihovny pro podporu tvorby jednotlivých typů dotazů a modelový didaktický test.

KLÍČOVÁ SLOVA

JavaScript, www, web, JS knihovna, didaktický test, dotazník, formulář

ANNOTATION

The introductory part of this Bachelor's thesis looks into existing types of didactic tests and searches for the forms adequate to be processed on a computer or in a web environment. This thesis also researches what JavaScript libraries and frameworks are, points out their differences and searches for the best methods for their development. In addition some possibilities of modern JavaScript are examined.

The main objective is to analyse development possibilities of the specific query types of didactic tests in the web environment. Practical outcome of this thesis is to design a library simplifying the creation of supported query types and a sample didactic test.

KEYWORDS

JavaScript, www, web, JS library, didactic test, survey, form

Obsah

1 Úvod **7**

2 Teorie didaktického testu **8**

2.1 Druhy didaktických testů 9

2.2 Dělení testů 9

2.3 Základní druhy testových úloh 13

3 Didaktické testy v prostředí webu **17**

3.1 Typy didaktický testů vhodných pro zpracovávání na počítači 17

3.2 Atypické testové úlohy a možné přístupy pro jejich realizaci 17

3.3 Některé z existujících knihoven pro tvorbu testů 18

4 Obecné vlastnosti knihovny **20**

4.1 Rozdíl mezi knihovnou a frameworkem 20

4.2 Kdy použít framework? 20

4.3 Jak by knihovna měla vypadat? 21

4.4 Co při vývoji knihovny neopomenout 21

4.5 Testování softwaru 24

5 Adekvátní metodiky vývoje pro realizaci JS knihovny **27**

5.1 Prototypový přístup 27

5.2 Spirálový přístup 28

5.3 Testy řízený vývoj 28

6 Některá z významných vylepšení v ECMAScriptu 6 oproti předchozím verzím JavaScriptu **30**

6.1 Arrow funkce 30

6.2 Klíčové slovo let a konstanty 30

6.3 Třídy 31

6.4 Promises 32

6.5 Strict mode 33

7 Vlastní vývoj knihovny **35**

7.1 Počátky vývoje . . . 35

7.2 Postupné změny v kódu . . . 37

7.3 Git a GitHub . . . 39

7.4 Rozvržení aplikace . . . 40

7.5 Tvorba dokumentace . . . 51

7.6 Načítání knihovny . . . 54

7.7 Sjednocení knihovny do jednoho souboru a její minifikace . . . 56

7.8 Tvorba ukázkového testu . . . 57

7.9 Funkčnost v různých webových prohlížečích . . . 58

7.10 Hodnocení zvolené metodiky vývoje . . . 61

8 Závěr **62**

9 Seznam použité literatury **64**

10 Přílohy **66**

1 Úvod

Jako téma bakalářské práce jsem si zvolil tvorbu JavaScriptové knihovny pro usnadnění zadávání atypických forem didaktických testů. Pod pojmem „atypické formy didaktických testů" je možné si představit testy obsahující interaktivní a rozšířené klasické otázky, tedy ty, které mají jinou formu než pro webové prostředí běžné úlohy spoléhající na zadávání textových údajů do formuláře.

Důvodem pro můj zájem o danou problematiku je především skutečnost, že podobných projektů, které navíc spoléhají čistě na nativní funkce prohlížeče bez přídavného zásuvného modulu, není mnoho. V dnešní době postupného útlumu doplňků třetích stran se jedná o zvláště aktuální téma.

Praktickým výstupem této práce je JavaScriptová knihovna, která zjednoduší zadávání atypických testů v prostředí webu. Ta by měla být využitelná v co nejvíce prohlížečích, avšak nikoliv na úkor moderních postupů a technologií. Součástí této knihovny bude implementace různých forem otázek, stejně jako řešení jejich ověřování. Práce je zaměřena ryze na klientskou část, validace na serveru není součástí knihovny.

Mezi cíle patří:

- Analýza možností vývoje specifických typů dotazů didaktických testů v prostředí www.
- Návrh knihovny pro podporu tvorby jednotlivých typů dotazů.
- Tvorba modelového didaktického testu.

Výsledná knihovna by měla dodržovat co nejvíce náležitostí, důležitých nebo vhodných pro vývoj JavaScriptové knihovny. A to především vzhledem k možnostem dalšího vývoje a rozšiřování.

2 Teorie didaktického testu

Didaktický test je definován různě, přesto se většina definic shoduje na tom, že „jde o zkoušku, která se orientuje na objektivní zjišťování úrovně zvládnutí učiva u určité skupiny osob". (Chráska, 1999, s. 12) Při tvorbě kvalitního testu a jeho hodnocení jsou pak dodržována určitá předem stanovená kritéria.

Podle citovaného zdroje je důležité si uvědomit, že didaktický test není pouze test písemný, jako například testy řízení vozidel či psaní na stroji, a zdaleka nemusí obsahovat jen úlohy s výběrem odpovědi.

2.1 Druhy didaktických testů

Na základě klasifikace P. Byčkovského (1982) můžeme didaktické testy dělit podle následující tabulky:

Klasifikační hledisko	**Druhy testů**		
měřená charakteristika výkonu	rychlosti		úrovně
dokonalost přípravy testu a jeho příslušenství	standardizované	kvazi-standardizované	nestandardizované
povaha činnosti testovaného	kognitivní		psychomotorické
míra specifičnosti učení zjišťované testem	výsledky výuky		studijních předpokladů
interpretace výkonu	rozlišující (relativního výkonu)		ověřující (absolutního výkonu)
časové zařazení do výuky	vstupní	průběžné (formativní)	výstupní (sumativní)
tematický rozsah	monotematické		polytematické (souhrnné)
míra objektivity skórování	objektivně skórovatelné	kvaziobjektivně skórovatelné	subjektivně skórovatelné

(Chráska, 1999, s. 13)

2.2 Dělení testů

2.2.1 Podle měřené charakteristiky výkonu

Testy rychlosti

Tento typ testů ověřuje, v jak krátkém čase dokáží testované subjekty řešit určitý

úkol. Náplní testů rychlosti jsou jednoduché úlohy a je dán pevně stanovený časový limit. Také se předpokládá, že všichni testovaní obsažené úlohy zvládají a liší se pouze v čase, který na jejich řešení potřebují. Příkladem takových testů jsou testy měření rychlosti čtení či přepisu textu na psacím stroji. (Chráska, 1999, s. 13)

Testy úrovně

Testy úrovně nejsou svázány žádnými časovými omezeními a zkoumá se úroveň vědomostí či dovedností zkoušeného. Z praktických důvodů je však téměř vždy potřeba nějaký limit nastavit, přičemž většina z nejpomalejších testovaných jsou žáci s nejslabšími vědomostmi a prodloužení času jim nepomůže dosáhnout lepšího skóre. Kromě toho je možné do testů úrovně zakomponovat jako vedlejší kritérium i rychlost testovaného při řešení úloh. České školní testy se nejčastěji přibližují právě testům úrovně. (Chráska, 1999, s. 14)

2.2.2 Podle dokonalosti přípravy testu a jeho příslušenství

Testy standardizované

Takto se označují testy připravované profesionálně a důkladně ověřeny tak, že jsou známy všechny jeho základní vlastnosti. Zpravidla se o jejich vydání starají specializované instituce. Součástí je také manuál obsahující informace o vlastnostech a správném použití testu. K dispozici bývá obvykle také norma pro hodnocení testovaných. (Chráska, 1999, s. 14)

Testy nestandardizované

Jedná se o všechny testy, u kterých při jejich tvorbě nebylo dbáno na všechny postupy běžné pro tvorbu standardizovaného testu. Neproběhlo u nich ověření na větším množství žáků a nejsou tak známy přesné vlastnosti testu. Nestandardizované testy si připravují učitelé sami pro vlastní užití, měli by však dbát na dodržování základních pravidel společných i pro testy standardizované.

Do této kategorie spadají i tzv. **testy kvazistandardizované**, neboli testy připravované s větším důrazem na kvalitu, přesto však stále nedosahujícím normy standardizovaného testu. Příkladem může být didaktický test zjišťující úroveň vědomostí žáků v daném předmětu na určité škole či skupině škol. Konstrukci takového testu se zpravidla věnuje větší pozornost. Bývají tak známy některé jeho vlastnosti a k dispozici také normy pro hodnocení. (Chráska, 1999, s. 14)

2.2.3 Podle povahy činnosti testovaného

Testy kognitivní a psychomotorické

Tyto didaktické testy vznikly na základě taxonomie cílů výuky podle B. S. Blooma, tj. učení kognitivní, afektivní a psychomotorické. Je-li cílem zjistit úroveň poznání, jedná se o testy kognitivní — například řešení úloh z matematiky či překlady textů do cizího jazyka. Naopak pokud je předmětem zjistit výsledky psychomotorického učení, jedná se o test psychomotorický — například test psaní na stroji. V pedagogické praxi se s psychomotorickými testy téměř nesetkáme, převládají tedy testy kognitivní. Třetí kategorie, zjišťování výsledků afektivního učení, využívá například dotazníky, nikoliv didaktické testy. (Chráska, 1999, s. 15)

2.2.4 Podle míry specifičnosti učení zjišťované testem

Testy výsledků výuky a studijních předpokladů

Standardním školním prostředkem pro zjištění úrovně poznání žáka je obyčejně test výsledků výuky. Testy studijních předpokladů jsou využívány spíše tam, kde je potřeba zjistit obecnější znalosti jedince potřebné k dalšímu studiu, tj. takové testy by se měly používat převážně u přijímacích testů na vyšší typ školy. Tento typ testů je však náročnější na konstrukci a vyžaduje kromě pedagogické kvalifikace i kvalifikaci psychologickou. (Chráska, 1999, s. 15)

2.2.5 Podle interpretace výkonu

Testy rozlišující a ověřující

Na základě způsobu interpretace výkonu žáka se didaktické testy dělí na testy rozlišující (testy relativního výkonu) a testy ověřující (testy absolutního výkonu). U nás se v praxi téměř výhradně objevují testy rozlišující, jejichž výsledky se porovnávají vůči zbytku populace. Rozlišující testy jsou postaveny takovým způsobem, aby bylo možné určit, jak žák v testu dopadl v porovnání se zbytkem populace. Takže je možné posoudit, zda je žák průměrný, podprůměrný atd.

Na druhou stranu u testů ověřujících se výkon určuje vzhledem k úlohám z přesně vymezené oblasti učiva, takže nedochází ke srovnávání s ostatními žáky. Kritériem pro hodnocení je dopředu stanovený stupeň zvládnutí učiva. U vybraných základních úloh je požadováno prakticky úplné zvládnutí, zde je však dovolena jistá chybovost odpovídající náhodě. Cílem testu je rozhodnout, zdali žák dané učivo zvládl, nebo ne. Klíčový je výběr učiva a rozvržení úloh tak, aby dostatečně pokryla obsažená témata, což může vyžadovat zahrnutí většího počtu úloh. (Chráska, 1999, s. 15–16)

2.2.6 Podle časového zařazení do výuky

Testy vstupní, průběžné a výstupní

Na úvod výuky studijních celků bývají zařazovány testy vstupní. Jejich cílem je zjistit, jaké vědomosti a dovednosti žáci mají, aby učitel mohl přizpůsobit nadcházející výuku co nejlépe potřebám žáků. Výsledky takového testu jsou pro učitele cenné i z toho důvodu, že učiteli prozradí případné nerovnosti ve třídě.

Během výuky se pak používají průběžné didaktické testy poskytující učiteli jakousi zpětnou vazbu. Nejčastěji takové testy zkouší jen malou část učiva. Jejich cílem je zjistit, jak dobře či špatně žáci učivo chápou a jak si ho osvojují. Ke sledování formování vědomostí a dovedností žáka a primárně k hodnocení výuky (nejčastěji ne žáků samotných) pak slouží formativní testy.

Na závěr určitého celku či období jsou zadávány ukončující výstupní testy, které shrnují vyučovanou látku a poskytují informace potřebné pro hodnocení žáků. (Chráska, 1999, s. 16)

2.2.7 Podle tematického rozsahu

Testy monotematické a polytematické

Zatímco monotematické testy jsou zaměřeny na jediné téma, testy polytematické zkoumají vědomosti vícero tematických celků a jsou tak konstrukčně náročnější. (Chráska, 1999, s. 16)

2.2.8 Podle míry objektivity skórování

Testy objektivně skórovatelné

Tyto testy obsahují úlohy, u nichž lze jednoznačně říci, zda byly řešeny správně, či nikoliv. Jejich výhodou je, že jejich opravu může provádět kdokoliv, včetně strojového zpracování. Zde je však nutné si uvědomit, že ačkoliv velká část didaktických testů obsahuje úlohy objektivně skórovatelné, neznamená to, že test je vždy zkouška, která obsahuje výhradně tento typ úloh. (Chráska, 1999, s. 17)

Testy subjektivně skórovatelné

Naopak testy subjektivně skórovatelné jsou takové, pro jejichž hodnocení není možné stanovit jednoznačná pravidla. Spadají zde například otevřené úlohy, kde žák volně odpovídá na určitou otázku. Jejich výhodou je schopnost zkoušet mnohem komplexnější vědomosti a dovednosti, než je možné v úlohách objektivně skórovatelných. (Chráska, 1999, s. 17)

2.3 Základní druhy testových úloh

Při tvorbě didaktických testů má autor na výběr z poměrně velkého množství typů úloh. J.A.Průcha uvádí jako základní úlohy tyto:

1. Otevřené úlohy
   (a) Se širokou odpovědí
      i. Nestrukturované
      ii. Se strukturou
         A. S vymezenou strukturou (zkoušejícím definovaná struktura odpovědi)
         B. S danou konvencí (struktura vyplývá z konvence, kterou by měl zkoušený znát)
   (b) Se stručnou odpovědí
      i. Produkční
      ii. Doplňovací
2. Uzavřené úlohy
   (a) Dichotomické
   (b) S výběrem odpovědi
   (c) Přiřazovací
   (d) Uspořádací

Ve výčtu výše jsou úlohy rozděleny podle toho, zda je odpověď tvořena (otevřené úlohy) či zda je vybírána z nabízených možností (uzavřené úlohy). (Chráska, 1999, s. 26)

2.3.1 Otevřené úlohy se širokou odpovědí

Tento typ úloh požaduje delší odpověď na dané téma. Může vyžadovat pojednání na určité téma, návrh řešení určitého problému nebo popis určitého procesu. Výhodou těchto úloh je, že testují komplexní znalosti a dovednosti žáka. Dále se poměrně snadno navrhují, ale nejsou objektivně skórovatelné. (Chráska, 1999, s. 25)

2.3.2 Otevřené úlohy se stručnou odpovědí

Jedná se o úlohy, které nutí žáka vytvořit krátkou odpověď na dotazovanou otázku, například číslo, značku, vzorec či slovo až krátkou větu. Tyto úlohy se dále dělí na

produkční a doplňovací, kde produkční úlohy kladou nejprve otázku a za ní vymezují místo pro odpověď, zatímco doplňovací úlohy otázku nekladou, ale v uvedených větách či souvětích ponechávají volná místa, kde vyžadují doplnění vhodných vsuvek. Podobně jako úlohy se širokou odpovědí se i tyto tvoří poměrně snadno, navíc nedávají zkoušenému tolik prostoru pro uhodnutí správné odpovědi, jako je tomu u uzavřených testů. Naopak častým problémem je podle J.A.Průchy sice správná odpověď zkoušeného, ale jiná, než si představoval autor testu. (Chráska, 1999, s. 27–28)

Doporučení pro návrh úloh se stručnou odpovědí

1. Úloh užívejte jen tehdy, lze-li odpovědět velmi stručně (nejlépe jen jedním údajem).
2. Úlohu formulujte zcela jasně a jednoznačně.
3. Nevyžadujte doslovné opakování textu z učebnice.
4. Uvažte předem všechny možné odpovědi, a je-li jich mnoho, raději úlohu nepoužívejte.
5. Ponechejte v úlohách vždy dostatek místa pro uvedení odpovědi.
6. Dávejte přednost produkčním úlohám před doplňovacími. Chcete-li přece jen použít doplňovací úlohy, dodržujte následující doporučení:
   (a) Vynechávejte jen důležité údaje.
   (b) Z neúplné věty musí být patrné, co se má doplnit.
   (c) Údaj, který se má doplnit, umisťujte pokud možno na konec věty.
   (d) Pokud se má doplnit několik údajů, vynechte pro doplnění zhruba stejné místo.

(Chráska, 1999, s. 29)

2.3.3 Uzavřené dichotomické testové úlohy

V tomto typu úloh jsou testovanému předkládány dvě odpovědi s tím, že jedna je správná, a tu má nějakým způsobem označit. Můžou to být úlohy s klasickými možnostmi ano/ne, ale i s jinými dvěma variantami. Ač se velmi snadno navrhují, svádějí testování detailů či triviálností. Dalším problémem je nezanedbatelná pravděpodobnost uhodnutí správné odpovědi, s čímž lze ale bojovat zahrnutím většího počtu úloh do testu. (Chráska, 1999, s. 29)

Doporučení pro návrh dichotomických úloh

1. Tvrzení uváděné v úloze musí být jednoznačně správné, anebo nesprávné.
2. Nepoužívejte příliš dlouhých tvrzení.
3. V tvrzeních nepoužívejte dvojího záporu.
4. V tvrzeních nepoužívejte výrazů typu často, téměř, vždy, nikdy, zřídka apod.
5. Navrhujte zhruba stejný počet správných a nesprávných tvrzení.
6. Nepoužívejte vět vytržených z učebnice, ani je neobměňujte zařazením záporu.

(Chráska, 1999, s. 30)

2.3.4 Uzavřené úlohy s výběrem odpovědí

Jedná se o úlohy s větším množstvím definovaných odpovědí a dále se dělí na úlohy s jednou správnou odpovědí, jednou nejpřesnější odpovědí, jednou nesprávnou odpovědí a vícenásobnou odpovědí.

Patří zde také situační úlohy, které jsou zvláštní variantou úloh s výběrem odpovědi. V tomto typu úloh student vybírá z rozsáhlejšího seznamu možností, který však nebývá uveden, ale vyplývá z dané situace (například cifry 0–9). Kvůli velkému množství možností je pravděpodobnost uhodnutí správné odpovědi zpravidla nízká. (Chráska, 1999, s. 30–33)

Doporučení pro návrh úloh s výběrem odpovědí

1. Úlohami s výběrem odpovědí nezkoušíme pokud možno zapamatování konkrétních poznatků.
2. Ve formulaci úlohy se vyhýbáme slovům nebo údajům, které by mohly sloužit jako nápověda.
3. Pokud se ve formulaci úlohy vyskytuje zápor, zvýrazníme jej např. podtržením.
4. Soubor nabízených odpovědí k jedné úloze by měl být homogenní, tj. podobný obsahovým zaměřením i formou.
5. Distraktory [1] se nesmějí navzájem překrývat nebo jinou formou vyjadřovat totéž.

---

[1] Distraktor je pojem z oboru kognitivní psychologie a vyjadřuje necílový podnět stěžující vyhledávání.

KOHOUTEK, Rudolf. Pojem distraktor. In: *ABZ slovník cizích slov* [onli věne]. Ostrava: ABZ knihy, c2005-2017 [cit. 2017-04-10]. Dostupné z: http://slovnik-cizich-slov.abz.cz/web.php/slovo/distraktor

6. Umístění správné odpovědi mezi distraktory se má volit zcela náhodně.
7. Navrhujeme jen takové distraktory, u nichž je předpoklad, že budou využívány.
8. Při používání úloh s vícenásobnou volbou odpovědi a při používání neurčitých odpovědí na tuto skutečnost žáky upozorníme.
9. Při formulaci úloh s výběrem odpovědí dáváme přednost otázkám před neúplnými tvrzeními.
10. V úlohách s výběrem odpovědí se vyhýbáme příliš dlouhým slovním formulacím.

(Chráska, 1999, s. 37)

2.3.5 Uzavřené přiřazovací úlohy

Tyto úlohy sestávají ze dvou množin pojmů a jejich cílem je spojit pojmy z jedné množiny s pojmy v druhé množině. Velikost těchto dvou množin by se měla lišit tak, aby některé pojmy nebyly vůbec spárovány. To snižuje pravděpodobnost, že žák uhodne správné řešení, zvláště v případě, že některé odpovědi zná a zůstane mu jen pár nepřiřazených možností. (Chráska, 1999, s. 37)

2.3.6 Uzavřené uspořádací úlohy

Cílem těchto úloh je uspořádání jednotlivých prvků jedné třídy do řady. Součástí úlohy je vždy instrukce, podle jakého kritéria dané prvky seřadit, například podle velikosti, významu, chronologicky atp. (Chráska, 1999, s. 38)

3 Didaktické testy v prostředí webu

3.1 Typy didaktický testů vhodných pro zpracovávání na počítači

Z pohledu počítačového programu je sice možné realizovat didaktický test obsahující všechny kategorie úloh uvedené v předchozí kapitole, pakliže ale potřebujeme do programu zahrnout také automatickou kontrolu správnosti odpovědí, nedokážeme se stoprocentní jistotou hodnotit úlohy otevřené. Zde samozřejmě platí pravidlo, že čím stručnější či jednoznačnější odpověď vyžadujeme, tedy čím menší šanci dáváme k nečekané odpovědi, tím větší je šance, že ji náš algoritmus správně vyhodnotí.

Bezpečným příkladem budiž otázka typu: „Jaké je hlavní město České Republiky?“, na kterou očekáváme, že respondent odpoví pouhým názvem města „Praha“. Maličkosti jako nadbytečné mezery před a za slovem či záměna velkého písmene za malé lze poměrně snadno odchytit, aniž by ovlivnily hodnocení testu.

3.2 Atypické testové úlohy a možné přístupy pro jejich realizaci

Mezi atypické testové úlohy patří především uzavřené úlohy přiřazovací a uspořádací, pro něž neexistují ve webovém prostředí nativní HTML komponenty. Pro běžnější úlohy doplňovací, s výběrem odpovědi či zaškrtávací lze těchto komponent využít.

Úlohy přiřazovací a uspořádací se od sebe liší již v základu a to jak v počtu potřebných elementů, tak i ve způsobu hodnocení. Oboje je při vývoji nutné vzít v potaz.

3.2.1 Přiřazovací úlohy

V přiřazovacích úlohách je nutné počítat s tím, že nám nestačí pouze jedno vstupní pole, do kterého budeme přesunovat jednotlivé objekty, ale v naprosté většině případů budeme chtít zachovat následující poměr

$$n\colon m,\ n, m \in \mathbb{N},\ 1 < n \leq m,$$

kde n je počet objektů k přiřazení a m počet polí, kam lze jednotlivé objekty přiřadit. Pokud bychom chtěli zadávat úlohu s menším počtem objektů než cílů, pak by výše uvedený poměr pochopitelně neplatil — mnohem četnější jsou ale úlohy o stejném nebo větším počtu objektů.

Pro implementaci univerzálního nástroje pro přiřazovací úlohy tedy musíme nutně počítat s následujícími možnostmi:

1. Proměnný počet objektů k přiřazení i cílových oblastí
2. Možnost bijekce, injekce i surjekce, což je nutno zohlednit především při kontrole výsledků

Vzhledem k tomu, že je cílem nejen správný hodnotící algoritmus a vizuální efekt, tedy ono zmíněné přesunování objektů, je potřeba hodnocení postavit na skutečnosti, zda a které objekty se nacházejí v oblasti cílového pole. Je tedy potřeba porovnávat pozice objektů vůči cíli a zahrnout do nich také další parametry jako šířku a výšku jednotlivých elementů.

3.2.2 Uspořádací úlohy

Implementačně obdobně obtížným typem úloh jsou uspořádací úlohy, kde si ale vždy vystačíme pouze s jedním vstupním polem pro prakticky libovolné množství objektů. Aby měla uspořádací úloha smysl, bude vyhovovat poměru

$$n\colon m,\; n \in \mathbb{N} \setminus \{0, 1\}\,,\; m \in \{1\},$$

kde n je počet objektů k přiřazení a m počet polí. Je zjevné, že nemá smysl seřazovat pouze jeden objekt, který by sám o sobě nebyl k čemu porovnávat. I zde lze však situaci zkomplikovat na úlohy řazené vertikálně a horizontálně. Oboje je však z pohledu vyhodnocování prakticky totéž, pouze s malým rozdílem při zjišťování pořadí sledovaných objektů.

Podobně jako v přiřazovacích úlohách je i zde, vzhledem k nutnosti objekty posunovat, potřeba porovnávat pozice pro vytěžení údajů při odevzdávání testu.

3.3 Některé z existujících knihoven pro tvorbu testů

Mezi dostupné JavaScriptové knihovny pro tvorbu testů patří například Alpaca [2] nebo pokročilejší survey.js [3]. Obě tyto varianty generují formuláře na základě předaných objektů a předpokládají u nich určité atributy, jejichž podrobnější specifikaci si lze dohledat ve zveřejněné dokumentaci.

Jak Aplaca, tak survey.js podporují validaci s tím, že survey.js je v tomto ohledu o něco dále, neboť definuje několik předdefinovaných principů validace. Aplaca naopak nechává

[2] *Alpaca* [online]. Chicago (Illinois): Gitana Software, c2017 [cit. 2017-04-06]. Dostupné z: http://www.alpacajs.org/

[3] *Survey.js* [online]. Rusko: Andrew Telnov, c2015-2017 [cit. 2017-04-06]. Dostupné z: http://surveyjs.org/index.html

o něco více práce na autorovi aplikace, který konkrétní knihovnu využije, a požaduje vlastní ověřovací funkci pro každou jednotlivou otázku.

Žádný z uvedených nástrojů ovšem nepodporuje uspořádací ani přiřazovací otázky a soustředí se tak čistě na klasické webové fomuláře.

Čistě pro validaci fomuláře lze zvolit některý z existujících nástrojů, například knihovnu validate.js [4]. Ovšem ve spojení s formuláři generovanými některou z výše uvedených metod je třeba počítat s možnými komplikacemi a jejich možné vzájemné nekompatibilitě.

[4] *Validate.js* [online]. San Francisco (Kalifornii): Rick Harrison, 2016 [cit. 2017-04-06]. Dostupné z: http://rickharrison.github.io/validate.js/

4 Obecné vlastnosti knihovny

Pod pojmem knihovna rozumíme určitou skupinu funkcí, popřípadě procedur nebo v objektovém programování soubor objektů, které jsou nezávislé na samotné aplikaci a díky tomuto oddělení mohou být použity v mnoha různých aplikacích zároveň ve stejné nedotčené podobě. [5]

Naproti tomu framework je struktura, která poskytuje obecnou funkcionalitu, jež může být dodatečně definovaným kódem vývojáře ovlivněna, čímž vzniká výsledná aplikace. Jedná se prakticky o prostředí, které bývá součástí větší softwarové platformy a zjednodušuje práci s ní, čímž umožňuje vznik komplexních aplikací v podstatně kratším čase. [6]

4.1 Rozdíl mezi knihovnou a frameworkem

Knihovna i framework mají za cíl programátorovi usnadňovat tvorbu vlastní aplikace. Pod oběma pojmy se skrývá nezávislý kus kódu, který může vývojář použít a který zpřístupňuje určitou funkcionalitu.

Framework se ale od knihovny liší tím, že zachází ještě dál a definuje pravidla pro strukturu kódu v něm napsané aplikace. Při dodržení postupů potom framework sám volá kód aplikace. Knihovna na druhou stranu poskytuje své služby, aniž by měnila strukturu kódu — jedná se prakticky o balík funkcí nebo tříd, přičemž po jejich zavolání je aplikaci navrácena kontrola.

Oba mají definované určité API, ale knihovna zprostředkovává jen některou úlohu či některé úlohy v aplikaci, zatímco framework tvoří její kostru a API definuje, jak aplikaci poskládat tak, aby fungovala. (Kanted a K, 2011-2014)

Platí tedy, že knihoven může aplikace využívat více, zatímco framework je jen jeden. Stručně je možné situaci shrnout do následujícího tvrzení: „Aplikace využívá knihovny, ale je psána ve frameworku“. (Majda, 2009)

4.2 Kdy použít framework?

Pokud autorovi aplikace dostačují možnosti nabízené frameworkem, potom je vhodné je využít — vývoj půjde snáze a rychleji. Na druhou stranu stačí i jen maličkost,

[5] Knihovna (programování). In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2016 [cit. 2017-01-21]. Dostupné z: https://cs.wikipedia.org/wiki/Knihovna_(programování)

[6] Software framework. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2017 [cit. 2017-01-21]. Dostupné z: https://en.wikipedia.org/wiki/Software_framework

něco, co framework neumožňuje nebo dělá jinak, a programování se může proměnit ve strastiplný proces. Úprava jedné věci může vyústit v problém úplně někde jinde a tak se z vývoje aplikace stane záplatování překážek jednu po druhé, jak se postupně objevují. (Taylor, 2010)

4.3 Jak by knihovna měla vypadat?

„Až na pár výjimek by se měla knihovna vždy skládat z jednoho nebo několika souborů v jednom adresáři. Její kód by měl být spravován odděleně a měl by při její implementaci do vlastního projektu zůstat nedotčený. Knihovna by měla dovolit nastavit pro vlastní projekt specifickou konfiguraci nebo chování.“ (Severien, 2016)

V případě JavaScriptové knihovny uvádí Checkman na svém blogu požadavek na to, aby byla knihovna od zbytku kódu oddělena tak, že by měla být zabalena do vlastního objektu. Pokud by si definovala veškeré funkce a proměnné v globálním namespace, zvyšovalo by se riziko redeklarace již existujících názvů vlastním kódem, stejně jako potenciál pro narušení vlastní knihovny dále načítaným kódem. (Checkman, 2014)

4.4 Co při vývoji knihovny neopomenout

Knihovna obecně je ucelený balík kódu, který může vývojář ve svém projektu použít, aniž by se musel hlouběji zabývat jí pokrývanou problematikou. Proto je důležité, aby knihovna splňovala určité základní požadavky tak, aby se s ní lidem lépe pracovalo a nebyla více přítěží, než pomocí.

4.4.1 Rozsah a cíle

Před zahájením samotného programování je vhodné si rozmyslet, do jaké hloubky bude knihovna některý problém či skupinu problémů obalovat. Použití knihovny by rozhodně nemělo být náročnější než řešení problému přímo v jeho čisté podobě bez knihovny — proto platí, že čím jednodušší je API [7], tím snáze a rychleji se bude s knihovnou dále pracovat.

[7] API (Application Programming Interface), je soubor procedur, funkcí, tříd nebo protokolů programu, knihovny nebo i operačního systému, které může programátor využít ve své vlastní aplikaci. Cílem API je definovat způsob komunikace mezi těmito dvěma prvky tak, aby bylo zřejmé, jak příslušná volání realizovat.

API. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2016 [cit. 2017-04-03]. Dostupné z: https://cs.wikipedia.org/wiki/API

4.4.2 Přizpůsobitelnost

Jak již bylo zmíněno výše, knihovna by měla být do určité míry také flexibilní. Různé knihovny se v tomto ohledu chovají odlišně — některé umožňují programátorovi jen základní operace a neponechávají prostor pro další konfiguraci, jiné naopak zpřístupňují velké množství funkcí a stávají se tak univerzálně vhodné pro mnohem širší škálu aplikací.

Uživatelům knihovny může být možnost konfigurace zpřístupněna několika způsoby a podle toho je možné ovlivňovat chování knihovny, ať již předáváním parametrů (typicky konstruktoru), úpravou veřejných metod objektů knihovny či skrze zpětná volání (callbacky) a události.

Většinou se nastavení chování knihovny provádí při inicializaci, někdy je ale možné provádět úpravy za běhu.

Zdrojový kód 1 Konfigurace při inicializaci

```
1 var instanceKnihovnyXYZ = new XYZ({
2         separator: ';'
3 });
```

Zdrojový kód 2 Konfigurace za běhu použitím veřejné metody

```
instanceKnihovnyXYZ.nastavHodnotu('separator', '-');
```

Zdrojový kód 3 Konfigurace za běhu použitím veřejné proměnné

```
instanceKnihovnyXYZ.separator = '-';
```

Zdrojový kód 4 Callback - obvykle pro asynchronní úlohy

```
1 instanceKnihovnyXYZ.asynchronniUloha(function hotovo() {
2         // Kód, který se má spustit po provedení určité interní
             akce knihovny
3 });
```

Zdrojový kód 5 Události

```
1 instanceKnihovnyXYZ.on('nazev-udalosti', function hotovo(e, data
    ) {
2         // Kód, který se má spustit po provedení určité interní
             akce knihovny
3 });
```

4.4.3 Testování

Pokud má být knihovna robustní, tj. má být zajištěna funkcionalita, jejíž funkčnost se navíc při každé úpravě knihovny neomezí, je vhodné knihovnu doplnit také o automatické testy. K tomu je k dispozici několik frameworků, které lze pro testování použít. Ideální je pokrýt každou funkci knihovny testem, popřípadě několika testy, díky čemuž je možné zachytit většinu regresí ještě před vydáním nové verze knihovny.

4.4.4 Dokumentace

Čím větší je knihovna, tím důležitější je strávit čas také nad tvorbou dokumentace, která slouží jako odrazový můstek pro ty, kteří by ji chtěli ve svém projektu využít. Úvod dokumentace by měl obsahovat základní informace, především shrnutí toho, co knihovna dělá, aby bylo ostatním hned jasné, zda je pro ně vůbec vhodná.

Další část dokumentace by měla obsahovat popis API, nejlépe doplněnou ještě o ukázky použití. Popis API je možné si usnadnit psaním vhodných komentářů, kterým nástroje jako JSDoc [8] samy rozumí a dokáží na jejich základě vypracovat ucelený přehled.

Záleží také na volbě licence, která může být buď otevřená (open-source), nebo proprietární.

4.4.5 Publikování

Zveřejnění knihovny na vlastních webových stránkách je jednou z možností publikace knihovny. V tomto případě o ni ale asi mnoho lidí nebude jevit zájem a velmi pravděpodobně ji ani neobjeví. Mnohem vhodnější je pro uložení zdrojových kódů a veškeré dokumentace použít publikační platformu, jako například GitHub [9]. Veškeré soubory potom budou veřejně k dispozici, nebo v případě této služby za měsíční poplatek soukromé.

Velmi oblíbeným způsobem publikace hotové knihovny je pak npm [10], což je balíčkovací správce, v jehož repozitářích jsou k dispozici prakticky všechny velké knihovny a který výrazně usnadní použití knihoven možným zájemcům. Kromě npm ještě existuje podobný správce Bower [11], jeho vliv ale s rostoucím časem klesá. (Severien, 2016)

[8] JSDoc (http://usejsdoc.org/) je značkovací jazyk, který slouží pro komentování zdrojových kódů psaných v JavaScriptu.

[9] GitHub (https://github.com/) je na internetu dostupná platforma, která umožňuje hostovat a sdílet softwarové projekty, a využívá k tomu verzovací systém Git.

[10] npm (https://www.npmjs.com/) je správce balíčků, který usnadňuje instalaci mnoha různých nástrojů pro nejrůznější JavaScriptové projekty. Podporuje správu závislostí a umožňuje získávat ke sdíleným balíčkům odezvu od jejich uživatel.

[11] Bower (https://bower.io/) je správce balíčků, jež mají za cíl usnadnit vývojářům tvorbu webových stránek a to zjednodušením přístupu k potřebným knihovnám, frameworkům a dalším nástrojům.

4.4.6 Zjednodušení

Z výše zmíněných doporučení je ale zjevné, že toho do začátku není na vývojáře vůbec málo. Vytvořit kvalitní API s novými verzemi příliš neměnné, uzpůsobit knihovnu tak, aby se dala snadno při jejím vývoji testovat, a napsat dostatečně rozsáhlou a užitečnou dokumentaci zabere velké množství času. Zvláště pak v případě, nejedná-li se o týmovou práci. Z definice knihovny navíc tyto věci nevyplývají, jsou ale vývojářskou komunitou velmi vítané. (Chaudhary, 2014)

Je tedy zjevné, že ač jsou tyto prvky zcela běžnými a prakticky nepostradatelnými u velkých projektů, jako jsou například jQuery či Angular, nedá se s nimi jednoznačně počítat u menších projektů, které typicky vytváří jen jeden vývojář, často bez finanční podpory a mimo pracovní dobu.

4.5 Testování softwaru

Oblíbeným a běžně používaným způsobem pro zajištění definované funkcionality určitého programu či nástroje včetně správné vizuální reprezentace je využití některé z dostupných metod testování softwaru. Tím je možné odhalit celou řadu chyb, ať už jsou jejich příčiny jakékoliv. Tímto způsobem je také možné zamezit zavlečení nových chyb do funkčního celku programu, které se nejčastěji objevují při změnách existujícího kódu či jeho rozšiřování. [12]

Testování je možné rozdělit na dvě základní větve a to na funkcionální a nefunkcionální.

4.5.1 Funkcionální testování

Tento druh testování je postaven na znalosti chování softwaru, tedy jeho zevnějšku ve smyslu vstupů a výstupů — v případě testování nás nezajímá, co se děje uvnitř. Během testování je aplikaci vnucen nějaký vstup a na základě porovnání skutečného výstupu s očekávaným výstupem je možné dojít k závěru.

Unit testy

Jedná se o typ testování, které provádí sám vývojář. Cílem je zjistit, zda každá součást programu funguje tak, jak má. Jako každé testování ale není možné ani unit testy vykoušet všechny možné scénáře a tudíž ani zachytit všechny chyby v aplikaci. [13]

[12] Why is software testing necessary? In: *ISTQBExamCertification* [online]. ISTQBExamCertification [cit. 2017-01-19]. Dostupné z: http://istqbexamcertification.com/why-is-testing-necessary/

[13] Software Testing - Levels. In: *Tutorialspoint* [online]. Hyderabad (Indie): tutorialspoint, c2017 [cit. 2017-01-19]. Dostupné z: https://www.tutorialspoint.com/software_testing/software_testing_levels.htm

Zavedení těchto testů do již existujícího projektu je často velmi náročné a vyžaduje poměrně velké úpravy v kódu. Z toho důvodu je vhodné se rozhodnout již v počátcích vývoje, zda se tento typ testů bude v softwaru využívat. (Hlava, 2011)

Integrační testy

Na rozdíl od unit testů, které ověřují jednotlivé části softwaru odděleně, se integrační testy zabývají již určitými funkčními celky aplikace a jejich správné komunikaci mezi sebou. U větších projektů se o jejich spouštění typicky nestarají vývojáři, ale práce připadá na specializovaný testovací tým. (Hlava, 2011)

V případě testování je možné začít odspodu, nejprve testy jednotlivých součástí a následně pokračovat testováním celků, nebo naopak začínat z vyšší vrstvy a poté se věnovat jednotlivým modulům. [13]

Systémové testy

Prověřují správný chod programu jako celku a většinou je provádí speciální testovací tým. Cílem je zaručit určitou kvalitu softwaru, tedy splnění požadavků technické specifikace, a provádí se typicky v prostředí podobném tomu, kde má být výsledná aplikace nasazena. [13]

Testování probíhá z pohledu zákazníka a simulují se tak různé procesy, které mohou v praxi nastat. (Hlava, 2011)

Regresní testy

Mají význam při změnách v již existujícím programu a jejich cílem je zamezit regresím způsobeným úpravami, kdy by oprava jedné chyby či přidání nové funkcionality mohlo vyústit v chybu někde, kde předtím nebyla. [13]

Akceptační testy

Po vyhovění všech předchozích testů je software předán zákazníkovi, který si sám ověří funkčnost aplikace, čímž se zajistí, že software vyhovuje jak specifikacím, tak klientovi. (Hlava, 2011) To znamená, že umožňují předcházet překlepům, vizuálním chybám i nedostatkům v celkovém rozhraní aplikace a soustředí se také na stabilitu softwaru.

Kombinace unit testů, integračních testů a systémových testů je někdy označována za tzv. alfa testování. Následuje beta testování, které již probíhá na skutečných lidech a je zaměřeno na skupinu, pro kterou je produkt určen. [13]

4.5.2 Nefunkcionální testování

Na rozdíl od funkcionálního testování se v případě tohoto druhu testování nezkoumá jaké výsledky aplikace vrací v různých situacích, ale testují se vlastnosti, jako jsou výkon, bezpečnost či uživatelské rozhraní. [13]

5 Adekvátní metodiky vývoje pro realizaci JS knihovny

Před samotným vývojem knihovny je důležité zvolit systém, jakým bude její vývoj probíhat. Ať již víceméně intuitivně (stylem postupných přírůstků a nabalováním funkcionality na neúplný kód) nebo jinými způsoby (například testy řízeným vývojem).

5.1 Prototypový přístup

Jedná se o poměrně rozšířenou a oblíbenou metodiku vývoje softwaru, kdy během jednotlivých iterací dochází k tvorbě tzv. prototypů, pod nimiž si lze představit nekompletní, zjednodušenou implementaci celého systému, nebo naopak plně funkční část systému.

Proces prototypování lze shrnout do následujících kroků:

1. **Zjištění a pochopení požadavků**: Jedná se o zcela základní pochopení toho, co má software dělat ve smyslu jeho vstupů a výstupů.
2. **Vývoj prvního prototypu**: Jeho cílem je zakomponovat naprosto základní požadavky a poskytnout uživateli interface, skrze který uvidí, jak tento prototyp funguje nebo spíše co by měl umožňovat, protože větší část zatím nemusí fungovat.
3. **Zhodnocení**: Vytvořený prototyp je prezentován a odezva ve smyslu, co by se mělo přidat nebo upravit, je zaznamenána.
4. **Revize**: Na základě hodnocení je rozhodnuto, co se má změnit, a dochází opět k tvorbě nového prototypu. Tento cyklus se opakuje, dokud produkt nevyhovuje požadavkům.

Výhodou tvorby prototypů je skutečnost, že již v raných fázích vývoje je možné si systém do určité míry vyzkoušet a testovat. Nejedná se sice o plně funkční software, ale je z něj na první pohled patrné, o co se pokouší a co ještě chybí či nefunguje. Určitou nevýhodou tohoto postupu je, že se finální systém může stát příliš komplexním a složitým, neboť dopředu nemusejí být známy všechny požadavky. (Šmíd, 2002)[14][15]

[14] SDLC - Software Prototype Model. In: *Tutorialspoint* [online]. Hyderabad (Indie): tutorialspoint, c2017 [cit. 2017-02-04]. Dostupné z: https://www.tutorialspoint.com/sdlc/sdlc_software_prototyping.htm

[15] In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2017 [cit. 2017-02-04]. Dostupné z: https://en.wikipedia.org/wiki/Software_prototyping

5.2 Spirálový přístup

V korporátním prostředí se lze setkat se spirálovou metodikou blízkou prototypování, se kterou jsem se ovšem při vývoji vlastní knihovny nepotýkal. Tento způsob kombinuje prototypování společně s analýzou rizik. Podobně, jako ve výše uvedené metodice, se i zde pracuje v cyklech, přičemž v každém novém cyklu se na otestovanou funkční část nabaluje nový rozšiřující kód.

Postup vývoje prováděného touto cestou lze rozdělit do čtyř kvadrantů:

1. **Analýza**: Definice cílů a plán řešení.
2. **Vyhodnocení**: Analýza různých alternativních řešení, identifikace a řešení rizik.
3. **Vývoj**: Vývoj prototypu a jeho validace, zda pracuje v souladu s požadavky.
4. **Plánování pro příští iteraci.**

Tento model umožňuje podobně jako přidružený prototypový model rychle reagovat na požadavky klientů, ale kromě toho také dokáže analýzou rizik předcházet chybám. Výhodou obou modelů je také skutečnost, že je možné systém hodnotit již v jeho raných stádiích vývoje. [16]

5.3 Testy řízený vývoj

Jedná se o další způsob přístupu k vývoji softwaru. V případě testy řízeném vývoji dochází během procesu k malým, stále se opakujícím krokům, které celý vývoj zefektivňují. Funkcionalitu je potřeba si nejprve definovat a následně pro ni vytvořit test, který ji ověřuje. Až teprve poté přichází na řadu samotné psaní kódu a jeho úprava.

K testování se používají automatické unit testy, o jejichž existenci jsem se zmínil v sekci Testování softwaru. Jak již z názvu vyplývá, unit testy se kontrolují nejmenší jednotky programu, jako jsou třídy nebo jejich metody a funkce. Vzhledem k tomu, jak hluboko automatické testy jdou, spadají spíše do náplně práce samotných vývojářů, nikoliv testerů, neboť vyžadují porozumění kódu.

5.3.1 Vývojový cyklus

1. **Psaní testu**: Nejprve je potřeba vytvořit test, který si ohlídá, zda určitá část programu funguje podle očekávání. Jedná se prakticky o definici požadavků. Tento

[16] SDLC - Spiral Model. In: *Tutorialspoint* [online]. Hyderabad (Indie): tutorialspoint, c2017 [cit. 2017-04-08]. Dostupné z: https://www.tutorialspoint.com/sdlc/sdlc_spiral_model.htm

způsob nutí vývojáře zamýšlet se nad požadavky ještě před samotným psaním kódu.

2. **Spuštění testů „naprázdno“**: V dalším kroku se při spuštění testů vývojář ujistí, že testy neprojdou — žádná funkcionalita zatím není definována. Tímto krokem se částečně eliminují chyby v napsaných testech.

3. **Psaní vlastního kódu**: Nyní je již čas začít psát samotný kód aplikace. Cílem není psát od prvopočátku vyloženě efektivní a elegantní kód, ale jde spíše o to, aby všechny testy prošly. Pokud kód testy prochází, je možné přejít k dalšímu kroku.

4. **Refaktoring**: V tomto kroku se řeší opomenuté vlastnosti z bodu č. 3, tedy především čistota a efektivita napsaného kódů. Díky automatickým testům a jejím opětovném spouštění je možné kontrolovat, zda byla při úpravách zachována veškerá funkcionalita a nenastaly žádné chyby.

Při úpravách je vhodné testování spouštět co nejčastěji tak, aby bylo jednoduché případné destruktivní změny revertovat.

Výhodou tohoto postupu je rychlé, přesné a pohodlné odhalení chyby, a to díky opětovnému spouštění velkého množství testů na několik málo nových změn. [17] [18]

[17]In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2017 [cit. 2017-02-06]. Dostupné z: https://cs.wikipedia.org/wiki/Programování_řízené_testy

[18]Test Driven Development. In: *Tutorialspoint* [online]. Hyderabad (Indie): tutorialspoint, c2017 [cit. 2017-02-06]. Dostupné z: https://www.tutorialspoint.com/software_testing_dictionary/test_driven_development.htm

6 Některá z významných vylepšení v ECMAScriptu 6 oproti předchozím verzím JavaScriptu

6.1 Arrow funkce

ES6 přináší podporu pro nový zápis funkce ve formátu: `() = > { příkaz; }` nebo, předává-li se funkci parametr, dokonce jen takto: `parametr => { příkaz; }`

Mezi jeho přednosti patří například výrazně kratší syntaxe. Pokud je v těle funkce jen jeden výraz, složené závorky se dokonce mohou vynechat. Kosmetická změna ale není to jediné, v čem se tato funkce liší. Její pravděpodobně největší předností oproti klasické definici funkce je skutečnost, že nemění hodnotu parametru `this`. Pro zachování kontextu uvnitř anonymní funkce, definované například v rámci objektu nebo posluchače události, se běžně používal následující kód:

Zdrojový kód 6 Ukázka dříve běžného způsobu pro zachování přístupu k objektu

```
1  function Zpozdeni(id) {
2          this.id = id || 0;
3  }
4
5  Zpozdeni.prototype.spust = function() {
6          var self = this;
7          setTimeout(function(){
8                  console.log(self.id);
9          }, 1000);
10 }
11
12 var instance = new Zpozdeni(0);
13 instance.spust();
```

I přesto, že uvnitř těla funkce vnořené do časovače, již `this` není totéž, jako přímo v objektu, záložní proměnná `self` nám přístup k `this` umožňuje.

S ES6 a arrow funkcemi se nicméně situace změnila a uvnitř tohoto nového zápisu se hodnota `this` sama od sebe nemění.

Arrow funkce jsou vždy anonymní.

6.2 Klíčové slovo let a konstanty

Velkou novinkou je také zavedení slova `let`, které zastává podobnou funkci, jako slovo `var`. Umožňuje nám tedy definovat proměnnou s tím rozdílem, že takto definovaná

proměnná zasahuje jen tam, kam skutečně patří, zatímco `var` definuje proměnné buď globálně nebo lokálně, ale pro celou funkci.

Zdrojový kód 7 Rozdíl mezi proměnnou definovanou var a let

```
1  var i = 0;
2  let j = 0;
3
4  if(true) {
5          var i = 1;
6          let j = 1;
7          console.log(i, j); // Vypíše 1, 1
8  }
9
10 console.log(i, j); // Vypíše 1, 0
```

Ve výše uvedeném kódu je dobře patrný rozdíl mezi `var` a `let`. Podobně platí také to, že definujeme-li `let` proměnnou ve `for` cyklu takto:

Zdrojový kód 8 Ukázka for cyklu s použitím slova let v JavaScriptu

```
1  for(let i = 0; i < 10; i++) {
2          console.log(i);
3  }
```

potom níže, po ukončení těla `for` cyklu, není proměnná `i` definována, zatímco v případě proměnné uvozené slovíčkem `var` by stále existovala.

Pro deklaraci konstanty s přístupem pouze pro čtení je nově definováno slovo `const` a celý zápis vypadá například takto:

Zdrojový kód 9 Použití konstanty

```
const vaha = 10;
```

6.3 Třídy

Ve starších verzích JavaScriptu neexistovalo pro definici třídy klíčové slovo `class`. Funkce a jejich metody se definovaly nejčastěji prototypy objektů nebo se, nevadilo-li, že se metody znova vytvářejí s každou novou instancí objektu, vkládaly přímo do hlavní funkce jako proměnné. (Stefanov, 2006)

Dnešní verze JavaScriptu umožňuje o něco jednodušší a standardnější zápis tříd a jejich metod a stejně tak podporuje dědičnost, volání předka `super`, instanční a statické

metody a také konstruktory. Deklarace třídy využívající všechny zmíněné prvky může vypadat například takto:

Zdrojový kód 10 Třída a její dědičnost v ECMAScriptu 6

```
1  class Zednik extends Clovek {
2          constructor(data) {
3                  this.jmeno = data.jmeno;
4                  this.prijmeni = data.prijmeni;
5          }
6
7          get celeJmeno() {
8                  return this.jmeno + " " + this.prijmeni ;
9          }
10
11         povyrust() {
12                 super.vek(super.vek + 1);
13                 console.log(this.celeJmeno + "je nyní o jeden
                      rok starší");
14         }
15
16         static typ() {
17                 return "člověk";
18         }
19 }
```

I když je výše demonstrovaná syntaxe v JavaScriptu stále novinkou, neboť jí spousta vývojářů kvůli zpětné kompatibilitě nepoužívá, pro zkušenější vývojáře nejde o nic převratného a vzhledem k podobnosti se zápisem deklarací tříd v jiných jazycích si na ni rychle zvyknou.

6.4 Promises

Promise je objekt, který reprezentuje konečný výsledek asynchronní operace. Ačkoliv je tento koncept starší než revize ES6, doposud nebyl nativní součástí JavaScriptu. Jeho síla spočívá ve standardizaci realizace asynchronních úloh. K interakci s Promise se používá metoda `then`, která přijímá callback funkce pro úspěch, v tom případě jí dokáže předat také výsledek asynchronní operace. Na druhou stranu metoda catch přijímá funkci pro neúspěch, přičemž v takovém případě může funkci předat důvod neúspěchu.

Jedna z možností využíti Promise:

Zdrojový kód 11 Použití Promise

```
1  function test() {
2          return new Promise(
3                  function(resolve, reject) {
4                          window.setTimeout(() => {
5                                  resolve("Hodnota");
6                          }, 1000);
7                  }
8          );
9  }
10
11 test().then((hodnota) => { console.log(hodnota); }).catch((
       reason) => { console.error(reason) });
```

Kromě uvedených nových prvků se v ECMAScriptu 6 objevily také další věci, jako nové kolekce — například mapa pro ukládání hodnot v páru s klíči, generátory — tedy zvláštní iterátory umožňující navázání předchozího kontextu, moduly — které slouží pro import a export a šablonovací textové řetězce, do kterých lze na specifikovaná místa vkládat výsledky výrazů.

Jejich detailnější popis ovšem není potřeba v práci uvádět, neboť tyto prvky nejsou v projektu využity. (Rieseberg, 2015)

6.5 Strict mode

K výše zmíněným a použitým novinkám ECMAScriptu 6 jsem se také rozhodl psát celý kód v restriktivním režimu JavaScriptu „strict mode". Na rozdíl od normálního stavu eliminuje strict mód některé tiché chyby a hlásí je. Dále se odstraňují problémy z minulosti, které zpomalují JavaScriptový engine, díky čemuž mohou některé části kódu běžet rychleji než v režimu normálním. Strict režim je tedy rychlejší v odstraňování neduhů předchozích verzí JavaScriptu. Další vlastností je zákaz používání některých slov, s nimiž se počítá do budoucna v rámci implementace nové funkcionality.

Striktní režim se objevil až ve verzi JavaScriptu ECMAScript 5, nicméně dřívější verze JavaScriptu jeho definici ignorují, neboť se nejedná o příkaz ale o textový výraz. Uvození strict módu tedy vypadá takto:

Zdrojový kód 12 Deklarace striktního režimu

```
"use strict";
```

Tento mód lze definovat pro celý skript nebo ho vnořovat do funkcí a používat ho tak jen na specifických místech.

Jednou z velkých změn restriktivního módu je mj. změna hodnoty `this` uvnitř těla volané funkce. Zatímco standardně se za `this` skrývá globální objekt window, ve strict módu je hodnota `this` odpovídá `undefined` . [19] [20]

[19] Strict mode. In: *Mozilla Developer Network* [online]. Mountain View (Kalifornie): Mozilla Contributors, 2017 [cit. 2017-02-14]. Dostupné z: https://developer.mozilla.org/cs/docs/Web/JavaScript/Reference/Strict_mode

[20] Transitioning to strict mode. In: *Mozilla Developer Network* [online]. Mountain View (Kalifornie): Mozilla Contributors, 2017 [cit. 2017-02-14]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode/Transitioning_to_strict_mode

7 Vlastní vývoj knihovny

Na začátku vývoje knihovny jsem stál před rozhodnutím, jaké další knihovny a frameworky využívat a jaký styl programování nasadit. Tedy zda se snažit spíše o strukturovaný či o objektový přístup.

I přes to, že jsem měl v této fázi již velkou část problematiky prostudovánu, narazil jsem v první iteraci na skutečnosti, jejichž rozsah jsem si zpočátku neuvědomoval a které jsem se posléze rozhodl změnit.

7.1 Počátky vývoje

Pro tvorbu knihovny jsem se rozhodl využít objektový přístup pro psaní kódu a závislost na velmi rozšířené knihovně jQuery. [21] Na její funkcionalitu jsem velmi intenzivně spoléhal ve všech částech knihovny do sebemenších detailů — od typického využívání selektorů, vytváření DOM elementů, přidávání tříd jednotlivým elementům i vkládání textu. [22]

7.1.1 Členění kódu

Vlastní kód jsem začal dělit do jednotlivých tříd podle toho, co která třída obsluhuje. Od počátku jsem postupoval odshora dolů, čímž jsem vytvořil obalující třídu, která drží určitý soubor úloh a dále se rozpadá na několik podtříd (například otázka v kvízu, která se dále za pomoci dalších tříd stará mj. o jednotlivá vyplňovací pole spadající k dané otázce). Tímto dělením se mi podařilo dosáhnout určitého zjednodušení kódu i přes velikost a především komplexnost celého projektu.

7.1.2 Návrh a zpracování

Knihovnu jsem od počátku navrhoval tak, že si v ideálním případě vytváří veškeré objekty na stránce sama. Nepočítá tedy s možností, že by stavěla na existujícím formuláři. Důvodem pro toto mé rozhodnutí byla především nutnost pracovat s validací a držet si pro celý soubor i jednotlivé otázky komplexní data. Tímto vývojem se mi z velké části podařilo oddělit reprezentaci formuláře od konkrétních dat. Knihovna se také stala o něco univerzálnější v tom smyslu, že ji lze snáze propojit s aplikacemi, které ji používají při konstrukci testu nebo v návazných operacích.

[21] jQuery je JavaScriptová knihovna kompatibilní s většinou i starších prohlížečů, která usnadňuje manipulaci s objekty v HTML, včetně věcí, jako jsou události a animace.

[22] S rostoucím časem jsem si začal uvědomovat, že povinná závislost na této či jiných knihovnách nemusí být vyhovující a jQuery jsem se začal postupně zbavovat.

Při vývoji knihovny jsem se držel prototypování a začínal s velice jednoduchou verzí ukázkového testu, která toho příliš neuměla a obsahovala pouze primitivní textovou otázku. S její pomocí jsem se pokoušel rozvrhnout si, jakým způsobem bude probíhat interakce mezi aplikací a knihovnou a jaké parametry nastavení bude ve svém počátečním vývojovém stádiu knihovna podporovat. To znamená, že jsem se zabýval především podobou aplikačního rozhraní API a tím, co všechno by má knihovna mohla obecně vzato umožňovat.

Níže je k vidění ukázka definice textové otázky se všemi atributy, které jsem si dal za cíl u tohoto typu otázky podporovat:

Zdrojový kód 13 Možná definice textové otázky

```
1  ukazkovyTest.addQuery({
2          id: "name",
3          type: "text",
4          title: "Jak se jmenujete?",
5          subtitle: "Zadejte vaše celé jméno v ASCII",
6          label: "Jméno:",
7          checkRule: ["/^[a-zA-Z ]+$/"],
8          error: "Toto není platné jméno",
9          css: "background: #3ff;"
10 });
```

V úvodní fázi vývoje knihovna ještě ani zdaleka nedokázala všechny výše definované parametry využít, například zcela opomíjela kontrolu uživatelem zadaných dat a její validaci na základě definovaných pravidel. Stejně tak zatím neumožňovala vkládání pokročilejších typů otázek.

K jejich zařazování jsem se uchýlil až v momentě, kdy knihovna dokázala správně fungovat s větším množstvím textových otázek v jednom kvízu a podporovala většinu prvků ze stanovené funkcionality. To znamená, že po textových otázkách jsem postupně přidával i přiřazovací a uspořádací otázky, a ve finále také otázky s výběrem jedné (radio) a více (checkbox) odpovědí. Programově nejsložitější byla v tomto ohledu právě implementace přiřazovacích a uspořádacích úloh, které vyžadovaly schopnost interakce na tažení myší a rozpoznání umístění v rámci jednotlivých cílových polí dané otázky.

Dalším neméně důležitým krokem byla validace. K tomu, aby ji ale bylo vůbec možné realizovat, bylo nutné několika kroků. Především přijetí definice správných odpovědí a dovednost rozpoznání uživatelsky zadaných výsledků. Tato schopnost sama o sobě byla komplikovaná především pro otázky využívající tažení, neboť je pro její funkčnost nutné porovnávat umístění předdefinovaných polí s pozicemi uživatelem přetažených možností.

Samotná validace potom probíhá v několika krocích a to proto, že dokáže porovnávat zadané údaje s textovými řetězci, booleovskými hodnotami i s regulárními výrazy, díky čemuž není problém v testu validovat například správně zadanou emailovou adresu či telefonní číslo. Stejně tak je možné všechny 3 typy hodnot přiřadit do pole, díky čemuž je validace velmi přizpůsobitelná mnoha různým požadavkům a nemusí se tak omezovat pouze na několik málo možností. Pro zadavatele testu je tento způsob daleko jednodušší zvláště v kombinaci více regulárních výrazů. Pro správnou odpověď stačí, aby vyhovoval alespoň jeden z nich, a tak není vždy potřeba vymýšlet komplexní všechny možnosti pokrývající regulární výraz.

Poslední viditelnou částí je vyhodnocení kvízu, které může být podle požadavků odesláno na server a může okamžitě zobrazit seznam otázek s informacemi o správnosti odpovědí uživateli. Na potvrzení formuláře je možné si připojit vlastní funkce přímo z aplikace a tímto schopnosti knihovny rozšířit nebo si je upravit k obrazu svému.

Knihovna nevynucuje žádné vlastní kaskádové styly. Nepřibaluje žádný soubor, který by styly obsahoval, nicméně dokumentace doporučuje určitá nastavení pro některé elementy, bez jejichž dodržení nemusí být samotný test funkční. Pro jednoduchost je možné přímo využít ukázkový test nebo se jím alespoň inspirovat.

7.2 Postupné změny v kódu

Během prvních pár týdnů vývoje jsem se rozhodl změnit svůj přístup k jQuery. Především jsem se v celé aplikaci přesunoval na čistý JavaScript a začal více využívat ECMAScript ve verzi 6, který podporuje snadnou práci s DOM objekty. [23]

Důvodem pro tento tah bylo především mé rozhodnutí nenutit uživatele knihovny přikládat ke stránce jQuery, které třeba jinak nepoužívají. K odstranění závislosti na jQuery mi velmi pomohly nové prvky JavaScriptu, které do určité míry tuto knihovnu nahrazují. Díky tomu jsem tak mohl psát poměrně příjemně čistý JavaScriptový kód. Většinu částí, které jsem již měl napsány v jQuery, bylo možné také celkem nenáročně přepsat bez potřeby větších úprav či nutnosti vymýšlet nové postupy.

Původně jsem také implementoval jednotlivé třídy knihovny pomocí objektů a jejich prototypů. S přesunem k ECMAScriptu 6 jsem si mohl dovolit využít o něco příjemnější a jazykově univerzálnější zápis ve standardní a známé formě definic tříd, jejich konstruktorů a metod. Stejně tak bylo možné odstranit některé starší, ne tak přehledně psané, konstrukty.

[23] DOM (Document Object Model) je objektovou reprezentací XML či HTML dokumentu, používanou ve webových prohlížečích. Všechny aktuální verze prohlížečů využívají pro tento model standard organizace W3C.

7.2.1 Posun objektů na stránce: Jquery Draggable, interact.js

Při odstraňování závislosti na jQuery mě ovšem čekal jeden hůře řešitelný problém s knihovnou jQuery Draggable, která jQuery využívá a jež po tento moment zprostředkovávala přesun objektů v rámci jednotlivých otázek kvízu. V případě zjišťování pozic objektů po jejich přetažení jsem dokázal poměrně snadno a rychle odstranit závislost na jQuery, neboť je obdobně možné tuto informaci získat také standardní cestou. Ovšem co se samotného přetahování položek otázky týče, stál jsem před možností zůstat závislý na jedné knihovně nebo nutností napsat si poměrně komplexní kód pouze pro samotné tahání, což by znamenalo duplikovat již po několikáté práci mnoha jiných JavaScriptových knihoven.

Protože jsem ale v žádném případě nechtěl nechat vlastní knihovnu záviset na jQuery, zvláště v momentě, kdy již byla její funkce v knihovně minimalizována, rozhodl jsem se jít zlatou střední cestou. Díky tomu jsem nemusel vytvářet vlastní kód pro tahání objektů v prohlížeči, ale na místo toho se mi povedlo otevřít vrátka mnoha různým rozšířením a udělat tak knihovnu nezávislou na jediném řešení.

To znamená, že aktuálně knihovna nativně podporuje jQuery Draggable a Interact.js, pro které má napsány obslužné metody. Jedna zastřešující metoda jim potom předává vše, co pro nastavení přetažitelných objektů potřebují. Tedy především element, který má přesun umožňovat, a akci, která se má spustit po jeho přetažení. Jednu z podporovaných možností je si dále možné zvolit předáním hodnoty „interact“ nebo „jQuery“ při inicializaci formuláře.

Takto by se ale stále nejednalo o zcela univerzální řešení, neboť dvě zabudované možnosti nezahrnují všechny způsoby tahání elementů. Proto knihovna kromě toho umožňuje zvolit si vlastní plug-in [24] pro přesun objektů, což musí být vlastní funkce se specifickým rozhraním, která tak může obalovat příslušné schopnosti libovolné jiné knihovny, nebo také vlastní, „na koleně“ napsaný způsob umožňující interaktivní přemisťování předaného objektu.

Případná funkce třetí strany nemusí nutně splňovat veškeré podmínky a implementovat veškerou funkcionalitu, což může mít za následek nefunkční pohyb objektů nebo třeba jen omezené schopnosti, jako nechtěné umožnění přemístění objektů mimo vlastní otázku. Závisí tedy na uživateli, aby vše implementoval tak, aby bylo chování formuláře v souladu se zabudovanými metody. Na druhu stranu dává tento způsob vývojářům prostor přizpůsobit si přetahování podle představ včetně dalšího možného rozšíření.

Výhodou tohoto přístupu, a tedy univerzálnosti, je především snížení datového toku

[24] Plug-in, někdy také plugin či zásuvný modul, je software, který slouží jako doplňkový modul k jiné aplikaci a rozšiřuje tím jeho funkčnost, přičemž sám o sobě není funkční.

v případě, že už vývojář na stránce, kde využívá knihovnu pro tvorbu kvízů, používá nějakou další knihovnu, která posuny umožňuje. Tím pádem nemusí přikládat ke stránce vnucené závislosti pouze kvůli tomuto formuláři, ale může si napsat jednoduchou obalovou funkci pro interakci s jím zvolenou knihovnou obsahující vlastních implementací této schopnosti.

Například pro jQuery Draggable může být příslušná funkce až takto jednoduchá:

Zdrojový kód 14 Příklad definice funkce pro přesun možností otázky

```
1 function draggablePlugin_jQuery(data) {
2         $(data.item).draggable({
3                 containment: data.container || null,
4                 stop: data.onEnd || null
5         });
6 }
```

Nadále tak má knihovna využívá jQuery a jQuery Draggable pouze tehdy, pokud si o to uživatel řekne. Ve výchozím stavu se naopak knihovna přiklání k interact.io, a tím je závislá na menší a především jen jediné knihovně na místo dvou.

7.3 Git a GitHub

Krátce po započetí vývoje jsem začal svůj kód kvůli potřebě zaznamenávání změn a pro případ, kdy by bylo nutné vrátit se o krok či několik kroků zpět, verzovat systémem Git. [25] [26]

Pro ten jsem se rozhodl vzhledem k jeho oblíbenosti mezi vývojáři, mnoha dostupným návodům a kompletní dokumentaci. Kromě toho mi tento systém přišel vhod také pro sdílení kódu na internetu, neboť existuje velké množství různých služeb, které umějí přímo s Gitem pracovat. Mezi nejčastější v této oblasti patří webový repozitář GitHub, jehož služby jsou u veřejných projektů k dispozici zdarma a který je používán pro distribuci kódu nespočetně mnoha jednotlivci i firmami. Interface GitHubu je také důmyslně propracovaný a zpřístupňuje všechny verze, přičemž navíc zobrazuje jejich rozdíly.

[25] Git je multiplatformní distribuovaný systém správy verzí vytvořený Linusem Torvaldsem — tvůrcem operačního systému Linux. Slovy Linuse je Git nejlepší volbou pro správu verzí a mnohem kvalitnější a použitelnější, než zkostnatělý a centrálně distribuovaný Subversion (SVN), pro což uvádí ve svém projevu také několik příkladů (Torvalds, 2007). Git je vhodný pro malé i rozsáhlé projekty a je dostupný pod otevřenou licencí GPL v2.

[26] *Git: --distributed-even-if-your-workflow-isnt* [online]. New York: Software Freedom Conservancy, 2017 [cit. 2017-02-14]. Dostupné z: https://git-scm.com/

Zálohy vlastního projektu jsem současně prováděl tak, že jsem si lokální repozitář Git se soubory z projektu zrcadlil právě na serveru GitHub.

Stránka projektu společně se zdrojovým kódem jsou k dispozici na webové adrese: https://github.com/zubozrout/DT

7.4 Rozvržení aplikace

7.4.1 Třídy

FormsHolder

Cílem třídy FormsHolder je udržovat si informaci o všech formulářích na jedné webové stránce, které byly instanciovány s využitím vlastní knihovny. Jelikož ale nejspíše obecně platí, že vývojář bude chtít mít na určité stránce jen jeden formulář, je její význam aktuálně víceméně hypotetický a spíše skrývá jistý potenciál do budoucna.

Třída FormsHolder původně měla využívat návrhového vzoru singleton, jehož implementace ale v JavaScriptu verze ECMAScript 6 vyžaduje použití globální proměnné, která si udržuje stav třídy. Konkrétně je v takové proměnné uložena reference na instanci třídy nebo `null`. Celý proces volání konstruktoru pak v tomto případě vypadá následovně:

Pokusí-li se někdo danou instanci vytvořit, konstruktor třídy si vždy zkontroluje, zdali se náhodou již na adrese uložené v proměnné nenachází instance patřičné třídy. Pokud ještě ne, zapíše referenci na sama sebe do dané proměnné nebo v opačném případě proměnnou nepřepíše. Nakonec konstruktor vždy vrátí to, co se v proměnné nachází. Tedy první zapsanou referenci, ať již je to aktuální instance nebo instance, která byla vytvořena dříve, čímž se zajistí přístup k jedné unikátní instanci po celou dobu chodu programu.

Zdrojový kód 15 Implementace návrhového vzoru Singleton v ECMAScript 6

```
1  let singleton = null;
2
3  class TridaSingleton {
4          constructor() {
5                  if(typeof singleton === typeof undefined) {
6                          // Chyba, proměnná singleton neexistuje.
7                  }
8
9                  if(!(singleton instanceof TridaSingleton)) {
10                         // V globální proměnné singleton zatím
                              není instance TridaSingleton,
                              konstruktor zde tak vloží sebe sama.
11                         singleton = this;
12                         return this;
13                 }
14                 // V globální proměnné singleton již instance té
                      to třídy je. Konstruktor ji tedy vrátí a dál
                      nic nedělá.
15                 return singleton;
16         }
17 }
18
19 singleton = new TridaSingleton();
```

Protože ale tento přístup vytváří prostor pro chybu z hlediska redeklarace proměnných v globálním prostoru, rozhodl jsem se ho nakonec nevyužít. Raději, bude-li autor testu spoléhat na tuto třídu, nechat na něm, aby si pro celou svou aplikaci vytvořil ideálně pouze jednu její instanci. (Virk, 2015)

CommonFormFunctions

Jejím cílem je zpřístupnit určité převážně obecné metody v celé knihovně. Obsahuje tedy zjednodušený interface pro některé typy úloh, jako jsou odstranění všech potomků HTML elementu, získání pozice objektu, přidání inline stylu, sloučení dvou znakových řetězců s použitím předaného separátoru a nebo také přístup ke XMLHttpRequestu pro jednodušší nahrávání dat na server.

Kromě zmíněných metod zprostředkovává také funkcionalitu přesunu pro libovolný objekt a to generickou funkcí napojitelnou na jakýkoliv nástroj, který tažení podporuje. Ve výchozím stavu obsahuje dvě metody. Jednu pro použití s jQuery Draggable a druhou s interact.js. Další způsoby vyžadují předání adekvátní funkce uživatelem při inicializaci formuláře.

Všechny metody v této třídě jsou vytvořeny jako metody statické. Protože nemá smysl,

aby sama třída byla instanciovaná, je vedena jako třída abstraktní. Veškeré přístupy k jejím metodám jsou tedy prováděny stejným způsobem, jako v následujícím příkladu:

`CommonFormFunctions.empty(domElement);` [27]

CommonFormMethods

Abstraktní třída CommonFormMethods obsahuje základní metody děděné dalšími třídami, konkrétně InputItemWrapper, InputItem a OptionItem. Některé její metody vyžadují přístup k datům, které tato třída sama nemá k dispozici. Tím se její abstraktnost stává nutnou, neboť při její přímé inicializaci by se stala velmi fragilní.

Interface, který tato třída zpřístupňuje, usnadňuje mimo jiné přístup k relevantní DOM části, tedy přidruženému elementu (jako je vstupní textové pole v případě InputItem), jeho pozici a k datům z instance třídy QueryDataBlock, která je předávána všem zmíněným třídám. Touto funkcionalitou jim zajišťuje určitý jednotný standard.

Form

Třída Form je základem každého formuláře a drží jeho strukturu. Na základě předaných parametrů vygeneruje nadpis a rozhoduje o tom, jak se bude formulář chovat, včetně toho co se má dít po odeslání odpovědí. Dále realizuje přidávání otázek, přičemž pro každou z nich generuje také pomocnou instanci QueryDataBlock.

Je odpovědná za volání validace při snaze odeslat formulář, za tvorbu obalu formuláře a odesílacího tlačítka nebo na vyžádání použití vlastních elementů — formuláře a tlačítka submit. Sumarizuje data do textové podoby, dokáže je odesílat v serializované podobě na definovanou webovou adresu a také vypsat přehled odpovědí v závěru po odeslání formuláře.

Během inicializace třídy je možné, jak již bylo zmíněno výše, definovat vlastní formulář a odesílací tlačítko. Další parametry se týkají odesílání. Například, zdali se má po pokusu o odeslání uživateli vynutit oprava se zvýrazněním chyb s možností skoku na první chybnou otázku nebo zda se má formulář rovnou odeslat. Kromě toho je také možné předat třídě vlastní funkci, která se zavolá po události odeslání formuláře. Té je v argumentu předána událost formuláře a její parametr `this` je nastaven na konkrétní instanci Form pro přímý přístup k metodám této třídy.

Díky tomu je možné s třídou Form pracovat velice univerzálně. Veškeré základní a běžné úkony dokáže řešit sama, ale zároveň je možné jí velké množství i základních úkonů parametrizovat tak, aby byly čistě v rukou vývojáře aplikace, který knihovnu používá.

[27]Static. In: *Mozilla Developer Network* [online]. Mountain View (Kalifornie): Mozilla Contributors, 2017 [cit. 2017-02-17]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/static

Díky této škálovatelnosti a rozšířitelnosti je knihovna použitelná v mnohem větším rozsahu, než pro co je sama uzpůsobena. Je tak možné vytvořit například daleko propracovanější shrnující obrazovku po odeslání formuláře nebo vytvořit si jinou vlastní sadu funkcí v návaznosti na odesílání formuláře.

QueryDataBlock

Jedná se o pomocnou třídu potřebnou pro inicializaci každé instance třídy QueryItem. Udržuje informace o všech možnostech, kterých může třída QueryItem nabývat. Jejím cílem je usnadnit přístup k datům každé otázky i v nižších třídách, a to především poskytnutím jednotného rozhraní. Ukazatele na konkrétní instanci třídy se dále předávají i níže v hierarchii otázky a zprostředkovávají tak přímý přístup k datům.

QueryValidator

Tato třída realizuje veškeré ověřování dat každé z otázek formuláře. Řeší tedy jednoduché textové vstupní hodnoty, otázky s výběrem jedné nebo více odpovědí i komplexní otázky využívající přesunu možnosti, které spadají mezi uspořádací a přiřazovací formy.

Dokáže pojmout pravidla pro validaci ve třech různých podobách a to jak obyčejný text, tak logické hodnoty true a false a také regulární výrazy. Tím otevírá zadavateli testu poměrně širokou škálu možností, které může v otázkách využít. Kromě toho navíc třída dokáže zpracovávat nejen jednu takovou hodnotu, ale také celé pole hodnot jednoho z podporovaných typů či i jejich kombinace.

Třída realizuje odlišný způsob validace pro různé typy otázek a tuto informaci si zjistí z hlavní datové třídy QuerDataBlock. Zároveň v případě výčtu více možných odpovědí podporuje také jejich validaci s použitím and nebo or logiky. To znamená, že dokáže vyžadovat, aby byly uživatelem zadány všechny odpovědi vyskytující se v definovaném poli hodnot správně nebo jim vyhovovala alespoň jedna jedna z nich.

Pro uspořádávání možností otázky navíc zachovává zadavatelem testu definované pořadí a na jeho základě provádí následnou validaci. O správné pořadí uživatelem zadaných hodnot ze seřazovaných hodnot se ale QueryValidator nestará a závisí tak na třídě InputItem, aby jí předala korektně uspořádané hodnoty.

QueryItem

QueryItem je třída, jež obaluje jednotlivé otázky ve formuláři. Udržuje si jak seznam input elementů, tak „tahatelných“ objektů, jsou-li v otázce využívány. Tato třída dále řeší kontrolu pozic uživatelem přetažených objektů vůči umístění input boxů na stránce.

Při inicializaci je QueryItem předán QueryDataBlock, na jehož základě je vygenerován HTML obal pro jednotlivé input boxy a případné posunovatelné možnosti. Součástí je

také výpis titulku a popisku. Třída se dále stará o inicializaci tříd InputItemWrapper a OptionItem, které jsou vytvářeny na základě dat otázky z QueryDataBlock.

InputItemWrapper

Třída InputItemWrapper nejen obaluje konkrétní jeden HTML input tag, ale řeší také výpis popisku, tzn. label, přidruženého ke konkrétnímu HTML input objektu. Kromě toho se stará také o výpis informace o špatné odpovědi v případě, že je tato zpětná vazba povolena.

InputItem

Tato třída se již stará pouze o výpis input elementu a jeho validaci. Jako jediná v celé knihovně emituje událost změny obsahu HTML input elementu a obsahuje metodu, která na tuto událost dokáže navázat callback.

OptionItem

Řešení možností je na rozdíl od inputů veskrze jednodušší a je realizováno pouze jednou třídou. Úspora kódu je umožněna skutečností, že jednotlivé možnosti, tedy „tahatelné“ objekty, jsou realizovány pouze jedním HTML elementem a také se u nich neřeší validace. Ta přísluší třídě InpuItem a jí volané obslužné třídě QueryValidator.

Změna pozice objektu je zachycena již o třídu výše, tedy u QueryItem, a dále skrze metodu OptionItem přeposlána objektu InputItem, do kterého byla DOM reprezentace OptionItem přesunuta.

7.4.2 Průběh validace

Validace probíhá na formuláři formou kaskády, kdy v případě potřeby formulář rozešle požadavek na validaci každé z otázek a tyto požadavky pak nadále proplouvají až k cílovým uzlům otázek (k jednotlivým inputům) nebo přesněji jejich obalující třídě InputItem. Výsledek validace pak opět prochází celou strukturou zpět až ke zdroji volání a obalová třída Form ve finále získá od každé z otázek data, která může dále zpracovávat.

Samotná validace obsažená ve třídě InputItem je závislá na typu otázky. V případě uspořádací otázky jsou ještě před samotnou validací hodnoty seřazeny podle toho, jak jsou zastoupeny v dokumentu na základě uživatelské interakce. Následně jsou patřičné hodnoty, ať již jen jedna nebo více ve správném pořadí, odeslány metodě QueryValidator, která byla inicializovaná s metodou QueryDataBlock a která si udržuje všechny hodnoty k porovnávání konkrétního vstupního pole. Tato metoda dokáže porovnávat

hodnoty datových typů String, Boolean a regulární výrazy RegExp včetně zachování pořadí pro správné vyhodnocení uspořádacích otázek.

Pro názornější přehled je k dispozici přiložený diagram tříd.

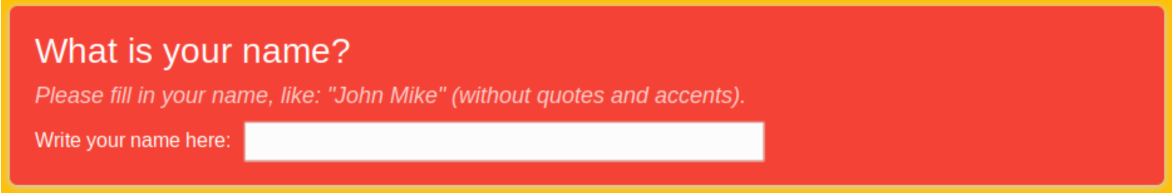
7.4.3 Podporované typy otázek

Aktuálně knihovna podporuje pět základních typů otázek:

Textová otázka

Jedná se o typ otázky s jedním vstupním polem, která umožňuje vkládání znaků z klávesnice. Validace tohoto typu může spočívat v porovnávání zadaného textu s předdefinovanými řetězci, s logickými hodnotami i s regulárními výrazy včetně jejich kombinací.

Ukázka pro tuto metodu je k dispozici zde:
http://kraken.pedf.cuni.cz/~kozubm/dt/example/query-examples/text.html



Obrázek 7.1: Textová otázka

Otázka s výběrem jedné odpovědi

Tento typ otázky využívá prvky radio, které umožňují výběr právě jedné odpovědi z předem definovaného seznamu možností. Validace může spočívat v porovnávání logických hodnot true nebo false.

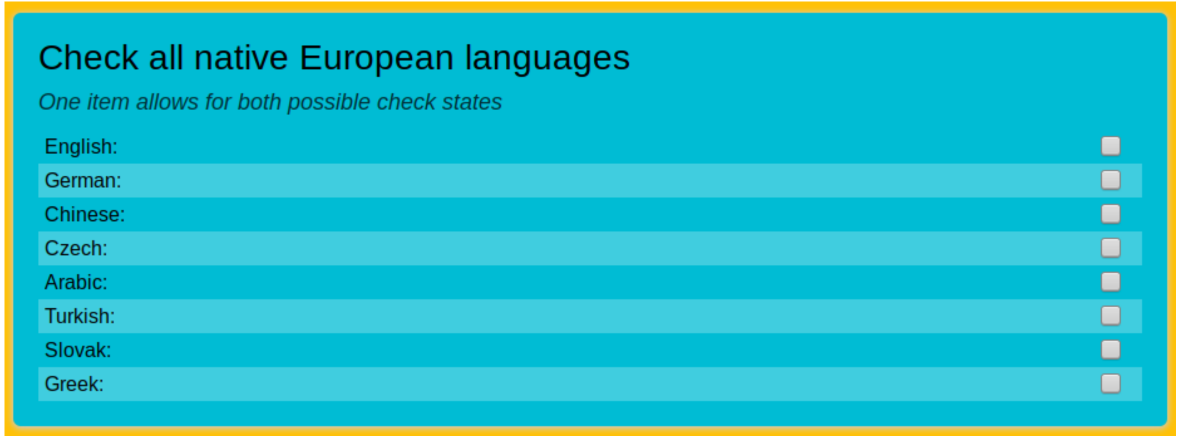
Ukázka pro tuto metodu je k dispozici zde:
http://kraken.pedf.cuni.cz/~kozubm/dt/example/query-examples/radio.html

Obrázek 7.2: Otázka s výběrem jedné odpovědi

Otázka s výběrem více odpovědí

V této kategorii se využívá checkbox elementů, které povolují dotazovanému zvolit žádnou nebo libovolné množství odpovědí. Podobně jako v případě výběru jedné možné odpovědi se i zde pro validaci využívá booleovských hodnost true a false.

Ukázka pro tuto metodu je k dispozici zde:
http://kraken.pedf.cuni.cz/~kozubm/dt/example/query-examples/checkbox.html

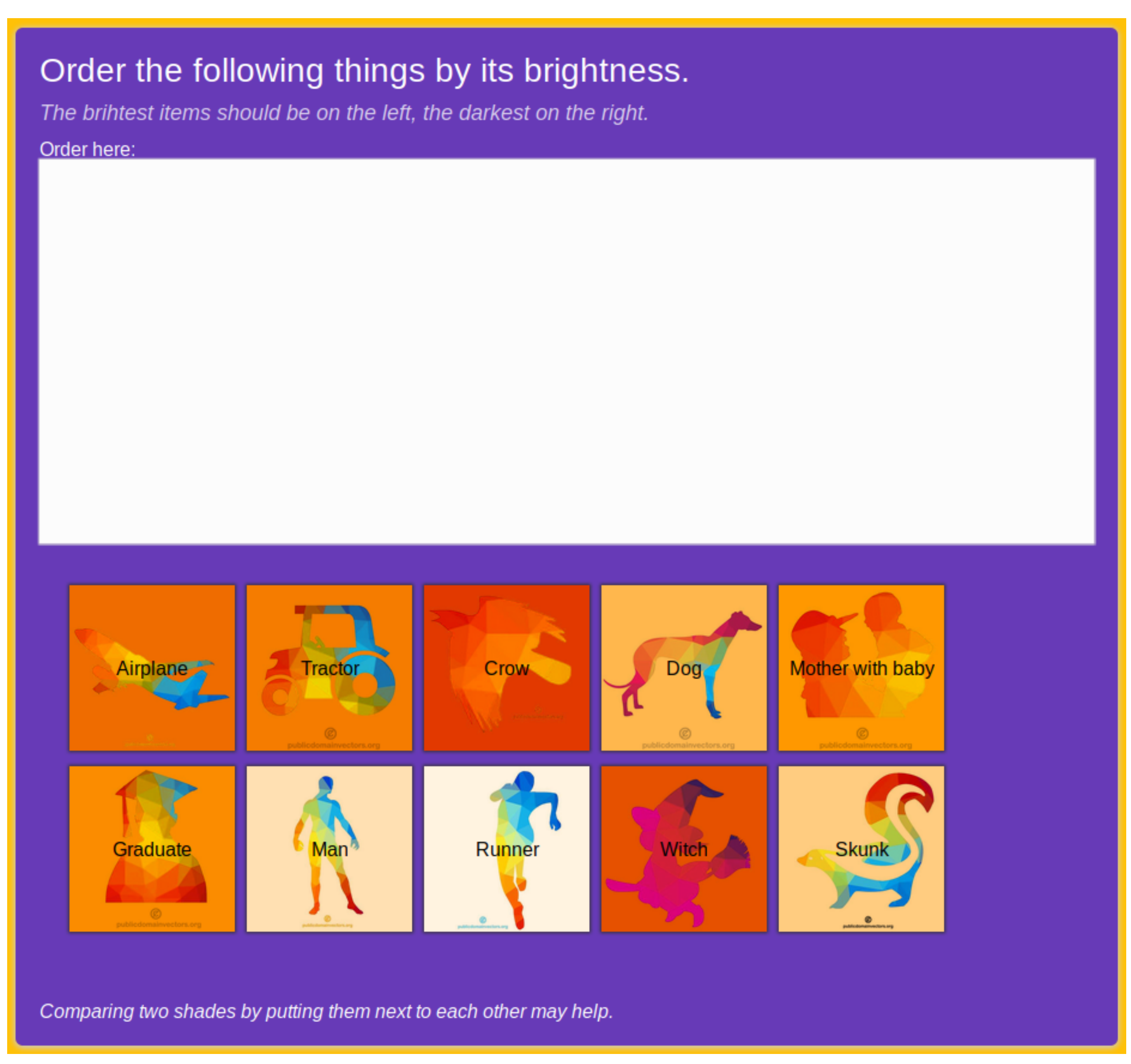


Obrázek 7.3: Otázka s výběrem více odpovědí

Uspořádací otázky

V případě otázek uspořádacích je jako vstup použito umístění přidružených možností nad jediným vstupním polem. Ty se zapisují v pořadí zleva doprava či shora dolů v závislosti na zvoleném typu konkrétní otázky. Validace probíhá na základě textových řetězců nebo regulárních výrazů uvedených v poli v řadě za sebou, tzn. ve stejném pořadí, v jakém mají být odpovědi vloženy testovaným subjektem.

Ukázka pro tuto metodu je k dispozici zde:
http://kraken.pedf.cuni.cz/~kozubm/dt/example/query-examples/order.html

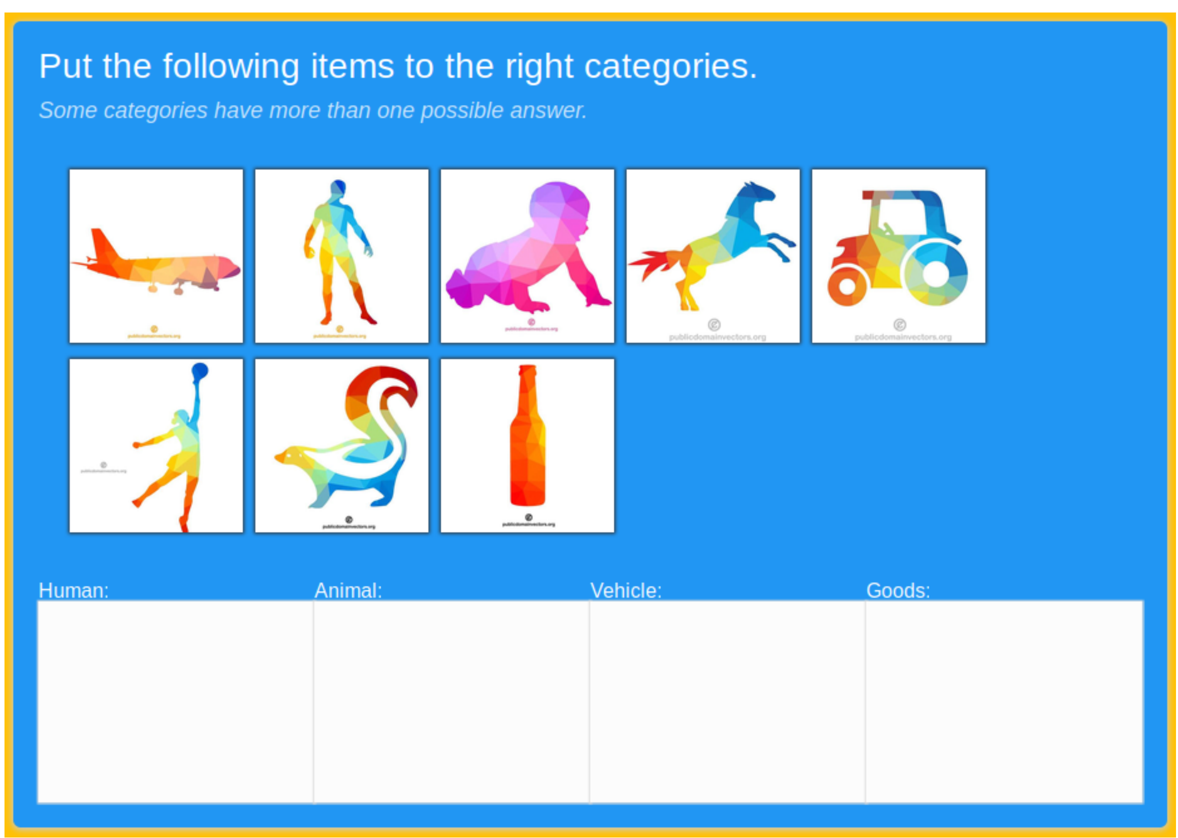


Obrázek 7.4: Uspořádací otázka

Přiřazovací otázky

U tohoto typu se využívá zadavatelem testu zvoleného počtu vstupních polí. Do nich se postupně, obdobně jako v uspořádacích otázkách, přetahují jednotlivé možnosti. Zde se již neporovnávají jednotlivé pozice uvnitř konkrétního pole, ale v rámci procesu validace se pouze ověřuje, zda je možnost nebo více možností přesunuto do správného pole. Stejně jako u výše uvedené otázky je i zde validace vázána na porovnávání dvou textových řetězců nebo textového řetězce a regulárního výrazu.

Ukázka pro tuto metodu je k dispozici zde:
http://kraken.pedf.cuni.cz/~kozubm/dt/example/query-examples/assign.html



Obrázek 7.5: Přiřazovací otázka

Škálovatelnost všech uvedených forem otázek je poměrně velká. Napomáhá tomu jak variabilita možných atributů, které lze ke každé otázce při inicializaci přiřadit, tak i poměrně mocná validační základna.

7.4.4 Atributy otázek

Každé otázce je možné předat některý z univerzálních atributů:

- id, které nemusí být univerzální, neboť to si zajišťuje sama knihovna
- textový titulek (title), jež se použije jako nadpis otázky
- podtitulek (subtitle)
- chybovou hlášku (error), která se má zobrazit v případě, že uživatel zadal odpověď špatně a pokud je v testu povolena oprava chyb před odesláním
- text, HTML či DOM element, který se má vložit do otázky (before, after)
- kaskádové styly dané otázky (css)

Některé unikátní či různě předávané parametry na základě typu otázky Kromě toho se pro každou otázku konkrétního typu vztahují ještě další možná pravidla, která umožňují funkcionalitu dále přizpůsobit.

Pro jednopoložkové otázky typu text a uspořádací otázky se při definici přímo uvádí také kontrolní pravidla (checkRules), jež musí vyhovovat struktuře, které QueryValidator rozumí. Pro připomenutí se jedná o textové hodnoty, boolean hodnoty, RegExp výrazy a pole těchto hodnot včetně kombinací různých typů.

Uspořádací otázky dále vyžadují definici přesunutelných možností skrze atribut options. Ten obsahuje pole objektových struktur s klíči:

- id — parametr, který se následně využije pro validaci
- text — hodnota, který se vypíše do oblasti tahací možnosti
- css — který umožní definovat pro danou možnost vlastní inline kaskádové styly

Ostatní vícepoložkové otázky mají některé parametry definovány skrze speciální klíč items a pro přiřazovací otázky též options, jejichž struktura i význam opakovaných prvků jsou stejné jako v případě uspořádací otázky. Pole items obsahuje stejně jako options seznam objektů, které mohou mít definovány hodnoty:

- label — obsahuje textový popisek konkrétního vstupního pole
- id

- checkRules — kontrolní pravidla, jejichž definice je stejná jako u jednopoložkových otázek
- arrayMatch — definuje, jedná-li se o otázku, kde musí být alespoň jedna z vyplněných položek správná nebo zda musí splňovat všechny požadavky
- error — chybová hláška
- css — opět definuje kaskádové styly konkrétní položky

Otázky využívající možností, tj. uspořádací a přiřazovací otázky, dále mohou akceptovat parametr optionsOrder, který definuje, zda se nejprve vykreslí vstupní pole a pod ním seznam možností nebo naopak. Uspořádací otázky navíc dokáží posuzovat pozice ve vertikálním či horizontálním směru, což lze definovat klíčem direction a textovou hodnotou „horizontal" nebo „vertical".

7.4.5 Plugin pro parsování HTML kódu

Protože využití knihovny vyžaduje intenzivnější práci s JavaScriptem, jedná se o ne zcela vhodné řešení pro rychlou tvorbu testů a ne příliš adekvátní řešení pro ty, kteří JavaScriptu dostatečně nerozumí.

Z toho důvodu jsem se rozhodl vytvořit dodatečný kód pro transformaci HTML do formátu, kterému knihovna rozumí. Její funkcionalita je sice lehce omezená, tyto restrikce jsou ovšem způsobeny především tím, že při tvorbě testu tímto způsobem uživatel spoléhá pouze na HTML. Předpokládá se, že uživatel v takovém případě nepíše vlastní JavaScriptový kód. Není tak například možné skrze tento doplněk definovat vlastní funkci, která se má zavolat po odeslání formuláře, či si zvolit některou z nativně nepodporovaných knihoven pro tahací funkcionalitu.

Důvod, proč jsem se rozhodl tento kód držet odděleně od zbytku knihovny a vytvořit z něj tedy jakýsi plugin, spočívá v odlišnosti naprosto základních pochodů těchto dvou separátních celků. Zatímco má knihovna vychází primárně z dat předávaných JavaScriptové části a na základě nich generuje příslušný HTML kód, v případě kódu pro navázání funkcionality na existující HTML se jedná o zcela opačný postup. Protože mým záměrem bylo ponechat knihovnu tak, jak funguje nyní, musel jsem vytvořit mezikus, který z HTML kódu dokáže vytvořit datovou strukturu, kterou si knihovna může přečíst a na základě ní znovu sestavit test. Jakmile si plugin zjistí, čeho chtěl autor HTML kódu dosáhnout, smaže jím vytvořený soubor otázek a nechá knihovnu vytvořit si na jeho místě patřičný nový formulář s odpovídajícími položkami. Závisí pak již jen na rozsahu pluginu, co vše bude z původního HTML kódu převedeno nebo zachováno.

Kód, který jsem k tomuto účelu vytvořil, je cílen spíše jako ukázka toho, jak něco podobného provést. Je potenciálním vývojářům k dispozici, aby si jej upravili, rozšířili či se jím pouze inspirovali při tvorbě vlastního rozšíření.

Kompletní zdrojový kód pluginu je přiložen v přílohách níže na straně 69.

7.4.6 Automatické testování

Pro automatické testování se ve vlastní knihovně přímo vybízely datové třídy QueryDataBlock a QueryValidator. Obě zmíněné třídy byly tedy vyvíjeny převážně testy řízeným vývojem a testovány automatickými unit testy.

K samotnému testování jsem použil framework Jasmine, jehož nespornou výhodou je především velmi snadná implementace. K tomu, aby jej bylo možné použít, totiž stačí z repozitáře projektu na GitHubu stáhnout archiv pojmenovaný „jasmine-standalone“, uložit jeho rozbalenou strukturu do zvoleného adresáře a potřebné JavaScriptové soubory pouze připojit k testovací stránce. Jasmine nezávisí na žádných dalších knihovnách ani frameworcích, čímž se stává velmi lehkým a univerzálním testovacím nástrojem i pro prostředí mimo webový prohlížeč. [28]

7.5 Tvorba dokumentace

Pro dokumentaci vlastního kódu je možné zvolit si prakticky mezi dvěma cestami. Ta první, více rudimentární, je psaní komentářů ke zdrojovému kódu bez dodržování nějaké standardizované struktury. To má ale za následek mj. výrazně zhoršenou čitelnost komentářů.

Kromě toho, je-li potřeba vytvořit také nějaký přehled dokumentovaného kódu (tedy ideálně všech jeho tříd a metod či funkcí), vyžaduje to další úsilí ve formě ručního zpracování takového dokumentu. Stejně tak v případě větších změn v kódu je potřeba tuto změnu reflektovat nejen v komentáři, ale také ve vedlejší dokumentaci.

Z toho důvodu se jeví jako lepší cesta použít k dokumentaci některý z již existujících nástrojů a přizpůsobit jim i formu, jakou se komentáře v kódu zapisují. Takové nástroje potom nejen drží vývojáře v určitých kolejích tak, aby psal komentáře důkladně, ale postarají se též o následný export, nejčastěji ve formě dokumentační webové stránky.

[28] Jasmine: Simple JavaScript testing framework for browsers and node.js. *GitHub* [online]. San Francisco: Gregg Van Hove, 2017 [cit. 2017-03-02]. Dostupné z: https://github.com/jasmine/jasmine

7.5.1 Existující nástroje pro dokumentaci projektů psaných v JavaScriptu

Mezi známější nástroje patří například JSDoc, YUIDoc, Doxx a Docco a z pohledu uživatele, tedy toho, kdo tento nástroj používá pro svůj kód, se liší v několika ohledech. Jedním z rozdílů je odlišná syntaxe komentářů, kde například Docco využívá jakýchkoliv komentářů a zobrazuje je prakticky jen vedle výpisu skutečného kódu. Jiné nástroje jako YUIDoc nebo JSDoc využívají komplexnějších komentářů, na jejichž strukturu jsou zvyklí také vývojáři PHP — PHPDoc a Javy — Javadoc. Díky tomu dokáží s daty mnohem lépe pracovat. Na základě určitých značek dokáží rozpoznat, o co se jedná, a díky tomu mohou vygenerovat také přehledný a informativní dokument.

Rozdíly lze vidět také ve formátu výstupu, ne jen v již zmíněné informační struktuře, která je závislá na tom, jaká data dokáže dokumentační nástroj rozpoznat, ale také na samotném vzhledu. YUIDoc navíc ve výchozím stavu podporuje vyhledávání, díky čemuž je možné rychle najít jakoukoliv proměnnou, třídu či metodu. JSDoc nemá vyhledávání ve standardním režimu povoleno, nicméně je možné ho také zapnout.

Kromě toho je v nástroji JSDoc také možné generovat další přídavné stránky jako návody požití anotací `@tutorial`, která se zapisuje do komentářů ve zdrojovém kódu a vede na soubor obsahující Markdown syntaxi.

JSDoc dále výrazně vyniká ještě v jedné oblasti a to je parsování kódu. Zatímco ostatní nástroje spoléhají pouze na komentáře, z nichž následně vychází i generátor stránek dokumentace, JSDoc dokáže rozpoznat, co kód dělá, a díky tomu zamezit situaci, kdy obsah komentáře nesedí s tím, co je v kódu.

Z výše uvedeného se jevil právě nástroj JSDoc jako nejlepší volba. A i když pro dokumentaci knihovny s jistotou nevyužiji všechny jeho funkce, jedná se o nejlepší nástroj, který dokáže eliminovat vlastní chyby v dokumentaci, a stejně tak o nejlepší volbu z pohledu perspektivy do budoucna. (Modak, 2016)

7.5.2 Dokumentace v JSDoc

Popis prvků zdrojového kódu probíhá, jak již jsem zmínil, ve formě komentářů do zdrojového kódu aplikace. Vzhledem ke komplexnosti většiny popisů a také odlišnosti od klasických komentářů ale nestačí klasický jednořádkový komentář, v JavaScriptu uvozený dvěma lomítky: `// komentář`. Je potřeba využít potencionálně víceřádkového komentáře, navíc v syntaxi JSDoc s duplicitní * v úvodu, tedy: `/** komentář*/`.

Pro definici třídy uvozené klíčovým slovem `class` postačí tento jednoduchý zápis:

Zdrojový kód 16 Ukázka dokumentace třídy v nástroji JSDoc

```
1 /** Definice třídy. */
2 class NázevTřidy { }
```

Jednotlivé funkce, metody a také konstruktor se poté komentují následujícím způsobem:

Zdrojový kód 17 Ukázka dokumentace funkcí a metod v nástroji JSDoc

```
1 /**
2 * Definice metody, funkce nebo konstruktoru.
3 * @param {typ} nazev_promenne - Popis k čemu proměnná slouží.
4 * @return {typ} - Popis toho, co se vrací.
5 */
6 run(nazev_promenne) {
7         return this.value;
8 }
```

Argumentů může mít funkce více a případné další se zapisují na novém řádku, vždy uvozeny klíčovým slovem `@param`. Pomlčka mezi popisem proměnné z pohledu typů a názvu a slovním popisem není v komentáři důležitá a hraje roli pouze v přehlednosti samotného komentáře ve zdrojovém kódu, nikoliv ve vygenerované dokumentaci.

Mezi pseudotypy, které se využívají v dokumentaci, patří například číslicový — `number`, textový — `string` a logický — `boolean`. Nenazývají se přímo typy, neboť nejsou součástí definice programovacího jazyka, ale slouží pro zjednodušení dokumentace. (Vrána, 2010)

Pokud může proměnná nabývat více typů hodnot, oddělují se svislou čarou |: `{(number|boolean)}`. Může-li proměnná či návratová hodnota nabývat mnoha typů, je možné si zápis zjednodušit zápisem typu jako: `{mixed}`. [29] [30]

Pro samotné generování dokumentace jsem v kořenovém adresáři projektu vytvořil shellový skript, který se automaticky postará o spuštění nástroje jsdoc se zvolenými, pro projekt specifickými, argumenty. Skript počítá s instalací jsdoc na standardní cestě při instalaci skrze správce balíčků npm do domovského adresáře, tzn., že pro běh na jiné instalaci bude vyžadovat úpravu. Stejně tak je nutné jej spustit z kořenového adresáře projektu, aby dokumentační nástroj dokázal zdrojové soubory pro dokumentaci nalézt.

Kompletní dokumentace k projektu je součástí repozitáře git a je k dispozici na internetové adrese: https://zubozrout.github.io/DT/index.html

[29] @use JSDoc [online]. JSDoc 3, c2011-2014 [cit. 2017-02-15]. Dostupné z: http://usejsdoc.org/

[30] JSDoc 3: n API documentation generator for JavaScript. *GitHub* [online]. San Francisco: Michael Mathews, 2016 [cit. 2017-02-12]. Dostupné z: https://github.com/jsdoc3/jsdoc

7.5.3 Jazyk

Jazyk komentářů a dalších textů by měl být zvolen podle toho, jaká skupina lidí bude se zdrojovým kódem nejspíše dále pracovat a kdo bude číst vytvořenou dokumentaci. V dnešní době ale ve většině případů přichází v úvahu primárně univerzální angličtina, která se v technickém světě počítačů objevuje takřka všude. (Vrána, 2010) Protože vlastní projekt hostuji na serveru GitHub a nechci tento projekt omezovat pouze na česky mluvící uživatele, i já jsem se rozhodl psát veškeré komentáře a dokumentaci k projektu anglicky.

7.6 Načítání knihovny

Při vývoji a pro testování jsem potřeboval vyřešit problém, jak všechny potřebné soubory knihovny načíst tak, aby software fungoval jako celek. Vhodným řešením pro můj problém se jevila tvorba vlastního jednoduchého nástroje pro načítání externích JavaScriptových souborů. Na něj jsem dále mohl spolehlivě napojit další kód, který se spustí po stažení všech definovaných skriptů a který se stará o samotnou tvorbu testu.

Třída LibraryLoader obsahuje metodu load, která vrátí Promise a může po načtení všech jednotlivých skriptů přímo zavolat callback funkci, je-li metodě v argumentu předána.

Zdrojový kód 18 Metoda load třídy LibraryLoader

```
load(callback) {
	if(this.libraryStructure.directory && this.libraryStructure.libraries) {
		let loadPromises = [];
		for(let i = 0; i < this.libraryStructure.libraries.length; i++) {
			loadPromises.push(this.loadSingle(this.libraryStructure.libraries[i]));
		}

		return Promise.all(loadPromises).then(() => {
			if(typeof callback !== typeof undefined && typeof callback === "function") {
				callback();
			}
			return "All scripts loaded";
		});
	}
	else {
		return Promise.reject(new Error("No scripts to load."));
	}
}
```

Pro samotnou práci s knihovnou jsem poté na základě kladné odpovědi Promisu již jen spustil potřebný kód pro inicializaci formuláře. Zjednodušeně si to lze představit takto:

Zdrojový kód 19 Ukázka využití metody load

```
libraryLoaderInstance.load().then((response) => {
	vytvoritDidaktickyTest();
});
```

Ač se nejedná o přímou součást knihovny, tvorba nástroje pro načítání doplňkových souborů byla technicky zajímavá a výsledný kód vhodný nejen pro testovací účely současně vznikající knihovny. S přehledem mi zajišťuje načtení všech potřebných souborů, včetně hlášení chyb při problémech s jejich načtením, a přitom se jedná o relativně přehledný a ucelený kód.

Kompletní zdrojový kód třídy LibraryLoader je k nahlédnutí v příloze tohoto dokumentu na straně 68.

7.7 Sjednocení knihovny do jednoho souboru a její minifikace

Protože se ale v produkčním prostředí z důvodů datového toku a latence příliš nehodí výše zmíněné načítání jednotlivých souborů knihovny, bylo potřeba zdrojový kód sjednotit a ideálně také minimalizovat. Nicméně nástrojů, které by podporovaly ECMAScript verze 6 není mnoho, a protože jsem pro jednoduchost požadoval spouštění z příkazové řádky, rozhodl jsem se použít nástroj UglifyJS 2, který splňuje všechny požadované vlastnosti. [31] [32]

Pro automatické generování jsem si vytvořil v kořenovém adresáři projektu skript, který pro minifikaci a sjednocení používá následující příkaz:

```
~/node_modules/.bin/uglifyjs $ZDROJ/*.js > $NAZEV
```

Obdobně jako v případě skriptu pro tvorbu dokumentace i zde platí, že cesty odpovídají instalaci nástroje UglifyJS do domovského adresáře a konkrétnímu projektu. Navíc je nutné skript spouštět z kořenového adresáře knihovny.

Protože bylo potřeba zachovat názvy všech metod a proměnných v čitelné podobě tak, aby nenastal problém při snaze volat tyto metody i z vnějšku, bez nutnosti pamatovat si, co který zástupný symbol znamená, nezapínal jsem pro minifikaci jejich zkracování.

Kromě toho jsem se následně, po provedení minifikace kódu, rozhodl přidat do svého skriptu příkaz pro odstranění duplicitních definic striktního módu `"use strict";`. Tato deklarace je totiž uvedena v úvodu každého zdrojového souboru a po jejich sjednocení do jednoho celku se její opakování stává redundantní. Příkaz, který je součástí skriptu pro minifikaci a spouští se hned po ní, využívá jednoduchý regulární výraz a nástroj sed:

```
sed -i -- 's/"use strict";//2g' $NAZEV
```

[31] Uglify-js-harmony: JavaScript parser, mangler/compressor and beautifier toolkit. *Npm* [online]. Oakland (California): npm, 2017 [cit. 2017-02-28]. Dostupné z: https://www.npmjs.com/package/uglify-js-harmony

[32] Minify tool that can be executed through terminal. In: *Ask Ubuntu* [online]. New York (NY): Stack Exchange, 2016 [cit. 2017-02-28]. Dostupné z: http://askubuntu.com/a/827800/25771

7.8 Tvorba ukázkového testu

Cílem ukázkového testu bylo uživatelům knihovny představit všechny typy otázek a také různé možnosti, které knihovna podporuje.

7.8.1 JavaScriptem definovaný test

Jako základní ukázku jsem zvolil knihovnou nativně podporovaný způsob komunikace, tedy případ, kdy autor testu musí pracovat s JavaScriptem. Z jeho pohledu je ale tvorba testu velmi snadná a navíc mnohem škálovatelnější než při použití čistého HTML. V základní podobě se jedná prakticky jen o instanciování třídy FormsHolder a na ní vytvořeném testu:

Zdrojový kód 20 Inicializace třídy FormsHolder a vytvoření formuláře

```
1 let formsholder = new FormsHolder();
2 let ukazkovyTest = formsholder.createForm( data pro příslušný
     test );
```

Protože metoda createForm třídy FormsHolder vrátí nově vytvořenou instanci třídy Form, mohu ji dále využít pro inicializaci otázek:

Zdrojový kód 21 Obecný způsob tvorby otázek

```
ukazkovyTest.newQueryItem( data pro konkrétní otázku );
```

Těch je do testu možné přidat knihovnou neomezené množství, takže počet otázek může být velmi malý i početný. Stejné metody se využívá pro všechny typy otázek a záleží pouze na předaných datech, o jakou konkrétní otázku se bude jednat.

Pro ukázku jsem využil všech podporovaných typů otázek a tam, kde mi přišlo být zajímavé ukázat i další možnosti určitého typu, i jejich podvariant.

7.8.2 HTML definovaný test

Jako druhou ukázku jsem zvolil v HTML zapsaný test, který využívá pluginu, jež překládá HTML do formy, kterou dokáže knihovna zpracovat. Tento plugin není přímou součástí knihovny a je nutné ho ke stránce dodatečně připojit. Jeho výhodou je, že se po načtení stránky spustí sám a zadavatel testu tak nemusí s JavaScriptem vůbec pracovat.

Plugin automaticky zpracovává každý nalezený formulář na stránce, který má nastaven atribut `data-dt="yes"`, a pro převod otázek do svého formátu vyžaduje následující strukturu:

Zdrojový kód 22 Příklad HTML pro parser HTML kódu

```
1 <h3>What is your name?</h3>
2 <dtwrapper data-dttype="text">
3         <input id="name" data-dtlabel="Name" data-dtcheckrules
             ="/^[a-zA-Z]+$/u" required>
4 </dtwrapper>
```

Nadpis otázky může být definován výše uvedeným způsobem (libovolným h1–6 elementem) nebo v atributu tagu dtwrapper: `data-title=""`.

Při tvorbě ukázek jsem se snažil dodržet paritu obou formulářů, tedy aby oba podporovaly stejné typy otázek. V případě HTML definice jsem ne zcela dodržel vzhled některých elementů, neboť jsem v JavaScriptové ukázce mohl snadněji využívat komplexnějších inline kaskádových stylů, které tolik nebránily přehlednosti kódu definujícího ukázkový kvíz.

7.9 Funkčnost v různých webových prohlížečích

K ověření funkčnosti na různých platformách a v různých webových prohlížečích jsem využil několik sestav s odlišnými operačními systémy. Na nich jsem prošel ukázkové testy a otestoval funkcionalitu přetažitelných objektů včetně jejich správného přiřazování do adekvátních buněk. Dále jsem také spustil automatické testy, čímž jsem potvrdil, že ani různé interprety jazyka JavaScript takto testované třídy nerozbijí. K tomu bylo nutné nahrát potřebné soubory na veřejně dostupný server, kam jsem mohl pohodlně přistupovat ze všech pro tento účel využívaných zařízení.

7.9.1 Desktop

Vlastní testování jsem prováděl na třech nejrozšířenějších operačních systémech. [33]

Windows

Na platformě Windows 10 jsem knihovnu úspěšně otestoval v prohlížečích Microsoft Edge 38, Firefox 52 a Google Chrome 57.

[33] Statistiky počtu přístupů různých desktopových operačních systémů podle NetMarketShare: https://www.netmarketshare.com/operating-system-market-share.aspx

macOS

V operačním systému macOS Sierra jsem běh na knihovně postaveném didaktickém testu ověřil v prohlížeči Safari 10.1.

Linux

Jako desktopovou distribuci jsem k testování využil instalaci Ubuntu, respektive Xubuntu, a webové prohlížeče Firefox 52 a Chromium 56.

7.9.2 Mobilní platformy

Na telefonech jsem knihovnou vytvořené formuláře vyzkoušel v následujících operačních systémech:

Android

K testování jsem měl k dispozici mobilní zařízení se systémem CyanogenMod 13, který je postavený na Androidu 6.0. Knihovna si v této verzi systému nerozumí s nativním prohlížečem Androidu, ale funguje s webovým prohlížečem Google Chrome 57. Dále jsem knihovnu s úspěchem spustil v prohlížeči Firefox 52.

Díky nástroji Anbox jsem dále dokázal potvrdit kompatibilitu také s nativním prohlížečem v Androidu 7.1.1, který využívá novější vykreslovací jádro. [34]

iOS

S platformou iOS 9 není knihovna kompatibilní, ale je kompatibilní s prohlížečem Safari na iOS 10. Protože Google Chrome na iOS využívá stejné jádro jako prohlížeč Safari a liší se pouze v uživatelském rozhraní, není knihovnu na iOS 9 možné zprovoznit ani s tímto prohlížečem. (Gruber, 2012)

Ubuntu Phone

K testování jsem dále využil operační systém Ubuntu Phone a nativní webový prohlížeč Ubuntu Web Browser, který využívá stejné vykreslovací jádro jako prohlížeče Google Chrome a Chromium. (Coulson, 2013) I zde byl chod knihovny bezproblémový.

7.9.3 Podpora starších prohlížečů

Některé prohlížeče nebo jejich starší verze nejsou knihovnou kvůli použitému ECMASrciptu 6 přímo podporovány, avšak částečné kompatibility lze dosáhnout s vyu-

[34] Anbox. *GitHub* [online]. San Francisco: GitHub, 2017 [cit. 2017-04-14]. Dostupné z: https://github.com/anbox/anbox

žitím překladače Babel, jež dokáže moderní kód přetransformovat do starší verze. [35] [36] Takto je například možné zprovoznit knihovnu v nativním prohlížeči Androidu 6.0, ovšem ani takto upravený kód nedokáže knihovnu zcela zprovoznit v prohlížeči Internet Explorer.

Babelem přetransformovaný kód včetně scriptu pro jeho generování je součástí repozitáře projektu. Nástroj Babel se s potřebnými parametry spouští následovně:

```
~/node_modules/.bin/babel source -d babelized \
--presets es2015-ie
```

7.9.4 Ovládání testů na menších obrazovkách

Z testování ovšem vyplynulo, že na mobilních zařízeních mohou nastat komplikace při vyplňování některých otázek, především těch, které obsahují přetažitelné objekty. Pro jejich přesun nezbývá na malé obrazovce telefonu při zachování větších velikostí elementů dostatečné množství místa. Posunování je tak nutné realizovat na několik iterací a během toho posunovat též celou stránkou. Velmi však záleží na definovaných kaskádových stylech, které si vytváří sám autor testu, nepoužije-li ukázkové.

Z tohoto ohledu je zjevné, že bude autor testu muset v případě, že bude počítat i s podporou mobilních zařízení, otázky optimalizovat takovým způsobem, aby prostor nabízený menší obrazovkou dostačoval pro jejich pohodlné vyplnění. Je tedy vhodné některé prvky na menších obrazovkách zmenšit. Je-li potřeba seřazovat větší množství položek v jednom poli, potom je vhodné preferovat vertikálně skládané odpovědi na místo těch horizontálních, neboť se u uživatelů mobilního telefonu počítá spíše s rozložením stránky na výšku a posunem stránky ve stejném vertikálním směru.

7.9.5 Shrnutí výsledků testů

Výsledky testů kompatibility prokázaly, že je knihovna dostatečně univerzální a může být využívána na všech moderních platformách. A i když jsem vynechal množství méně používaných prohlížečů, které podporují moderní JavaScript, lze předpokládat, že ani s nimi nebude mít knihovna žádné potíže.

[35] *Env preset. Babel* [online]. Berlín: Babel, 2017 [cit. 2017-03-05]. Dostupné z: http://babeljs.io/docs/plugins/preset-env/

[36] *ECMAScript 6 compatibility table. Stuff by kangax* [online]. New York: Juriy Zaytsev, 2017 [cit. 2017-03-06]. Dostupné z: https://kangax.github.io/compat-table/es6/

7.10 Hodnocení zvolené metodiky vývoje

Ze zkušeností z vývoje a následného testování mohu říci, že metodiky vývoje knihovny, které jsem na počátku zvolil (tj. testy řízený vývoj a prototypový přístup), mi nikterak nebránily knihovnu vytvořit a dotáhnout do funkčního stavu, v němž se nyní nachází. Naopak především při vývoji základního kamene verifikace odpovědí, třídy QueryValidator, mi automatické testování velmi pomohlo v přidávání nových funkcí, aniž bych přitom něco negativně ovlivnil. Zde mi na místo zdlouhavého testování a zkoušení všech možných variací testů bohatě stačilo spustit jeden důkladně propracovaný automatický test, který mě sám informoval o nefunkčních částech.

Tento způsob vývoje jsem však použil pouze omezeně a na části, jež se pro automatické testování samy vybízely. Ostatní části knihovny, ač by z většiny byly snadno automaticky testovatelné, je možné ověřovat též jednoduchým průchodem ukázkového testu s otázkami. Důvodem je především skutečnost, že zatímco testovacích možností lze v tak univerzálním prostředí, jaké jsem v knihovně implementoval, definovat velké množství, typů otázek je jen pár a pro ověření, zda se zobrazují a chovají tak, jak mají, postačí ukázkový test vyzkoušet. Jistá část s vizuální stránkou věci spojených problémů může být také zapříčiněna neadekvátními kaskádovými styly, které se definují nezávisle na samotném kódu knihovny, a pro jejichž testování jsou mnou využité unit testy nevhodné.

Pro zbylé třídy jsem se tak rozhodl automatické testování neimplementovat a spoléhat na prototypový přístup postupného vývoje. Tvořil jsem tedy kód v mnoha iteracích a v každé z nich ručně znova a znova testoval správnou funkčnost. Nicméně, hlavní kostru knihovny jsem, jakmile byl odrazový můstek pro další vývoj se základní funkcionalitou hotov, příliš neměnil a prováděl na ní pouze relativně minoritní úpravy. Jakákoliv chyba v kostře by se navíc při průchodu ukázkovým testem s vysokou pravděpodobností projevila. Menší obslužný kód, který jsem na tuto strukturu přidával, by neměl v případě změn negativně ovlivnit zbytek knihovny.

Ač mně tedy zvolené metodiky vyhovovaly a nenarážel jsem kvůli nim na zásadní obtíže, mohl jsem si mírně ulehčit testovaní softwaru tak, že bych vytvořil automatické testy pro větší část knihovny. Celkem bez potíží by bylo možné testovat zbylé třídy a nechat pouze vizuální stránku věci na manuálním hodnocení. Na druhou stranu, ruční procházení ukázkových formulářů by bylo, bez speciálních testů schopných automatizovat kontrolu grafického rozhraní, i nadále potřeba. [37]

[37] *SeleniumHQ: Browser Automation* [online]. Chicago: Selenium, 2017 [cit. 2017-03-08]. Dostupné z: http://www.seleniumhq.org/

8 Závěr

Během této bakalářské práce se mi podařilo prozkoumat existující formy otázek didaktických testů, vybrat adekvátní typy otázek pro prostředí webu a vyvinout knihovnu, která může učitelům, lektorům a ostatním zájemcům usnadnit tvorbu interaktivních didaktických testů. Obrovskou výhodou této knihovny je schopnost tvorby otázek, při kterých uživatel využívá myš nebo v případě dotykových obrazovek prst pro přiřazování a uspořádávání odpovědí. Ty se hodí především pro menší děti, jelikož je lze kombinovat i s vlastními styly a obrázky. Definice takovýchto otázek je přitom poměrně snadná a implementace pro zadavatele testu mnohem snazší s využitím této knihovny.

Další předností vytvořené knihovny je síla validačního nástroje, která autorovi testu zpřístupňuje celou škálu možností, jak definovat správné odpovědi. Povolenými hodnotami jsou textové řetězce, logické hodnoty i regulární výrazy. Tyto parametry jsou navíc volitelně předávány v poli a může jich tedy být i více. Dále je možné mixovat různé typy. A kromě toho také v případě přiřazovacích odpovědí nastavit, zda stačí, aby byla zadána jen jedna správná odpověď nebo aby odpovědi vyhovovaly všem požadavkům.

Protože se ale ověřování správnosti odehrává pouze na straně klienta, což je část, kterou se tato práce zabývala, není možné zajistit, že v průběhu testu nedojde k podvádění. Z tohoto pohledu tedy platí, že pro lidi znalé práce s počítačem takto vytvořené testy obsahující validační část nejsou vhodné a je tak potřeba implementovat kontrolu na straně serveru. S tím ovšem knihovna počítá a sama umožňuje odesílání vyplněných dat pro pozdější zpracování na definovanou adresu.

Při tvorbě vlastní knihovny jsem se rozhodl využít jak prototypový přístup, tak testy řízený vývoj. Přičemž jsem především z časových důvodů preferoval především první uvedenou metodiku.

Výsledné formuláře jsou velmi přizpůsobitelné z hlediska funkčnosti a také z hlediska vzhledové rozmanitosti. Díky tomu, že kaskádové styly jsou čistě v rukou tvůrce formuláře, může být výsledkem strohý monochromatický čistě textový formulář stejně jako pestrobarevný obrázkový kvíz.

Jistým nedostatkem knihovny, který souvisí s faktem, že si celé formuláře generuje sama, je vedle komplexnosti také skutečnost, že nepodporuje veškeré otázky definované klasickým HTML. Také z těchto důvodů je zavedena alespoň podpora pro běžnou textovou otázku a zaškrtávací možnosti. Zvolený způsob vývoje však přinesl pozitiva v jednodušší správě kooperace všech částí knihovny mezi sebou a také, v případě nativního využití knihovny, v přehlednější a kompaktnější definici testových otázek. V rámci budoucího vývoje je ovšem možné chybějící formy otázek mezi podporované zařadit.

Dalším důvodem pro vlastní definici všech otázek a jejich generování z prostředí JavaScriptu je usnadnění možnosti využití knihovny v jiných aplikacích a schopnost tvorby testu přímo z nich. Takto může knihovna poskytovat jednotné API pokrývající podporované typy otázek. Podobně by se jednalo o výhodu také v situaci, kdy by měla být knihovna využita mimo prostředí webu, kde HTML model chybí. Přizpůsobení knihovny by v takovém případě nemělo přinést příliš mnoho obtíží.

Lze konstatovat, že se mi v úvodu stanovených cílů podařilo dosáhnout a obsah tématu naplnit.

Zdrojový kód knihovny je k dispozici na přiloženém CD. Jedná se o kopii repozitáře git, naposledy aktualizovanou dne 3. 4. 2017.

9 Seznam použité literatury

COULSON, Chris. Introducing Oxide. In: *Chris Coulson's blog* [online]. London: Chris Coulson, 2013 [cit. 2017-04-14]. Dostupné z: http://www.chriscoulson.me.uk/blog/?p=196

DAS MODAK, Kaustav. Choosing a JavaScript Documentation Generator – JSDoc vs YUIDoc vs Doxx vs Docco. In: *FusionBrew* [online]. Kolkata (Indie): FusionBrew, 2016 [cit. 2017-02-15]. Dostupné z: http://www.fusioncharts.com/blog/2013/12/jsdoc-vs-yuidoc-vs-doxx-vs-docco-choosing-a-javascript-documentation-generator/

GRUBER, John. Google Announces Chrome for iPhone and iPad, Available Today. In: *Daring Fireball* [online]. Filadelfie (Pensylvánie): The Daring Fireball Company LLC., 2012 [cit. 2017-04-14]. Dostupné z: https://daringfireball.net/linked/2012/06/28/chrome-ios

HLAVA, Tomáš. Fáze a úrovně provádění testů. In: *Testování softwaru: Portál zabývající se problematikou ověřování kvality software. Manuální i automatizované testování.* [online]. Praha: Tomáš Hlava, 2011 [cit. 2017-03-22]. Dostupné z: http://testovanisoftwaru.cz/tag/integracni-testovani/

CHAUDHARY, Mukund. Importance of Software Documentation in Software Development. In: *LinkedIn* [online]. Sunnyvale (California, USA): Mukund Chaudhary, 2014 [cit. 2017-01-19]. Dostupné z: https://www.linkedin.com/pulse/20140612054919-69055394-importance-of-software-documentation-in-software-development

CHECKMAN, Jordan. Creating a JavaScript Library. In: *Jordan Checkman* [online]. Philadelphia (USA): Jordan Checkman, 2014 [cit. 2017-01-19]. Dostupné z: http://checkman.io/blog/creating-a-javascript-library/

CHRÁSKA, Miroslav. *Didaktické testy: příručka pro učitele a studenty učitelství.* Brno: Paido, 1999. Edice pedagogické literatury. ISBN 80-859-3168-0.

KANTED, Siddarth a Gautam K. What is the difference between a JavaScript framework and a library? In: *Quora* [online]. Mountain View (USA): Quora, 2014 [cit. 2017-01-21]. Dostupné z: https://www.quora.com/What-is-the-difference-between-a-JavaScript-framework-and-a-library

MAJDA, David. Knihovny vs. frameworky. In: *David Majda* [online]. Prague: David Majda, 2009 [cit. 2017-01-21]. Dostupné z: https://majda.cz/blog/265

RIESEBERG, Felix. ECMAScript 6: A Quick Intro to the Future of JavaScript. In: *Felix Rieseberg* [online]. Německo: Felix Rieseberg, 2015 [cit. 2017-02-13]. Dostupné z: https://felixrieseberg.com/ecmascript6-introduction/

SEVERIEN, Tim. Design and Build Your Own JavaScript Library: Tips & Tricks. In: *SitePoint* [online]. Collingwood (Austrálie): Tim Severien, 2016 [cit. 2017-01-19]. Dostupné z: https://www.sitepoint.com/design-and-build-your-own-javascript-library/

ŠMÍD, Vladimír. Životní cyklus informačního systému. In: *Vladimír Šmíd* [online]. Praha: RNDr. JUDr. Vladimír Šmíd, CSc., 2002 [cit. 2017-02-04]. Dostupné z: http://www.fi.muni.cz/ smid/mis-zivcyk.htm

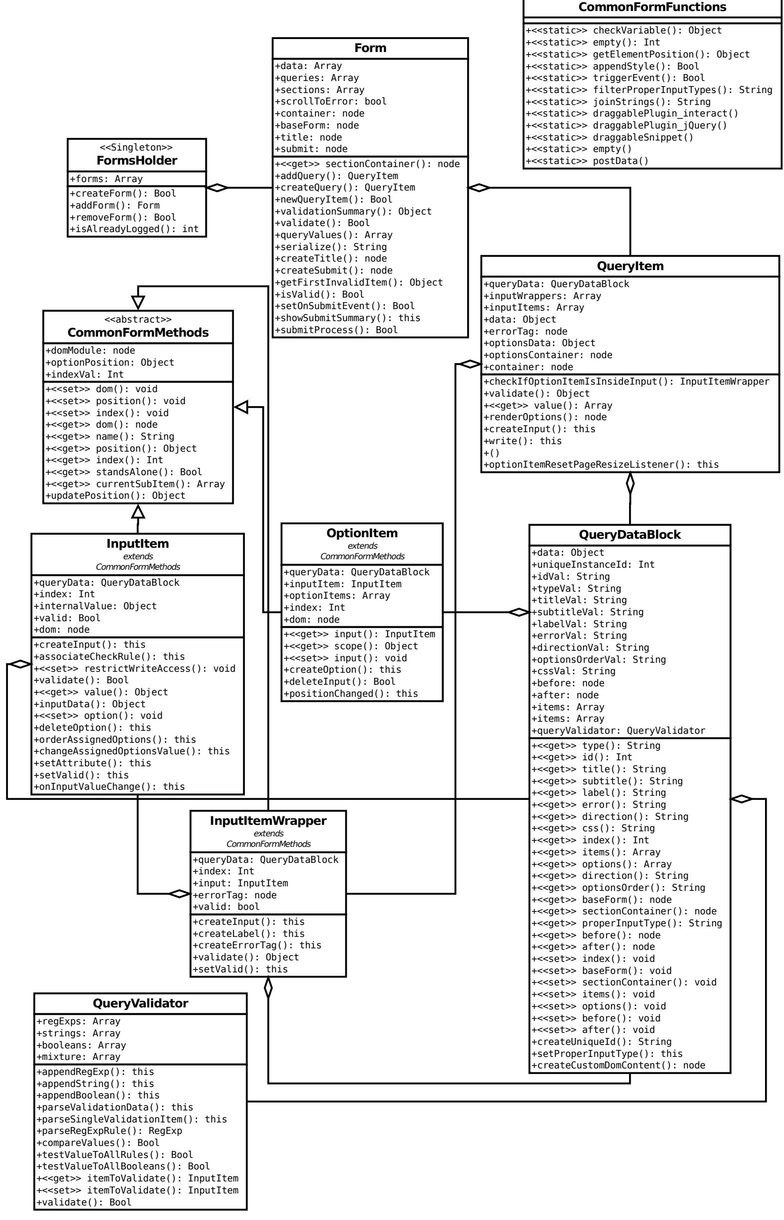
TAYLOR, Mike. Tangled Up in Tools: What's Wrong with Libraries, and What to Do about It. In: *The Pragmatic Bookshelf* [online]. Raleigh (USA): The Pragmatic Bookshelf, 2010 [cit. 2017-01-21]. Dostupné z: https://pragprog.com/magazines/2010-04/tangled-up-in-tools

TORVALDS, Linus. Tech Talk: Linus Torvalds on git. In: *YouTube* [online]. Mountain View: Google, 2007 [cit. 2017-03-20]. Dostupné z: https://www.youtube.com/watch?v=4XpnKHJAok8

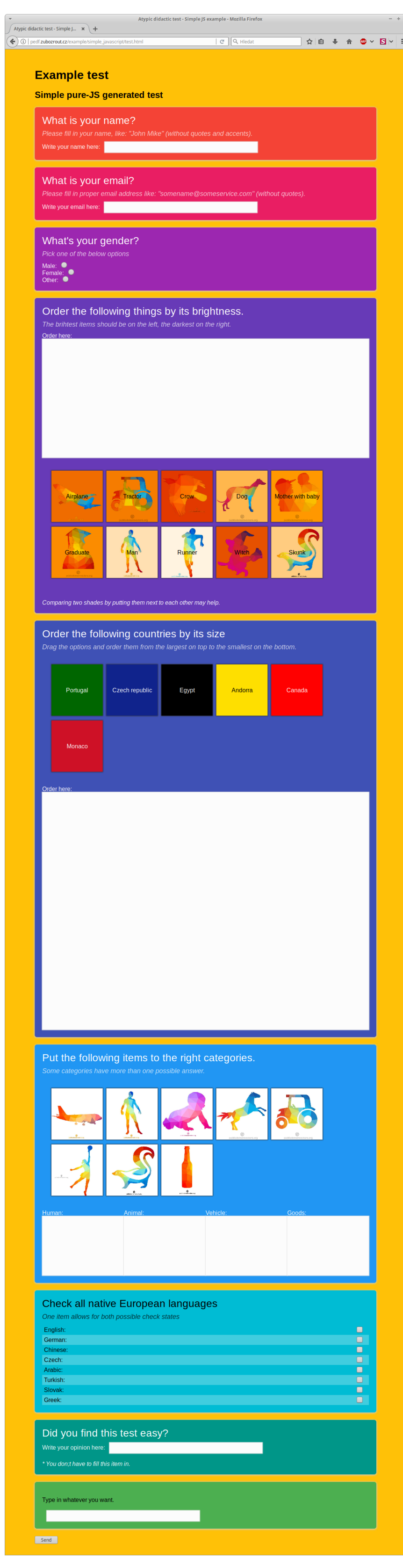
VIRK, Harminder. Singleton Classes In Es6. In: *Caffiene Blogging: UI , design, dev and beauty around the web* [online]. Indie: Harminder Virk, 2015 [cit. 2017-02-17]. Dostupné z: http://amanvirk.me/singleton-classes-in-es6/

VRÁNA, Jakub. *1001 tipů a triků pro PHP*. Brno: Computer Press, 2010. ISBN 978-80-251-2940-1.

10 Přílohy



Obrázek 10.1: Diagram tříd knihovny pro tvorbu formulářů



Obrázek 10.2: Snímek obrazovky kompletního ukázkového testu tak, jak je vykreslen ve webovém prohlížeči Firefox.

Výpis 1: Třída LibraryLoader pro snazší načítání více souborů se skripty do jedné stránky

```
1   "use strict";
2
3   class LibraryLoader {
4           constructor(libraryStructure) {
5                   this.libraryStructure = libraryStructure || {
6                           directory: "source/",
7                           libraries: []
8                   };
9                   return this;
10          }
11
12          load(callback) {
13                  if(this.libraryStructure.directory && this.libraryStructure.libraries) {
14                          let loadPromises = [];
15                          for(let i = 0; i < this.libraryStructure.libraries.length; i++) {
16                                  loadPromises.push(this.loadSingle(this.libraryStructure.libraries[i]));
17                          }
18
19                          return Promise.all(loadPromises).then(() => {
20                                  if(typeof callback !== typeof undefined && typeof callback === "function") {
21                                          callback();
22                                  }
23                                  return "All scripts loaded";
24                          });
25                  }
26                  else {
27                          return Promise.reject(new Error("No scripts to load."));
28                  }
29          }
30
31          loadSingle(libraryName) {
32                  libraryName = libraryName || "";
33                  return new Promise((resolve, reject) => {
34                          let newScript = document.createElement("script");
35                          let scriptUrl = this.libraryStructure.directory + libraryName;
36                          newScript.setAttribute("src", scriptUrl);
37                          document.head.appendChild(newScript);
38
39                          newScript.addEventListener("load", () => {
40                                  resolve(scriptUrl);
41                          });
42
43                          newScript.addEventListener("error", () => {
44                                  reject(new Error("Loading " + scriptUrl + " failed"));
45                          });
46                  });
47          }
48  }
```

Výpis 2: Plugin pro parsování HTML do datové struktury, jež je následně předána vlastní knihovně pro tvorbu testů

```
1   "use strict";
2
3   /**
4    * A DT library plugin to convert a simple HTML structure to a DT enhanced test form.
5    */
6   (function(window, document) {
7           function init() {
8                   let formsholder = new FormsHolder();
9
10                  let domForms = document.getElementsByTagName("form");
11                  let forms = [];
12                  for(let i = 0; i < domForms.length; i++) {
13                          if(domForms[i].dataset.dt !== "yes") {
14                                  break;
15                          }
16
17                          let inputs = domForms[i].getElementsByTagName("input");
18                          let submitBtn = null;
19                          for(let j = 0; j < inputs.length; j++) {
20                                  if(inputs[j].type == "submit") {
21                                          submitBtn = inputs[j];
22                                  }
23                          }
24
25                          forms[i] = formsholder.createForm({
26                                  baseForm: domForms[i],
27                                  allowCorrections: domForms[i].dataset.dtallowcorrections ? domForms[i].dataset.
                                        dtallowcorrections === "true" : false,
28                                  customSubmit: domForms[i].dataset.ctcustomsubmit || false,
29                                  scrollToError: domForms[i].dataset.dtscrolltoerror || false,
30                                  type: domForms[i].dataset.dttype || "",
31                                  submitBtn: submitBtn,
32                                  submitUrl: domForms[i].dataset.submitUrl || null,
33                                  draggable: domForms[i].dataset.dtplugin || null
34                          });
35                          parseForm(forms[i]);
36                  }
37          }
38
39          function parseForm(form) {
40                  let dataWrappers = form.baseForm.getElementsByTagName("dtwrapper");
41                  for(let i = 0; i < dataWrappers.length; i++) {
42                          switch(dataWrappers[i].dataset.dttype) {
43                                  case "text":
44                                          parseText(form, dataWrappers[i], i);
45                                          break;
46                                  case "assign":
47                                          parseAssign(form, dataWrappers[i], i);
48                                          break;
49                                  case "order":
50                                          parseOrder(form, dataWrappers[i], i);
51                                          break;
52                                  case "radio":
53                                          parseRadio(form, dataWrappers[i], i);
54                                          break;
55                                  case "checkbox":
56                                          parseCheckbox(form, dataWrappers[i], i);
57                                          break;
58                                  default:
59                                          console.error("Unknown type");
60                          }
61                  }
62          }
63
64          function getElementStyle(element) {
65                  return String(element.style.cssText);
66          }
67
68          function reorderItems(items) {
69                  if(items && items.length > 1 && items[0].order) {
70                          items = Array.from(items).sort(function(a, b) {
71                                  return parseInt(a.order || 0) - parseInt(b.order || 0);
72                          });
73                  }
74                  return items;
```

```
75          }
76
77          function parseGeneral(domItems, id, customStructure) {
78                  domItems = domItems || [];
79                  let items = [];
80
81                  for(let i = 0; i < domItems.length; i++) {
82                          id = id || domItems[i].id || "";
83
84                          if(customStructure) {
85                                  items.push(customStructure(domItems[i], i, id));
86                          }
87                          else {
88                                  items.push({
89                                          label: domItems[i].dataset.dtlabel || "",
90                                          id: domItems[i].value || CommonFormFunctions.joinStrings(id, i, "-"),
91                                          text: domItems[i].value,
92                                          checkRules: domItems[i].dataset.dtcheckrules || null,
93                                          arrayMatch: domItems[i].dataset.dtarraymatch || null,
94                                          error: domItems[i].dataset.dterror || null,
95                                          required: domItems[i].hasAttribute("required") ? true : false,
96                                          css: getElementStyle(domItems[i]),
97                                          order: domItems[i].dataset.order || ""
98                                  });
99                          }
100                 }
101
102                 while(domItems.length > 0) {
103                         let item = domItems[domItems.length - 1];
104                         item.parentNode.removeChild(item);
105                 }
106                 return reorderItems(items);
107         }
108
109         function getQueryTitle(element) {
110                 /* Check if custom query title is set */
111                 if(element.dataset.title) {
112                         return element.dataset.title;
113                 }
114                 /* Else: Check for previous siblings to get the title name from. Delete the tag if present. */
115                 let previousSibling = element.previousElementSibling;
116                 if(previousSibling) {
117                         /* If tag is a header */
118                         if(/h[1-6]/.test(previousSibling.tagName.toLowerCase())) {
119                                 let title = previousSibling.textContent;
120                                 previousSibling.parentElement.removeChild(previousSibling);
121                                 return title;
122                         }
123                 }
124                 /* Return null if nothing present */
125                 return null;
126         }
127
128         /* Check whether any of the items is required */
129         function anyItemRequired(items) {
130                 for(let i = 0; i < items.length; i++) {
131                         if(items[i].required === true) {
132                                 return true;
133                         }
134                 }
135                 return false;
136         }
137
138         function parseText(form, element, index) {
139                 let items = parseGeneral(element.getElementsByTagName("input"));
140
141                 if(items.length === 1) {
142                         let query = form.newQueryItem({
143                                 id: items[0].id,
144                                 title: getQueryTitle(element),
145                                 subtitle: element.dataset.dtsubtitle || null,
146                                 type: "text",
147                                 label: items[0].label,
148                                 checkRules: items[0].checkRules,
149                                 error: element.dataset.dterror || null,
150                                 required: items[0].required,
151                                 css: getElementStyle(element)
152                         }, true);
```

```
153                         element.appendChild(query.container);
154                 }
155         }
156 
157         function parseAssign(form, element, index) {
158                 let inputs = parseGeneral(element.getElementsByTagName("input"), "assign-item");
159                 let selects = element.getElementsByTagName("select");
160                 if(inputs.length >= 1 && selects.length === 1) {
161                         let select = selects[0];
162                         let options = parseGeneral(select.getElementsByTagName("option"), "draggable-item",
                              function(option) {
163                                 return {
164                                         id: option.value || "",
165                                         label: option.innerHTML,
166                                         text: option.innerHTML,
167                                         css: getElementStyle(option)
168                                 };
169                         });
170                         select.parentNode.removeChild(select);
171                         let query = form.newQueryItem({
172                                 id: "assign",
173                                 title: getQueryTitle(element),
174                                 subtitle: element.dataset.dtsubtitle || null,
175                                 type: "assign",
176                                 items: inputs,
177                                 options: options,
178                                 error: element.dataset.dterror || null,
179                                 optionsOrder: element.dataset.dtoptionsorder || null,
180                                 css: getElementStyle(element)
181                         }, true);
182                         element.appendChild(query.container);
183                 }
184                 else {
185                         console.error("Exactly one select element must be present in one dtwrapper, " + selects.
                              length + " present");
186                 }
187         }
188 
189         function parseOrder(form, element, index) {
190                 let inputs = element.getElementsByTagName("input");
191                 let selects = element.getElementsByTagName("select");
192                 if(inputs.length <= 1 && selects.length === 1) {
193                         let input = inputs.length === 1 ? inputs[0] : null;
194                         let inputDirection = input && input.dataset.dtdirection ? input.dataset.dtdirection : "
                              vertical";
195                         let inputCheckRules = input && input.dataset.dtcheckrules ? input.dataset.dtcheckrules :
                              null;
196                         let inputOptionsOrder = input && input.dataset.dtoptionsorder ? input.dataset.
                              dtoptionsorder : null;
197                         input.parentNode.removeChild(input);
198 
199                         let select = selects[0];
200                         let options = parseGeneral(select.getElementsByTagName("option"), "draggable-item",
                              function(option) {
201                                 return {
202                                         id: option.value,
203                                         label: option.innerHTML,
204                                         text: option.innerHTML,
205                                         css: getElementStyle(option),
206                                         order: option.dataset.order || ""
207                                 };
208                         });
209                         select.parentNode.removeChild(select);
210                         let query = form.newQueryItem({
211                                 id: "draggable-" + inputDirection,
212                                 title: getQueryTitle(element),
213                                 subtitle: element.dataset.dtsubtitle || null,
214                                 type: "order",
215                                 direction: inputDirection,
216                                 checkRules: inputCheckRules,
217                                 error: element.dataset.dterror || null,
218                                 options: options,
219                                 optionsOrder: inputOptionsOrder,
220                                 css: getElementStyle(element)
221                         }, true);
222                         element.appendChild(query.container);
223                 }
224                 else {
```

```
225                             console.error("Exactly one select element must be present in one dtwrapper, " + selects.
                                     length + " present");
226                     }
227             }
228 
229             function parseRadio(form, element, index) {
230                     let items = parseGeneral(element.getElementsByTagName("input"), "radio-item");
231                     let query = form.newQueryItem({
232                             id: "radio-" + index,
233                             title: getQueryTitle(element),
234                             subtitle: element.dataset.dtsubtitle || null,
235                             type: "radio",
236                             required: anyItemRequired(items),
237                             error: element.dataset.dterror || null,
238                             items: items,
239                             css: getElementStyle(element)
240                     }, true);
241                     element.appendChild(query.container);
242             }
243 
244             function parseCheckbox(form, element, index) {
245                     let items = parseGeneral(element.getElementsByTagName("input"), "checkbox-item");
246                     let query = form.newQueryItem({
247                             id: "checkbox-" + index,
248                             title: getQueryTitle(element),
249                             subtitle: element.dataset.dtsubtitle || null,
250                             type: "checkbox",
251                             required: anyItemRequired(items),
252                             error: element.dataset.dterror || null,
253                             items: items,
254                             css: getElementStyle(element)
255                     }, true);
256                     element.appendChild(query.container);
257             }
258 
259             window.addEventListener("load", (event) => {
260                     init();
261             });
262     })(window, document);
```