Polish Japanese Institute of Information Technology

Polsko-Japońska Akademia Technik Komputerowych
Zamiejscowy Wydział Informatyki w Gdańsku

Imię i nazwisko dyplomanta: Marek Marecki

Nr albumu: 14288

Kierunek studiów: Informatyka **Rodzaj studiów:** niestacjonarne

Praca dyplomowa

Temat pracy: Viua VM w akcji
Implementacja języka wysokiego poziomu i prostej aplikacji

Temat w języku angielskim: Viua VM in action
Implementation of a high-level programming language and a simple application

Opiekun pracy: dr hab. Marek. A. Bednarczyk, prof. PJWSTK

Wykonawcy: Marek Marecki i Krzysztof Franek

Streszczenie: Głównym pytaniem, na które odpowiadamy w pracy jest „Czy Viua VM umożliwia tworzenie niezawodnego, wysoce współbieżnego, nietrywialnego oprogramowania?”
Odpowiadamy na nie projektując język programowania wysokiego poziomu (ViuAct) i implementując jego kompilator, oraz pisząc czat (ViuaChat) w tymże języku.

Gdańsk, 2019 rok

Spis treści

Spis rysunków

Spis listingów

Praca została złożona za pomocą systemu LaTeX.

Część I

Cel, strategia wykonania, i przebieg prac

Rozdział 1

Wprowadzenie

We wprowadzeniu chcielibyśmy poruszyć ogólne tematy związane z naszą pracą, oraz przedstawić Czytelnikowi spójny wstęp zapewniający solidne podstawy do dalszej lektury. Zakładamy, że Czytelnik ma doświadczenie z językami programowania i umie biegle posługiwać się co najmniej jednym „*mainstreamowym*" językiem, np. C++, Java, czy Python.

Wprowadzenie jest dla nas miejscem, w którym przedstawimy problemy dręczące popularne obecnie języki programowania oraz powiemy skąd się te problemy biorą i dlaczego w miarę postępu technologii są one coraz trudniejsze do zignorowania.

Po krótce przedstawiona zostanie również maszyna wirtualna Viua, która jest platformą, na której nasza praca się opiera.

1.1 Przedstawienie problemu

Tytułem pracy jest „Viua VM w akcji". Problemem, który poruszamy jest sprawdzenie czy Viua VM w stanie „zastanym" (tj. w takim w jakim znajdowała się jej implementacja na początku prac nad projektem inżynierskim) umożliwia pisanie niezawodnego, wysoce współbieżnego, nietrywialnego oprogramowania. Jest to zawarte w głównym pytaniu, na które staramy się w naszej pracy odpowiedzieć:

> **Czy Viua VM umożliwia tworzenie niezawodnego, wysoce współbieżnego, nietrywialnego oprogramowania?**

W naszej pracy prezentujemy język programowania, który z założenia ma pozwalać na tworzenie oprogramowania niezawodnego i wykorzystującego potencjał współbieżności w stopniu wyższym niż powszechnie używane, „mainstreamowe" języki programowania.

Aby udowodnić, że w wytworzonym języku możliwe jest tworzenie nietrywialnego oprogramowania prezenetujemy aplikację użytkową – czat – napisany w tym języku. Czat umożliwi komunikację (zorganizowaną w „pokoje") wielu użytkownikom naraz. Wybór rodzaju aplikacji (czat) jest warunkowany tym, że oprogramowanie komunikacyjne powinno charakteryzować się tymi cechami, które chcemy uzyskać:

1. niezawodnością – jeśli jedno połączenie ulegnie awarii to pozostałe powinny działać dalej
2. izolacją procesów – każde połączenie powinno być izolowane od wszystkich innych
3. współbieżnością – wiele połączeń musi być obsługiwanych w tym samym czasie

1.1.1 Ogląd sytuacji

Współczesny *hardware* zmierza coraz bardziej w stronę współbieżności oraz przetwarzania równoległego. Firma AMD w roku 2018 wprowadziła na rynek konsumencki procesory wielordzeniowe z serii Threadripper ([1]) prezentujące nawet 32 rdzenie logiczne.

Współczesny *software* stoi w miejscu. Poza oprogramowaniem specjalistycznym (np. Blender[1] mało

[1] `https://www.blender.org/`

który program jest w stanie wykorzystać więcej niż kilka wątków. Współbieżności szuka się nieco „na siłę"; np. przeglądarki internetowe uruchamiają współbieżnie obsługę wielu kart.

Mainstreamowe języki w większości korzystają z wątków (np. Java, C++) lub są stricte jednowątkowe i niedostosowane do przetwarzania współbieżnego i równoległego (np. Python).

Dla takich języków tworzone są biblioteki ułatwiające wykorzystanie mechanizmów systemu operacyjnego (np. wieloprocesowość dla języka Python) lub cała infrastruktura umożliwiająca rozproszenie pracy, np. sewery pracy (np. Gearman[2], Celery[3]) czy kolejki wiadomości (np. RabbitMQ[4], ZeroMQ[5]). To wszystko są jednak jedynie „łatki" mające za zadanie dodać do istniejących języków mechanizmy, z którymi pierwotnie nie były one projektowane. Istnieje dla takiego działania angielski termin bardzo dobrze oddający jego postać – *retrofitting*[6]:

> **retrofit**
> *verb*
> to provide a machine with a part, or a place with equipment, that it did not originally have when it was built

Polski odpowiednik tego słowa to „doposażyć", patrz [2].

Istnieją środowiska i języki zaprojektowane od zera z myślą o współbieżności i programowaniu równoległym, a nawet rozproszonym. Najbardziej znanym przykładem takiego środowiska jest BEAM, maszyna wirtualna języka Erlang[7]. To środowisko wraz z językiem jest z powodzeniem wykorzystywane w sprzęcie telekomunikacyjnym firmy Ericsson, oraz do tworzenia aplikacji użytkowych, których działanie obejmuje niemal z definicji działanie rozproszone i współbieżne, na przykład w serwerach komunikatora Discord[8].

1.2 Cel

Ta praca inżynierska motywowana jest chęcią stworzenia bazy programistycznej (środowiska uruchomieniowego i języka programowania) umożliwiającej programistom pisanie oprogramowania od początku uwzględniającego przetwarzanie współbieżne na pierwszym miejscu, oraz charakteryzującego się wysokim poziomem niezawodności i stabliności działania.

Osiągniemy to dzięki zbudowaniu języka wyposażonego w łatwo dostępne konstrukcje umożliwiające wprowadzenie współbieżności do programu, oraz uruchamianiu programów wynikowych w środowisku zdolnym do rozłożenia pracy na całość dostępnych zasobów sprzętowych.

1.2.1 Problemy i ryzyko

Postawienie współbieżności na pierwszym miejscu powoduje, że problemy związane z pisaniem poprawnych programów są bardziej liczne niż w innych (niewspółbieżnych) językach – oprócz zapewnienia poprawności pojedynczego procesu, programista musi zadbać o poprawność interakcji między procesami, oraz o stabilność działania oprogramowania w momencie awarii któregoś z procesów składowych programu.

Zminimalizujemy ryzyko płynące z wprowadzenia współbieżności do warsztatu programistów udostępniając im mechanizmy umożliwiające opanowanie awarii, propagowanie informacji o błędach, oraz izolację poszczególnych procesów składowych.

---

[2] `http://gearman.org/`
[3] `http://www.celeryproject.org/`
[4] `https://www.rabbitmq.com/`
[5] `http://zeromq.org/`
[6] `https://dictionary.cambridge.org/dictionary/english/retrofit`
[7] `http://www.erlang.org/`
[8] `https://discordapp.com/`

1.3 Układ pracy inżynierskiej

W rozdziale 4. Język ViuAct i jego kompilator – założenia (na stronie 37) przedstawiamy język ViuAct, a w rozdziale 6. jego implementację - kompilator. Są to narzędzia wykorzystywane do stworzenia aplikacji użytkowej przedstawionej w rozdziale 7. Program ViuaChat – założenia (na stronie 85).

Wkład własny członków zespołu przedstawiony jest w rozdziale 10. na stronie 137.

1.4 Priorytetyzacja zadań

W wielu rozdziałach pracy członkowie zespołu dokonywali priorytetyzacji poszczególnych zadań lub wymagań. W tym celu konsekwentnie wykorzystywano tzw. skalę „MoSCoW". Poszczególne wielkie litery w nazwie skali oznaczają:

- **„M"** (z ang. *must*) - zadania (zasady, wymagania), których spełnienie jest niezbędne dla realizacji systemu
- **„S"** (z ang. *should*) - są to zadania (zasady, wymagania) o wysokim priorytecie, które powinny zostać spełnione, o ile tylko jest to możliwe;
- **„C"** (z ang. *could*) - dobrze byłoby zrealizować takie zadania (zasady, wymagania), ale zależy to od czasu i zasobów, jakie pozostaną do dyspozycji po ukończeniu zadań oznaczonych jako „M"i „C";
- **„W"** (z ang. *won't*) - takie zadania (zasady, wymagania,) po dyskusji, zostały wycofane z dalszej realizacji.

1.5 Słownik pojęć

W tym rozdziale prezentujemy słownik pojęć używanych w pracy, a których znaczenie może być niejednoznaczne lub nieznane czytelnikowi. Pojęcia są ułożone w kolejności alfabetycznej.

1.5.1 Pojęcia ogólne

BEAM	maszyna wirtualna, na której działa Erlang
CLI	ang. *command line interface*; interfejs linii poleceń znany ze standardowych narzędzi unixowych jak np. `ls`, `grep` czy `sed`
Erlang	język programowania zaprojektowany w 1986 roku na potrzeby firmy Ericsson przez zespół, w którego skład wchodzili: Joe Armstrong, Robert Virding, Mike Williams
FFI	(ang. *foreign function interface*) interfejs umożliwiający wywoływanie z jednego języka funkcji napisanych w innym języku
interakcje języka z platformą	wykorzystanie zasobów sprzętowych, operacje I/O, oraz wszelkie efekty uboczne będące wynikiem działania programu
kernelspace	(ang. przestrzeń jądra) abstrakcyjna przestrzeń, w której działa kod jądra systemu operacyjnego; w przypadku Viua VM oznacza przestrzeń, w której działa sama Viua VM
Lisp	język programowania zaprezentowany w 1958 roku przez John'a McCarthy'ego
userspace	(ang. przestrzeń użytkownika) abstrakcyjna przestrzeń, w której działają programy „użytkowe"; w przypadku Viua VM oznacza przestrzeń, w której działają procesy (aktory)

Viua VM	maszyna wirtualna, umożliwiająca uruchamianie programów wykorzystujących współbieżność
ViuAct	język wysokiego poziomu, oparty o modelu aktorów, kompilowany do języka asemblera Viua VM

1.5.2 Pojęcia związane z językiem i kompilatorem

aktory	procesy działające w programie, jednostka podziału programów na współbieżnie działające fragmenty (zobacz „Czemu *aktory*, a nie *aktorzy*?" na stronie 7)
biblioteka	zbiór modułów
compile-time	„czas kompilacji"; czas, w którym program jest kompilowany
jądro	podsystem Viua VM odpowiadający za uruchamianie programów (tzw. „kernel")
jednostka translacji	w przypadku języka ViuAct jest to pojedynczy moduł
kompilator	program tłumaczący kod w jednym języku (zazwyczaj wysokiego poziomu) na kod o takim samym znaczeniu w innym języku (zazwyczaj niższego poziomu)
leksem	ciąg znaków odpowiadający wzorcowi określającemu możliwe wartości tokenu
linker	program łączący wiele modułów w plik wykonywalny
moc funkcji	(ang. *arity*) ilość parametrów formalnych przyjmowanych przez funkcję (pojęcie zapożyczone z pojęcia mocy zbioru)
model aktorów	model przetwarzania współbieżnego, opierający się na podstawowych strukturach, nazywanych „aktorami", posiadających swój własny prywatny stan i porozumiewających się pomiędzy sobą za pomocą komunikatów
moduł	w zależności od kontekstu: *1/* kod źródłowy modułu w języku ViuAct, lub *2/* plik zawierający bytecode w formacie, który może zostać wykorzystany przez linker bądź jądro Viua VM do dołączenia, lub *3/* zbiór funkcji, wyliczeń, i modułów w języku ViuAct
PID	identyfikator procesu, unikalny w trakcie życia procesu w obrębie jednej maszyny wirtualnej (tj. żadne dwa procesy działające w jednym momencie na tej samej maszynie wirtualnej nie będą miały takiego samego identyfikatora)
plik wykonywalny	plik zawierający bytecode w formacie, który może zostać wykonany przez jądro Viua VM
runtime	„środowisko uruchomieniowe"; maszyna wirtualna bądź realna, na której wykonywany jest program
run-time	„czas wykonywania"; czas, w którym program jest wykonywany przez VM; przeciwieństwo *compile-time*
token	abstrakcyjna reprezentacja konkretnego elementu leksykalnego, np. słowa kluczowego lub identyfikatora, składająca się z nazwy typu tokenu i leksemu, który dana instancja tokenu zawiera
wiadomość	wiadomość jest to dowolna wartość (np. liczba całkowita, napis, struktura) wysłana przez jednego aktora do drugiego

wywołanie ogonowe (ang. *tail call*) lub: wywołanie w pozycji ogonowej, pozycja ogonowa; jest to wywołanie przez funkcję samej siebie lub innej funkcji w pozycji, w której to wywołanie jest ostatnią operacją na danej ścieżce wykonania – takie wywołanie pozwala nie dokładać kolejnej ramki wywołania na stos, a podmienić istniejącą, co pozwala korzystać z algorytmów rekurencyjnych bez obawy o przepełnienie stosu (jest to bardzo istotne w językach funkcyjnych, w których rekurencja jest głównym sposobem na wykonywanie obliczeń wymagających iteracji)

wzorzec wyrażenie regularne określające jaką formę mogą przyjąć leksemy danego typu tokenu

Czemu *aktory*, a nie *aktorzy*? *Aktorzy* to osoby odgrywające role w teatrze lub filmie. *Aktory* to współbieżnie działające, izolowane od siebie procesy. Ten podział nie jest sztywno ustalony w środowisku akademickim i poglądy na to, która forma jest poprawna są różne. W naszej pracy będziemy korzystać z formy *aktory* zawsze kiedy będzie mowa o aktorach w znaczeniu „procesów".

Odmiana przez przypadki względem formy „ludzkiej" różni się dla liczby mnogiej dla **mianownika**, **biernika** i **wołacza** gdzie używamy formy „aktory" zamiast „aktorzy". Innym tego typu słowem w języku polskim jest „pilot".

1.5.3 Pojęcia związane z chatem

Backend Warstwa aplikacji internetowej, obejmująca oprogramowanie wykonywane po stronie serwera

Frontend Warstwa aplikacji internetowej, obejmująca oprogramowanie wykonywane po stronie klienta (użytkownika)

Pokój Współdzielony czat, do którego dostęp ma równocześnie wielu uczestników, widzących nawzajem wysyłane przez siebie wiadomości

Prywatna konwersacja, Pokój PW Specjalny rodzaj pokoju; jego uczestnikami jest zawsze dwóch różnych użytkowników, a nazwa pokoju jest utworzona z alfabetycznie ułożonych nazw tych użytkowników, połączonych znakiem krzyżyka (#)

Wiadomości prywatne Wiadomości, które są wysyłane konkretnemu użytkownikowi i są widoczne wyłącznie dla nadawcy i odbiorcy takiej wiadomości dzięki mechanizmowi prywatnych konwersacji

Wpięcie użytkownika w pokój Rodzaj relacji, polegający na tym, że dany użytkownik ma możliwość nadawania i odbierania wiadomości w ramach określonego pokoju

Rozdział 2

Strategia prowadzenia prac

W tym rozdziale opisujemy metodologię prac nad poszczególnymi częściami projektu. Każda z nich była rozwijana innym sposobem z powodu dużych rozbieżności specyfiki konkretnych problemów – zupełnie inaczej pracuje się nad semi-formalną specyfikacją języka programowania niż nad aplikacją użytkową, więc podjęliśmy decyzję o rozdzieleniu tych części i poprowadzeniu ich jako osobne, mniejsze projekty wewnątrz pracy inżynierskiej.

2.1 Specyfikacja języka ViuAct

Tworzenie specyfikacji języka odbywało się w modelu kaskadowo-prototypowym i przeplatało się z implementacją prototypu kompilatora.

2.1.1 Określenie wymagań

Przed rozpoczęciem specyfikowania poszczególnych konstrukcji językowych należało podjąć decyzję co w języku powinno się znaleźć, a co nie powinno zostać włączone do specyfikacji. Pragmatyczne podejście pozwoliło ograniczyć do minimum liczbę potrzebnych elementów języka – zrezygnowano między innymi z umieszczenia pętli w języku ponieważ można je zastąpić rekurencyjnymi wywołaniami w pozycji ogonowej (*recursive tail calls*).

2.1.2 Specyfikacja konkretnej konstrukcji języka

Każda konstrukcja językowa była najpierw projektowana, następnie dokumentowana i omawiana (żeby wychwycić oczywiste błędy „grube" na wczesnym etapie prac). Potem proces specyfikacji zostawał zawieszony do momentu wytworzenia w kompilatorze prototypu implementacji danej konstrukcji, lub odrzucenia jej jako niewykonalnej w zakładanym czasie lub zakresie funkcjonalności. Na tym etapie specyfikacja była „luźna" i półformalna.

Wynik prac prototypowych (sukces bądź porażka) decydował o tym czy prace specyfikacyjne były prowadzone dalej. Jeśli konstrukcja okazywała się możliwa do zaimplementowania jej specyfikacja była formalizowana oraz przygotowywana była jej dokumentacja (obejmująca m.in. przykłady użycia). Po tym etapie specyfikacja była już w pełni formalna.

2.2 Kompilator języka ViuAct

Prace nad kompilatorem języka ViuAct były prowadzone w identyczny sposób jak prace nad Viua VM. Jest to połączenie strategii prototypowania i iteracji, które spawdza się przy projekcie rozwijanym w stylu *one-man show*, czyli tam gdzie za cały projekt odpowiada jeden programista. Jest to częsta sytuacja w przypadku projektów Free Software i na takim modelu wytwarzania był oparty sposób wytworzenia kompilatora języka ViuAct.

2.2.1 Określenie wymagań

Przed rozpoczęciem prac nad kompilatorem języka ViuAct należało określić wymagania jakie powinien spełnić. Z uwagi na krótki czas trwania projektu lista wymagań musiała być krótka. Zaowocowało to rezygnacją z funkcjonalności podnoszących *quality of life* programisty korzystającego z kompilatora, np. dokładnego raportowania błędów. Wymagania, które finalnie zostały zaakceptowane są opisane w rozdziale 4.2 na stronie 38.

2.2.2 „Swobodna" strategia

Strategia nie zakładała planowania znanych z Agile sprintów, przypisywania zadań do konkretnych terminów, czy tworzenia złożonych *workflow*. Jedynym planem była lista zadań, które musiały być zrealizowane aby projekt został z sukcesem ukończony i powiązanie ich w relacje *zależy-od*, gdzie jeśli zadanie A *zależy-od* zadania B, to zadanie B musi być ukończone przed zadaniem A. W wyniku takich powiązań powstało drzewo, które określało kolejność w jakiej poszczególne zadania musiały być wykonywane.

Takie względnie „swobodne" podejście do spraw planowania i formalnego zarządzania projektem pozwoliło na drastyczne zminimalizowanie biurokratycznego narzutu na projekt i umożliwiło przeznaczenie większej ilości czasu na prace programistyczne bądź przygotowywanie dokumentacji.

2.2.3 Narzędzia wspierające wybraną strategię

Zadania w tej części projektu były śledzone za pomocą narzędzia `issue`[1], które jest również wykorzystywane przez jednego z członków zespołu (Marka Mareckiego) do śledzenia zadań w pracy zawodowej oraz jego projektach Free Software i bezpośrednio wspiera opisane powyżej „swobodne" podejście do zarządzania zadaniami w projekcie.

Prace implementacyjne przeplatały się z pracami nad specyfikacją w celu weryfikowania na bieżąco co jest możliwe do wykonania w zamierzonym czasie w całości, co tylko w części, a co nie jest możliwe do wykonania przy zakładanych zasobach czasowo-ludzkich.

2.2.4 Testowanie

Prace nad kompilatorem języka ViuAct były „pilnowane" przez zestaw testów, w którym każdy test był prostym, ale kompletnym programem testowym.

Każdy test sprawdza czy dany program poprawnie się kompiluje i czy po wykonaniu daje oczekiwane wyniki. Ta strategia – brak testów poszczególnych elementów kompilatora, a jedynie weryfikacja, że całość działa poprawnie – została zaadaptowana z projektu Viua VM, gdzie jest z powodzeniem wykorzystywana. Zapewnia ona maksimum zysków przy minimum narzutu i kosztów czasowych:

1. wiemy, że kompilator jako całość działa ponieważ każdy z programów testowych (1) jest możliwy do skompilowania (2) daje oczekiwane wyniki

2. traktujemy kompilator jako „czarną skrzynkę" i nie testujemy wnętrza kompilatora, a jedynie weryfikujemy czy poprawnie przetwarza kod źródłowy na kod wynikowy co pozwala na dowolne manipulowanie implementacją bez zmiany testów

Takie podejście wymaga częstego uruchamiania testów w celu weryfikacji czy zmiany, które wprowadzamy nie powodują porażki wykonania zestawu testowego – łatwiej jest znaleźć błąd w czterech liniach zmian niż w czternastu czy czterdziestu – ponieważ taki zestaw testów nie jest pomocny w szukaniu błędów w implementacji. Świadomie podjęliśmy decyzję o przyznaniu większej wagi temu żeby sprawdzić czy w zakładanych przypadkach kompilator zachowuje się poprawnie niż na temu żeby testy pomagały w wyszukiwaniu błędów ponieważ, z uwagi na bardzo ograniczony czas na jego wytworzenie, ważniejsze było osiągnięcie przez kompilator „kompletności" niż jego odporność na błędy.

[1] Kod źródłowy dostępny jest pod adresem `https://github.com/marekjm/issue`. Narzędzie to jest również krótko opisane w dodatku D na stronie 179.

2.3 Program ViuaChat

2.3.1 mini-Scrum

Dla czatu ViuaChat, Krzysztof Franek, odpowiedzialny za tę część projektu, wybrał strategię mini-Scrum. Jest to wariant techniki Scrum, korzystający z tych podobnych wydarzeń (Sprint, Planowanie Sprintu, Przegląd Sprintu, Retrospektywa Sprintu), tych samych artefaktów (Backlog Produktu, Backlog Sprintu, Przyrost) i niektórych ról (Właściciel Produktu, Zespół Deweloperski).

Podstawową zaletą tej metodyki działania był podział zadań na mniejsze fragmenty, a także możliwość łatwego planowania i nadzorowania tempa prac poprzez obserwowanie wykresu wypalania produktu. Umówiono się, że rolę właściciela produktu (ang. *product owner*) będzie pełnił p. Marecki, zaś w rolę zespołu deweloperskiego wcieli się p. Franek. Z powodu mało licznego zespołu, w metodologi mini-Scrum dokonano pewnych uproszczeń w stosunku do oryginalnego modelu Scruma, zaproponowanego w oficjalnym podręczniku ([3]). Zrezygnowano z codziennych scrumów (ang. *daily scrums*), z uwagi na nieregularny charakter prac (uczestnicy projektu studiują w trybie zaocznym i realizują zadania w czasie wolnym od pracy), a także z odrębnej roli *scrum mastera*.

2.3.1.1 Kanban

Dodatkowo, jako technikę prowadzenia Sprintów mini-Scruma wybrano Kanban, w ramach którego każda z historyjek w Sprincie była umieszczana w postaci „karteczki" na wirtualnej tablicy i przemieszczana pomiędzy jej kolumnami, wraz z ich kolejnymi etapami realizacji. Na tablicy przewidziano następujące kolumny: „Do zrobienia", „W realizacji", „W testowaniu", „Zrealizowano". Ponadto, każda karteczka ma przypisaną etykietę z priorytetem.

Początkowo, dla każdego sprintu utworzona osobną tablicę. Karteczki reprezentujące historyjki, których nie udało się zrealizować w sprincie, miały być kopiowane do tablic kolejnych sprintów. Ostatecznie poprzestano na uproszczonym toku postępowania, współdzielącym jedną tablicę dla wszystkich sprintów – karteczki ze zrealizowanymi i przetestowanymi zadaniami poddawano archiwizacji, zaś niezrealizowane wyróżniano dodatkową, ostrzegawczą etykietą.

Oprogramowanie zaproponowane do utrzymywania mini-Scruma dla ViuaChat to Trello, umożliwiające łatwe zaimplementowanie techniki Kanban, rejestrowanie czasu pracy oraz integrację z najpopularniejszymi dostarczycielami repozytoriów Git.

2.3.2 Testowanie

Proces testowania systemu ViuaChat został rozbity na dwa, odrębne poziomy, realizowane w sposób równoległy.

Pierwszy z poziomów dotyczył testów jednostkowych backendu ViuaChat, tj. części systemu ViuaChat, która działa po stronie serwera. Aby zweryfikować działanie poszczególnych komend oraz prawidłowość zwracanych rezultatów, przygotowywane były specjalnie spreparowane pliki HTML. Ich esencją były osadzone skrypty w języku JavaScript, które samoczynnie próbowały łączyć się z serwerem czatu, a w razie sukcesu wysyłały serię zdefiniowanych komend. Przebieg komunikacji w był prezentowany w przeglądarce internetowej.

Drugi z poziomów obejmował typowe testy integracyjne. Proces testowania niejako „wbudowano" w metodologię mini-Scrum. Wynika to z faktu, iż każda z historyjek oraz każde z zadań posiada jasno określoną definicję ukończenia (ang. *Definition of Done*, *DoD*). Nie było zatem możliwe jego ukończenie bez spełnienia z góry określonych warunków.

Rysunek 2.1: Skrypt służący do zautomatyzowanych testów backendu ViuaChat

Rozdział 3

Przebieg prac

Środowisko pracy

Aby ułatwić prace rozwojowe nad składnikami tej pracy inżynierskiej został napisany skrypt przygotowujący środowisko wytwórcze. Zapewnia on spójną strukturę katalogów pomiędzy maszynami członków zespołu co eliminuje niepotrzebne różnice między środowiskami wytwórczymi. Skrypt pozwala na przygotowanie środowiska za pomocą jednego polecenia wydanego w powłoce:

```
wget -O https://viuavm.org/viuact-bootstrap.sh | bash -s
```

Skrypt ten utworzy w katalogu roboczym katalog `viuact-workspace`, pobierze najnowsze wersje każdego z komponentów projektu (klonując ich repozytoria Git) i skompiluje je w odpowiedniej kolejności.

Aby aktywować środowisko (ustawić zmienne środowiskowe istotne dla kompilatora języka ViuAct oraz jądra i assemblera dostarczanych z Viua VM) należy wykonać następujące polecenie wewnątrz utworzonego przez skrypt katalogu `viuact-workspace`:

```
source ./activate.sh
```

Takie podejście (oskryptowanie i automatyzacja przygotowanie środowiska pracy) znacząco zredukowało problemy wynikające z różnic, które członkowie zespołu mogli – nawet omyłkowo – wprowadzić do swojego lokalnego środowiska.

3.1 Specyfikacja języka ViuAct

Tworzenie formalnej specyfikacji języka ViuAct było, paradoksalnie, procesem mało sformalizowanym. Polegało na określeniu głównych założeń języka (opisanych w rozdziale 5.1 Model programowania na stronie 44), co zostało wykonane na jednym z pierwszych spotkań i nadało kierunek dalszemu rozwojowi języka, a następnie na zaprojektowaniu poszczególnych konstrukcji jezykowych w sposób zgodny z założoną wizją.

Równolegle z tworzeniem specyfikacji poszczególnych konstrukcji w języku powstawały ich prototypowe implementacje mające na celu szybką i jak najwcześniejszą weryfikację tego czy dana konstrukcja jest możliwa do zaimplementowania w założonym budżecie czasowym. Jeśli dana konstrukcja okazywała się niemożliwa do zaimplementowania w satysfakcjonującym czasie, lub możliwa do zaimplementowania jedynie w pewnym niewystarczającym fragmencie to była usuwana ze specyfikacji.

3.2 Implementacja kompilatora

Śledzenie prac nad rozwojem kompilatora było wspierane przez dwa narzędzia: system kontroli wersji Git[1] i narzędzie do śledzenia zadań Issue[2].

[1] `https://git-scm.com/`

[2] `https://github.com/marekjm/issue`

3.2.1 Zakończone zadania

Poniżej przedstawiona jest lista zakończonych zadań zarejestrowanych do wykonania podczas projektu. Na dzień 2019-05-19 wykonane zostało 64.44% zaplanowanych zadań. Tak niska wartość wiąże się z tym, że część zadań zostało otwartych z założeniem, że mogą nie zostać wykonane (na zasadzie „jeśli wystarczy czasu"). Mediana czasów całkowitego czasu (od zgłoszenia do zamknięcia) każdego z ukończonych zadań to nieco ponad sześć dni (6 dni i 11 godzin).

Repozytorium programu Issue jest dostępne wraz z repozytorium Git dołączonym jako załącznik do projektu.

3.2.1.1 Add an assembly driver tool for executables

Identyfikator zadania: `5b8f77f4714be2c16d1f87a02b03692c96b983c1`
Rozpoczęte 2018-12-18 23:20:11, zakończone 2018-12-20 16:00:54.

3.2.1.2 Basic CLI chat

Identyfikator zadania: `b8cea1dc8ac19f63ad579ac28196c82f6a9eb7aa`
Rozpoczęte 2019-01-19 22:15:28, porzucone 2019-03-11 19:04:01.

Zadanie zostało porzucone ponieważ powstał czat oparty o protokół WebSocket.

3.2.1.3 Create a module system

Identyfikator zadania: `509ee8af1c2f503e402b34c18224a84caf4c9252`
Rozpoczęte 2018-11-22 19:26:21, zakończone 2018-11-29 06:54:16.

Język ViuAct posiada działający system modułów opisany w rozdziale 5.3.6 Definicje modułów.

3.2.1.4 Create an EBNF notation to describe Viuact syntax

Identyfikator zadania: `3295295f3d3b08cd0a3ae46c1adebe40bb3ebad4`
Rozpoczęte 2018-12-19 00:04:24, zakończone 2019-03-26 20:00:00.

Zadanie zostało zakończone po umieszczeniu w pracy specyfikacji języka ViuAct, która opisuje składnię w notacji EBNF. Specyfikacja znajduje się w rozdziale 5 na stronie 43.

3.2.1.5 Dump intermediate representations

Identifykator zadania: `902ffbd31e61d8446375620f8511042ca88f31e6`
Rozpoczęte 2019-01-07 18:42:28, zakończone 2019-01-07 19:43:09.

Zadanie polegało na umożliwieniu „podejrzenia" tego na czym tak naprawdę pracuje kompilator. W jego wyniku powstał mechanizm, który wymusza na kompilatorze zapisanie do plików zserializowanej postaci strumienia tokenów, AST, lub obu tych rzeczy.

Do zasygnalizowania kompilatorowi tej potrzeby wykorzystywana jest zmienna środowiskowa. Przykłady użycia:

1. `VIUAC_DUMP_INTERMEDIATE=tokens` – zapis strumienia tokenów
2. `VIUAC_DUMP_INTERMEDIATE=exprs` – zapis AST
3. `VIUAC_DUMP_INTERMEDIATE=tokens,exprs` – zapis strumienia tokenów i AST

3.2.1.6 Emit dependency files for modules and executables

Identifykator zadania: `f841ba29230081e4e6e8eba87857532ae98aff0a`
Rozpoczęte 2018-12-18 23:28:01, zakończone 2018-12-20 16:00:13.

Aby poprawnie połączyć moduły `viuact-opt` potrzebuje informacji o tym, jakie moduły są importowane przez dany moduł lub plik wykonywalny. Te zależności są zapisywane w plikach `.d` opisanych dokładnie w rozdziale 6.5.3.2 na stronie 74.

3.2.1.7 Emit interface files for modules

Identifykator zadania: `86fd1a408cf62ecbebb7459b7d08298ce0a851bc`
Rozpoczęte 2018-12-18 23:13:55, zakończone 2019-03-27 18:09:37.

Kompilator podczas przetwarzania wyrażeń importowania modułów (rozdział 5.3.6.3 na stronie 51) potrzebuje informacji o tym jakie funkcje i wyliczenia dany moduł udostępnia. Te informacje zapisane są w plikach z interfejsami opisanych w rozdziale 6.5.3.1 na stronie 72.

3.2.1.8 Expose pointers

Identifykator zadania: `3208ed0629487b2c02e14cfe5173a333bc364121`
Rozpoczęte 2019-02-02 14:18:22, zakończone 2019-02-02 19:55:44.

Kompilator języka ViuAct potrafi śledzić to czy dana wartość jest wskaźnikiem i automatycznie wstawić dereferencję w odpowiednim momencie jedynie dla ograniczonego zakresu przypadków. W związku z tym potrzebne było udostępnienie programistom narzędzi do bezpośredniej obsługi wskaźników. Tym narzędziami są operator dereferencji (`^`) i funkcja `Std.Pointer.take()`.

Nie było by to potrzebne gdyby kompilator języka ViuAct wymagał weryfikacji typów wartości na etapie kompilacji – wtedy informacja o tym czy dana wartość jest wskaźnikiem czy nie musiałaby być dostępna, a więc nie było by potrzeby nadzorować tego ręcznie. Wskaźniki to jedno z miejsc gdzie brak czasu negatywnie wpłynął na specyfikację i implementację języka.

3.2.1.9 Expose the mechanism to set a process watchdog

Identifykator zadania: `cc03c40b74d5ce2a7e9fa398ae0912c9a107b02d`
Rozpoczęte 2019-03-25 16:23:52, zakończone 2019-04-01 21:40:18.

Watchdog to jeden z najważniejszych mechanizmów jakie język ViuAct „odziedziczył” po Viua VM. Jest to funkcja, która jest automatycznie wywyoływana w momencie awarii procesu, aby (1) zarejestrować wystąpienie awarii (2) postawać się naprawić awarię restartując proces (3) poinformować proces nadrzędny o awarii, lub w jakikolwiek inny sposób zareagować na awarię, której główna funkcja procesu nie dała rady obsłużyć.

Dokładny opis funkcji *watchdog* znajduje się w rozdziale 5.3.12 na stronie 54.

3.2.1.10 Implement an exception catching mechanism

Identifykator zadania: `d4bddf8f6b83c9957556465c8e9e112bddf747a1`
Rozpoczęte 2019-01-14 06:42:52, zakończone 2019-03-26 20:00:07.

Wyjątki to podstawowy sposób obsługi błędów w języku ViuAct, a ich obsłużenie jest pierwszą linią obrony przed awariami procesów.

3.2.1.11 Implement an impressive example program

Identifykator zadania: `b28ab36f969b4852a735850af1d1f509647b655c`
Rozpoczęte 2019-01-13 12:24:14, zakończone 2019-03-11 19:08:00.

W ramach tego zadania powstał prototypowy czat wykorzystujący protokół WebSocket, który prezentowaliśmy m.in. na seminarium.

3.2.1.12 Implement boolean values

Identifykator zadania: `0050c352061a18221e21438e697c11d61c11a5a7`
Rozpoczęte 2019-01-07 18:23:12, zakończone 2019-01-07 18:23:37.

Wartości typu boolowskiego są istotnym elementem programów i mogą określać np. czy dana funkcjonalność jest włączona lub wyłączona, czy autoryzacja się powiodła lub nie, itd.

3.2.1.13 Implement compound expressions

Identifykator zadania: `d452c8ee6fcc4b245e7021fec2ceb0958aad1f84`
Rozpoczęte 2018-12-18 23:12:42, zakończone 2019-01-12 12:20:08.

Wyrażenia złożone (opisane w rozdziale 5.3.1.1 na stronie 46) są jednym z ważniejszych elementów języka. Pozwalają na pisanie funkcji wykonujących kilka operacji dzięki temu, że umożliwiają zgrupowanie kilku prostszych wyrażeń w jedno.

3.2.1.14 Implement proper imports inside Viua VM

Identifykator zadania: `6f4f7baff22706628e7003303ff2aa0973460432`
Rozpoczęte 2018-12-22 16:08:28, zakończone 2019-01-14 06:41:41.

Mechanizm importowania modułów w Viua VM był zbyt prymitywny żeby poradzić sobie w elegancki sposób z importowaniem modułów jakie potrzebne było w języku ViuAct. Jedną z najbardziej oczywistych rzeczy, które musiały być poprawione w samej maszynie wirtualnej było importowanie modułów obcych, które nie mogły być linkowane statycznie do plików wykonywalnych lub modułów własnych Viua VM i musiały być linkowane dynamicznie.

Linkowanie dynamiczne musiało być wykonywane jawnie – wykonywany program musiał sam wykonać instrukcje odpowiadające za linkowanie modułu obcego. Po zakończeniu tego zadania (i odpowiadającego mu zadania `0f8c916c2499aab93cf8c55cc1fd6bbf354bd34d` w repozytorium Viua VM) moduły obce mogły być importowane automatycznie przez jądro Viua VM dzięki dodaniu do plików z bytecode sekcji informującej o modułach, które muszą być dolinkowane dynamicznie. W pliku z kodem źródłowym w języku assemblera Viua VM można zarządać takiego dołączenia za pomocą następującego polecenia:

```
.import: [[dynamic]] some::module
```

3.2.1.15 Implement structs

Identifykator zadania: `1f45e908dcde2f12b761da53183e2475619f64ef`
Rozpoczęte 2019-01-02 21:13:48, zakończone 2019-01-06 12:03:04.

Struktury w języku ViuAct są sposobem na tworzenie nowych typów przez programistę. Bez nich, tworzenie nietrywialnego oprogramowania byłoby mocno utrudnione. Struktury są opisane w rozdziale 5.2.2.2 na stronie 45.

3.2.1.16 Implement tail calls

Identifykator zadania: `b680202a36a7358006d87cf371686fc2d42aecdc`
Rozpoczęte 2019-01-06 22:02:12, zakończone 2019-01-06 22:04:33.

Wywołania w pozycji ogonowej są przydatne w wielu miejscach: przy obsłudze błędów (gdzie pozwalają zastąpić funkcję *watchdog* nową funkcją, restartując proces bez zmiany jego PID), zastępują pętle, oraz pozwalają implementować wzorzec „inicjalizacja-implementacja" (rozdział 5.3.8.1 na stronie 51).

3.2.1.17 Implement vectors

Identifykator zadania: `c499f685c5736b3a282ed81251f2d1bd869e3684`
Rozpoczęte 2019-01-10 06:46:42, zakończone 2019-01-12 12:18:49.

Wektory są podstawowym typem danych używanym do przechowywania kilku wartości tego samego typu (mają to samo zastosowanie co tablice i listy).

3.2.1.18 Integrate FFI imports

Identifykator zadania: `713ebf5bc4359b8c74222228e55fcad8d5a7dd05`
Rozpoczęte 2019-01-12 15:25:56, zakończone 2019-01-13 12:22:32.

To zadanie jest powiązane z zadaniem `86fd1a408cf62ecbebb7459b7d08298ce0a851bc` (rozdział 3.2.1.7 na stronie 15). Importowanie modułów obcych wymagało zdefiniowania jak napisać plik interfejsu ręcznie, oraz dodania do kompilatora logiki odpowiedzialnej za importowanie modułów obcych.

3.2.1.19 Make function bodies a single expression

Identifykator zadania: `0d911ab770e326a692cb06170eeaf32acc54b50a`
Rozpoczęte 2018-11-29 06:55:06, zakończone 2019-01-12 12:13:41.

Ciała funkcji w pierwotnej wersji języka ViuAct nie były pojedynczymi wyrażeniami tylko sekwencjami wyrażeń, co wymagało specjalnego kodu do obsługi. Było to niepotrzebne i duplikowało „funkcjonalności" języka, które były zawarte w wyrażeniach złożonych – znaczenie programu nie zmieniałoby się niezależnie od tego czy ciało funkcji byłoby sekwencją wyrażeń czy pojedynczym wyrażeniem złożonym (które samo jest sekwencją wyrażeń).

3.2.1.20 Make functions first class values

Identifykator zadania: `04dd87b88783077ea7ba64c57e9aa2aa1e8f5f76`
Rozpoczęte 2019-03-26 19:10:55, zakończone 2019-03-26 19:58:43.

Zdefiniowanie funkcji jako możliwych do przekazania jako argumenty, możliwych do zwrócenia jako wynik działania innych funkcji, itd. jest niezwykle przydatne. Funkcje jako wartości pierwszoklasowe[3] w języku ViuAct pozwalają na implementację m.in. funkcji w rodzaju `for_each()` czy `map()`.

3.2.1.21 Make the compiler installable

Identifykator zadania: `6b5d4b13a50dd9397c4db7cf326fcd2e6dbfabef`
Rozpoczęte 2019-03-28 22:00:26, zakończone 2019-04-01 21:40:55.

To zadanie było bardzo istotne ponieważ jeśli kompilatora języka ViuAct dałoby się używać jedynie z katalogu, w którym znajduje się jego kod źródłowy to jego użyteczność byłaby mocno ograniczona.

[3]Wartości pierwszoklasowe (ang. *first-class values*) są wartościami, które można (1) dowiązać do zmiennej (2) przekazać jako argument do funkcji (3) zwrócić jako wynik działania funkcji. W języku ViuAct istnieje mało rzeczy, których nie można wykorzystać w ten sposób – moduły, literały timeoutów i definicje wyliczeń. Są to wartości drugoklasowe (ang. *second-class values*).

3.2.1.22 Remove the parentheses around function args

Identifykator zadania: `bb64aeee220ffe1f0cebd7aaf01f19fd972845f5`
Rozpoczęte 2018-11-29 07:05:02, zakończone 2018-11-29 07:13:12.

W wyniku tego zadania składnia wywołań funkcji zmieniona z

```
(fn (arg foo bar))
```

na

```
(fn arg foo bar)
```

Te dodatkowe nawiasy dookoła argumentów funkcji były nadmiarowe i można je było usunąć ze specyfikacji bez strat dla języka.

3.2.1.23 Rework README for the Issue project

Identifykator zadania: `734808030c94028eef293326119d79ffff0d855d`
Rozpoczęte 2018-12-22 16:14:59, zakończone 2019-03-11 19:04:44.

To zadanie musiało być wykonane aby wszyscy członkowie zespołu mogli nauczyć się korzystać z narzędzia do śledzenia zadań wykorzystywanego przy pracach nad kompilatorem języka ViuAct.

3.2.1.24 Specify the module system

Identifykator zadania: `242ef1a8f720ff43d6828708277a917a7b977e59`
Rozpoczęte 2018-12-18 23:37:37, zakończone 2019-03-26 19:59:47.

To zadanie jest tutaj wyszczególnione jako jedyne z zadań polegających na wyspecyfikowaniu pewnych działań czy konstrukcji ponieważ jako jedyne wymagało zdefiniowania jakie produkty powinien generować kompilator języka ViuAct żeby móc się skomunikować z samym sobą – tylko system modułów wymaga żeby kompilator (`viuact-cc`) przetwarzał swoje własne pliki wynikowe.

Obejmowało to między innymi takie problemy jak zapis interfejsu modułów, zapis tego od jakich innych modułów zależy dany moduł, czy to jak moduły zdefiniowane *inline* powinny być zapisane w katalogu z plikami wynikowymi. Jest to zadanie unikalne pod względem tego jak dużą interakcję z systemem plików prowadzi kompilator języka ViuAct podczas wykonywania operacji związanych z nim.

3.2.1.25 Suppress unused register errors thrown by Viua VM assembler

Identifykator zadania: `cd43ec37335b40816059fca8f984088ebfac6339`
Rozpoczęte 2018-12-19 11:25:00, zakończone 2018-12-19 11:26:13.

To zadanie polegało na wyłączeniu części analizy statycznej jaką przeprowadza assembler Viua VM poprzed wywoływanie go z flagą `-Wunused-value`. Kompilator języka ViuAct generuje kod, który jest poprawny, ale czasami zbyt skomplikowany dla silnika analizy statycznej, który działa wewnątrz assemblera Viua VM i bardzo często generował fałszywe alarmy.

Z uwagi na presję czasu fragment analizy statycznej odpowiadający za odrzucenie programu, który zawiera nieużywane wartości nie został poprawiony, a jedynie wyłączony.

3.2.1.26 Test the compiler and assembly driver

Identifykator zadania: `561ce14aef1f418849c436eb3bde6bb5c35c03ad`
Rozpoczęte 2018-12-19 00:05:20, zakończone 2019-03-26 20:00:23.

Zadanie zostało zakończone w momencie, w którym większa część języka została już udokumentowana w specyfikacji i zaimplementowana w kompilatorze. Polegało na przygotowaniu zestawu testów

pokrywającego każdą możliwą konstrukcję językową – czyli stworzenie testu dla konstrukcji warunkowej, wyrażenia złożonego, definicji modułu, itd. Zadanie **nie** obejmowało stworzenia zestawu testów obejmującego kombinacje konstrukcji językowych ponieważ nie jest możliwe stworzenie takiego zestawu z uwagi na to, że ilość takich kombinacji jest nieskończona.

3.2.1.27 Write websockets FFI module

Identifykator zadania: `631bce2891cc7368c68e010a50e04bd55ffbaf03`
Rozpoczęte 2018-12-19 00:00:24, zakończone 2019-03-11 19:06:19.

Istniejące biblioteki implementujące protokoł WebSocket, które były brane pod uwagę do wykorzystania to `https://libwebsockets.org` i `https://github.com/zaphoyd/websocketpp`. Obie jednak zostały odrzucone ze względu na duży narzut koncepcyjny i poziom skomplikowania API (konieczność tworzenia obiektów zarządzających, klas, itd.) oraz konieczność wykorzystania *event loops* dostarczanych przez te biblioteki.

Wykorzystanie *event loop* dyskwalifikuje jakąkolwiek bibliotekę, która używa takiego modelu programowania ponieważ wymusza on przekazanie kontroli nad wykonaniem programu z Viua VM do jakiegoś zewnętrznego programu, co całkowicie niweluje jakiekolwiek gwarancje zapewniane przez Viua VM.

Oprócz tego czysto technicznego powodu odrzucenia istniejących bibliotek jest jeszcze powód subiektywny: stopień ich skomplikowania jest odpychający. Zamiast wystawić socket i udostępnić proste API pozwalające na odczytywanie i zapiswanie wiadomości oraz nawiązywanie i zrywanie połączenia, a kontrolę zostawić w rękach programu, który będzie ich używał – wymagają oddania kontroli w ich „ręce" i wpinania się w ich *event loop*.

Zamiast pozwolić wziąć się jako zakładnika, zadecydowaliśmy o implementacji biblioteki zapewniającej obsługę protokołu WebSocket od zera. Ta biblioteka jest opisana w dodatku C na stronie 175.

3.3 Implementacja ViuaChat

Poniżej przedstawiono zadania, jakie realizowano w ramach kolejnych sprintów. Część z nich odnosi się do pierwotnych wymagań (w tym *user stories*, zaś niektóre zostały sformułowane w trakcie samej realizacji, pod wpływem pierwotnych potrzeb.

3.3.1 Sprint 1

Termin realizacji: 2-9 marca 2019 r.

3.3.1.1 Cel sprintu

Celem sprintu było utworzenie podstawowej funkcjonalności związanej z komunikacją frontend-backend – ustanawiania połączenia WebSocket, dekodowania komunikatów bo obu stronach aplikacji, a także podstawowa walidacja funkcjonalności.

3.3.1.2 Zadania oparte o pierwotne wymagania

Identyfikator	WS-01
Treść	W rozwiązaniu należy wykorzystać środowisko Viua VM i język ViuAct
Nakład godzinowy (planowany / włożony)	Nakład godzinowy związany z zadaniem ujęto przy realizacji zadań ZZ-01 i ZZ-02.
Ukończono?	Tak.

Identyfikator	WS-02
Treść	Czat będzie użytkowany jako aplikacja webowa typu *single page application*
Nakład godzinowy (planowany / włożony)	Nakład godzinowy związany z zadaniem ujęto przy realizacji zadań ZZ-01 i ZZ-02.
Ukończono?	Tak.

3.3.1.3 Zadania wykraczające poza pierwotne wymagania

Identyfikator	ZZ-01
Treść	Frontend aplikacji jest w stanie nawiązać połączenie WebSocket z backendem.
Kryteria akceptacji	1. Frontend nawiązał połączenie z serwerem (tj. doszło do zmiany protokołu z HTTP na WS). 2. Frontend wysłał wiadomość do backendu, a wiadomość została odebrana. 3. Backend wysłał wiadomość do frontendu, a wiadomość została odebrana.
Nakład godzinowy (planowany / włożony)	5h / 5.5h
Ukończono?	Tak.

Identyfikator	ZZ-02
Treść	Obie strony (tj. frontend i backend) potrafią zakodować wiadomości w ustalonym formacie JSON przed ich wysyłką i odebrać je po ich odebraniu.
Kryteria akceptacji	1. Frontend potrafi zakodować wiadomość w formacie JSON – wiadomość synchroniczną oraz wiadomość asynchroniczną. 2. Frontend potrafi odkodować wiadomość w formacie JSON. 3. Frontend potrafi ustalić, czy odebrana wiadomość jest synchroniczna, czy asynchroniczna, a w przypadku synchronicznej – dopasować ją do wysłanej wcześniej wiadomości. 4. Frontend potrafi właściwie obsługiwać wyjątki wynikające z błędów w składni JSON odbieranych wiadomości oraz z błędów w strukturze obiektów JSON: braku obowiązkowych własności oraz nieprawidłowych typów tych wiadomości. Błędy te nie powodują krytycznego przerwania pracy działania programu. 5. Backend potrafi odkodować otrzymywane wiadomości w formacie JSON. 6. Backend potrafi ustalić, czy otrzymana wiadomość jest synchroniczna, czy asynchroniczna. 7. Backend potrafi odesłać odpowiedź pasującą do otrzymanej wiadomości synchronicznej. 8. Backend potrafi właściwie obsługiwać wyjątki z tytułu niewłaściwie skonstruowanych wiadomości. W razie otrzymania takiej wiadomości, nie dochodzi do przerwania pracy serwera (tj. aktora `WsConnector`).
Nakład godzinowy (planowany / włożony)	10h / 12h
Ukończono?	Tak.

3.3.2 Sprint 2

Termin realizacji: 10-23 marca 2019 r.

3.3.2.1 Cel sprintu

W ramach tego sprintu zaplanowano opracowanie mechanizm autoryzacji do czatu i możliwość przejrzenia listy pokojów.

3.3.2.2 Zadania oparte o pierwotne wymagania

Identyfikator	WF-01
Treść	Jako użytkownik serwera czatu, chcę się do niego zalogować, aby zobaczyć listę pokojów dyskusyjnych.
Kryteria akceptacji	1. Po wejściu na czat bez rozpoczętej sesji, pokazuje się monit o podanie nazwy użytkownika. 2. Po wpisaniu nazwy użytkownika i zatwierdzeniu, użytkownik rozpocznie sesję na serwerze czatu. 3. Tuż po rozpoczęciu sesji czatu, użytkownik zobaczy listę pokojów.
Nakład godzinowy (planowany / włożony)	15h / 21h
Ukończono?	Tak.

Identyfikator	HN-01
Treść	Długość nazwy użytkownika jest ograniczona od 2 do 32 znaków alfanumerycznych, w celu uniknięcia problemów z identyfikacją użytkownika na serwerze.
Powiązane zasady biznesowe	ZU-03 Nazwa użytkownika to ciąg od 2 do 32 alfanumerycznych znaków.
Kryteria akceptacji	1. Po wpisaniu do pola użytkownika nazwy krótszej niż 2 znaki, dłużej niż 32 znaki lub zawierającej inne znaki niż alfanumeryczne, zwracany jest błąd.
Nakład godzinowy (planowany / włożony)	Czas wynikający z tego zadania ujęto przy realizacji zadania WF-01.
Ukończono?	Tak.

Identyfikator	HN-02
Treść	Loginy i hasła administratorów są gromadzone w plikach konfiguracyjnych, a po uruchomieniu serwera – w jego pamięci operacyjnej.
Powiązane zasady biznesowe	ZU-07 Konta administratorów są utrzymywane na serwerze w postaci par wartości: nazwa użytkownika i hasło.
Kryteria akceptacji	1. Serwer jest wyposażony w pliki konfiguracyjne 2. Po załadowaniu serwera, z plików konfiguracyjnych są odczytywane dane kont administracyjnych 3. Serwer po uruchomieniu jest wyposażony w konta o nazwach i hasłach zgodnych z wpisami w plikach konfiguracyjnych.
Nakład godzinowy (planowany / włożony)	Czas wynikający z tego zadania ujęto przy realizacji zadania WF-01.
Ukończono?	Tak.

3.3.2.3 Zadania wykraczające poza pierwotne wymagania

Identyfikator	ZZ-03
Treść	Połączenie po autoryzacji powinno zostać zabezpieczone losowym łańcuchem znaków – kluczem sesji. Klucz sesji powinien być generowany automatycznie po stronie backendu i wysyłany frontendowi tuż po autoryzacji użytkownika. Od tej pory backend nie zaakceptuje żadnej wiadomości od frontendu, dopóki nie będzie zawierała dodatkowego parametru `session_key` z treścią klucza sesji. Celem tego środka jest uniknięcie sytuacji zwielokrotnienia klientów na tym samym kanale w razie zresetowania lub wielokrotnego nawiązywania połączenia z tego samego urządzenia.
Kryteria akceptacji	1. W momencie autoryzacji, backend wygeneruje 32-znakowy, alfanumeryczny, losowy klucz. 2. Backend odeśle klucz z informacją o autoryzacji do frontendu. 3. Frontent zapisze i utrzyma otrzymany klucz sesji 4. Frontent będzie automatycznie dodawać parametr `session_key`, otrzymany podczas autoryzacji, do każdej kolejnej, dodawanej wiadomości.
Nakład godzinowy (planowany / włożony)	2h / 1h
Ukończono?	Tak.

3.3.3 Sprint 3

Termin realizacji: 24 marca – 6 kwietnia 2019 r.

3.3.3.1 Cel sprintu

Po zakończeniu sprintu, powinno być możliwe podpięcie się do pokoju i rozmowa z innymi użytkownikami, którzy są podpięci do tego samego pokoju.

3.3.3.2 Zadania oparte o pierwotne wymagania

Identyfikator	WF-02
Treść	Jako użytkownik serwera czatu, chcę wpiąć się do pokoju, aby wziąć udział w dyskusji.
Kryteria akceptacji	1. Użytkownik, który ma otwartą sesję połączenia z serwerem czatu i nie jest wpięty do żadnego pokoju, zobaczy listę pokojów. 2. Użytkownik, po kilknięciu w liście pokojów na nazwę pokoju, zostanie do niego podpięty 3. Użytkownik po wpięciu się do pokoju zobaczy okno pokoju 4. Użytkownik, który ma otwartą sesję połączenia z serwerem i jest wpięty do pokoju, po odświeżeniu przeglądarki zobaczy okno pokoju, do którego jest wpięty
Nakład godzinowy (planowany / włożony)	7h / 5h
Ukończono?	Tak.

Identyfikator	WF-03
Treść	Jako użytkownik serwera czatu, chcę po wpięciu do pokoju zobaczyć ostatnie wiadomości wysłane przed moim dołączeniem, aby dowiedzieć się, co tam się obecnie dzieje.
Kryteria akceptacji	1. Użytkownik po wpięciu się do pokoju zobaczy 10 najnowszych wiadomości wysłanych do pokoju przed jego dołączeniem (lub mniej, jeżeli dotychczas nie wysłano do pokoju co najmniej 10 wiadomości)
Nakład godzinowy (planowany / włożony)	6h / 5h
Ukończono?	Tak.

Identyfikator	WF-04
Treść	Jako użytkownik serwera czatu, chcę wysłać wiadomość do pokoju w który jestem wpięty, aby zobaczyli ją inni uczestnicy dyskusji.
Kryteria akceptacji	1. Użytkownik wpisze tekst wiadomości w polu tekstowym u dołu czatu 2. Wiadomość wpisana w polu tekstowym zostanie wysłana po wciśnięciu klawisza „Enter”, gdy aktywne będzie pole tekstowe 3. Wiadomość wpisana w polu tekstowym zostanie wysłana po naciśnięciu przycisku „Wyślij”, widocznego obok pola tekstowego 4. Po wysłaniu wiadomości, pole tekstowe zostanie wyczyszczone (niezależnie od tego czy wiadomość zostanie doręczona) 5. Wiadomość wysłana do pokoju jest pokazywana wszystkim użytkownikom podpiętym do czatu u dołu strony 6. Nowa wiadomość jest pokazywana wraz z nazwą użytkownika wysyłającego u dołu konwersacji
Nakład godzinowy (planowany / włożony)	3.5h / 5h
Ukończono?	Tak.

Identyfikator	WF-07
Treść	Jako użytkownik serwera czatu, chcę odpiąć się od pokoju, aby wpiąć się do innego pokoju.
Powiązane zasady biznesowe	ZP-06 Użytkownik może się samodzielnie wypiąć z pokoju, do którego jest wpięty
Kryteria akceptacji	1. W oknie pokoju użytkownik zobaczy przycisk lub link „Opuść pokój”. 2. Po kliknięciu w „Opuść pokój”, użytkownik zobaczy listę pokojów.
Nakład godzinowy (planowany / włożony)	2h / 1h
Ukończono?	Tak.

Identyfikator	HN-07
Treść	Bufor pokoju niebędącego dedykowanym do wiadomości prywatnych zawiera do 10 wiadomości.
Kryteria akceptacji	1. Po przekroczeniu liczby 10 wiadomości w pokoju, bufor ulega „zawinięciu", usuwając najstarsze wiadomości.
Nakład godzinowy (planowany / włożony)	Nakład czasowy ujęto podczas realizacji zadania WF-03.
Ukończono?	Tak.

3.3.3.3 Zadania wykraczające poza pierwotne wymagania

Brak.

3.3.4 Sprint 4

Termin realizacji: 7-27 kwietnia 2019 r.

3.3.4.1 Cel sprintu

Sprint ma na celu wykonanie mechanizmu prywatnych wiadomości. Wlicza się do niego lista użytkowników obecnych na czacie, z którymi ma być prowadzona rozmowa, a także nawiązywanie rozmowy w pokoju PW oraz wysyłanie i odbiór wiadomości w pokoju PW.

3.3.4.2 Zadania oparte o pierwotne wymagania

Identyfikator	WF-08
Treść	Jako użytkownik serwera czatu, chcę zobaczyć okno wiadomości prywatnych, aby odczytać wiadomości, które wysłano specjalnie do mnie.
Kryteria akceptacji	1. Po kliknięciu w link „PW", użytkownik zobaczy okno prywatnych wiadomości 2. W oknie wiadomości prywatnych, użytkownik zobaczy listę użytkowników, od których otrzymał wiadomości prywatne. 3. Po kliknięciu w link z nazwą użytkownika, użytkownik zobaczy prywatne wiadomości, których nadawcą i odbiorcą jest wskazana osoba.
Nakład godzinowy (planowany / włożony)	6h / 5h
Ukończono?	Tak.

Identyfikator	WF-09
Treść	Jako użytkownik serwera czatu, chcę wysłać wiadomość prywatną do jednego użytkownika, aby prowadzić z nim ciągłą konwersację.
Kryteria akceptacji	1. Użytkownik wpisze tekst wiadomości w polu tekstowym u dołu okna wiadomości prywatnych 2. Wiadomość wpisana w polu tekstowym zostanie wysłana po wciśnięciu klawisza „Enter”, gdy aktywne będzie pole tekstowe 3. Wiadomość wpisana w polu tekstowym zostanie wysłana po naciśnięciu przycisku „Wyślij”, widocznego obok pola tekstowego 4. Po wysłaniu wiadomości, pole tekstowe zostanie wyczyszczone (niezależnie od tego czy wiadomość zostanie doręczona) 5. Wiadomość wysłana w oknie zostanie pokazana tylko użytkownikowi, z którym trwa otwarta konwersacja 6. Nowa wiadomość jest pokazywana wraz z nazwą użytkownika wysyłającego u dołu konwersacji
Nakład godzinowy (planowany / włożony)	2h / 1h
Ukończono?	Tak.

Identyfikator	WF-11
Treść	Jako użytkownik serwera czatu, chcę wysłać wiadomość prywatną do innego użytkownika, z którym wcześniej nie wymieniałem takich wiadomości, aby rozpocząć z nim prywatną konwersację.
Kryteria akceptacji	1. Użytkownik kliknie w oknie wiadomości prywatnych w przyciski „Nowy”. 2. Użytkownik zobaczy monit o podanie nazwy użytkownika, z którym chce rozpocząć rozmowę 3. Jeżeli użytkownik jest aktywny, wówczas 4. Wiadomość wpisana w polu tekstowym zostanie wysłana po wciśnięciu klawisza „Enter”, gdy aktywne będzie pole tekstowe 5. Wiadomość wpisana w polu tekstowym zostanie wysłana po naciśnięciu przycisku „Wyślij”, widocznego obok pola tekstowego 6. Po wysłaniu wiadomości, pole tekstowe zostanie wyczyszczone (niezależnie od tego czy wiadomość zostanie doręczona) 7. Wiadomość wysłana w oknie zostanie pokazana tylko użytkownikowi, z którym trwa otwarta konwersacja 8. Nowa wiadomość jest pokazywana wraz z nazwą użytkownika wysyłającego u dołu konwersacji
Nakład godzinowy (planowany / włożony)	11h / 5h
Ukończono?	Nie – przeniesiono do kolejnego sprintu.

Identyfikator	HN-05
Treść	Wiadomości prywatne są czyszczone niezwłocznie po rozłączeniu się przez dowolnego z rozmówców.
Powiązane zasady biznesowe	ZW-10 Wiadomości prywatne są utrzymywane dopóki nadawca i odbiorca mają aktywną sesję na serwerze.
Kryteria akceptacji	1. Po zamknięciu sesji użytkownika, wiadomości prywatne których był nadawcą lub odbiorcą ulegają usunięciu.
Nakład godzinowy (planowany / włożony)	–
Ukończono?	Nie – zadanie przeniesiono do kolejnego sprintu.

Identyfikator	HN-06
Treść	Bufor pokoju wiadomości prywatnych zawiera do 100 wiadomości.
Powiązane zasady biznesowe	ZW-11 Dla każdej pary użytkowników, na serwerze jest gromadzone co najwyżej 100 wiadomości prywatnych.
Kryteria akceptacji	1. Po przekroczeniu liczby 100 wiadomości prywatnych w pokoju, bufor ulega „zawinięciu”, usuwając najstarsze wiadomości.
Nakład godzinowy (planowany / włożony)	–
Ukończono?	Nie – zadanie przeniesiono do kolejnego sprintu.

3.3.4.3 Zadania wykraczające poza pierwotne wymagania

Brak.

3.3.5 Sprint 5

Termin realizacji: 28 kwietnia – 11 maja 2019 r.

3.3.5.1 Cel sprintu

Celem jest dokończenie mechanizmu prywatnych wiadomości, których nie udało się dokończyć w poprzednim sprincie – łączenia z wybranym użytkownikiem, z którym wcześniej nie prowadzono korespondencji.

3.3.5.2 Zadania oparte o pierwotne wymagania

Identyfikator	WF-09
Treść	Jako użytkownik serwera czatu, chcę wysłać wiadomość prywatną do jednego użytkownika, aby prowadzić z nim ciągłą konwersację.
Kryteria akceptacji	1. Użytkownik wpisze tekst wiadomości w polu tekstowym u dołu okna wiadomości prywatnych 2. Wiadomość wpisana w polu tekstowym zostanie wysłana po wciśnięciu klawisza „Enter”, gdy aktywne będzie pole tekstowe 3. Wiadomość wpisana w polu tekstowym zostanie wysłana po naciśnięciu przycisku „Wyślij”, widocznego obok pola tekstowego 4. Po wysłaniu wiadomości, pole tekstowe zostanie wyczyszczone (niezależnie od tego czy wiadomość zostanie doręczona) 5. Wiadomość wysłana w oknie zostanie pokazana tylko użytkownikowi, z którym trwa otwarta konwersacja 6. Nowa wiadomość jest pokazywana wraz z nazwą użytkownika wysyłającego u dołu konwersacji
Nakład godzinowy (planowany / włożony)	4h / 5h
Ukończono?	Tak.

Identyfikator	WF-11
Treść	Jako użytkownik serwera czatu, chcę wysłać wiadomość prywatną do innego użytkownika, z którym wcześniej nie wymieniałem takich wiadomości, aby rozpocząć z nim prywatną konwersację.
Kryteria akceptacji	1. Użytkownik kliknie w oknie wiadomości prywatnych w przyciski „Nowy”. 2. Użytkownik zobaczy monit o podanie nazwy użytkownika, z którym chce rozpocząć rozmowę 3. Jeżeli użytkownik jest aktywny, wówczas 4. Wiadomość wpisana w polu tekstowym zostanie wysłana po wciśnięciu klawisza „Enter”, gdy aktywne będzie pole tekstowe 5. Wiadomość wpisana w polu tekstowym zostanie wysłana po naciśnięciu przycisku „Wyślij”, widocznego obok pola tekstowego 6. Po wysłaniu wiadomości, pole tekstowe zostanie wyczyszczone (niezależnie od tego czy wiadomość zostanie doręczona) 7. Wiadomość wysłana w oknie zostanie pokazana tylko użytkownikowi, z którym trwa otwarta konwersacja 8. Nowa wiadomość jest pokazywana wraz z nazwą użytkownika wysyłającego u dołu konwersacji
Nakład godzinowy (planowany / włożony)	3h / 5h
Ukończono?	Tak.

Identyfikator	HN-07
Treść	Bufor pokoju niebędącego dedykowanym do wiadomości prywatnych zawiera do 10 wiadomości.
Kryteria akceptacji	1. Po przekroczeniu liczby 10 wiadomości w pokoju, bufor ulega „zawinięciu", usuwając najstarsze wiadomości.
Nakład godzinowy (planowany / włożony)	Nakład czasu ujęto w zadaniu WF-11.
Ukończono?	Tak.

3.3.5.3 Zadania wykraczające poza pierwotne wymagania

Brak.

3.3.6 Sprint 6

Termin realizacji: 12-26 maja 2019 r.

3.3.6.1 Cel sprintu

W tym sprincie zaplanowano wykonanie narzędzi administracyjnych, służących do zarządzania serwerem.

3.3.6.2 Zadania oparte o pierwotne wymagania

Identyfikator	WF-12
Treść	Jako administrator, chcę utworzyć nowy pokój, aby umożliwić użytkownikom konwersację w węższym gronie.
Kryteria akceptacji	1. Administrator kliknie w oknie z listą pokojów w przycisk „Nowy". 2. Administrator zobaczy monit o podanie nazwy nowego pokoju. 3. Administrator po podaniu nazwy i zaakceptowaniu, zostanie przeniesiony do listy pokojów, na której będzie widoczna nazwa dodanego pokoju. 4. Administrator i inni użytkownicy mogą wpiąć się do nowoutworzonego pokoju.

Identyfikator	WF-13
Treść	Jako administrator, chcę usunąć zbędny pokój, aby utrzymać porządek na swoim serwerze czatu.
Kryteria akceptacji	1. Administrator wejdzie do pokoju, który chce usunąć. 2. Administrator zobaczy obok tytułu z nazwą pokoju przycisk „Usuń”. 3. Administrator po kliknięciu przycisku zobaczy monit z potwierdzeniem działania. 4. Administrator potwierdzi decyzję w monicie. 5. Po potwierdzeniu decyzji o usunięciu, administrator zostanie przeniesiony do listy pokojów, na której nie będzie już widniała nazwa usuniętego pokoju. 6. Pozostali użytkownicy w usuniętym pokoju zostaną niezwłocznie od niego odpięci i zobaczą monit systemowy informujący o usunięciu pokoju.
Nakład godzinowy (planowany / włożony)	2h / 3h
Ukończono?	Tak.

Identyfikator	WF-14
Treść	Jako administrator, chcę usunąć użytkownika z pokoju, aby utrzymać należyty poziom konwersacji.
Kryteria akceptacji	1. Administrator wejdzie do pokoju. 2. Administrator najedzie na nazwę użytkownika którego chce usunąć z pokoju i kliknie na przycisk z nazwą „Usuń z pokoju”. 3. Administrator po kliknięciu przycisku zobaczy monit z potwierdzeniem działania. 4. Administrator potwierdzi decyzję w monicie. 5. Po potwierdzeniu decyzji o usunięciu, administrator (tak samo jak każdy inny użytkownik podpięty do pokoju) zobaczy wiadomość systemową o usunięciu z konwersacji. 6. Usunięty użytkownik zostanie niezwłocznie wypięty z pokoju, a także zobaczy monit o przyczynie wypięcia.
Nakład godzinowy (planowany / włożony)	6h / 5h
Ukończono?	Tak.

Identyfikator	WF-15
Treść	Jako administrator, chcę usunąć użytkownika z serwera, aby ukarać go za łamanie zasad netykiety.
Kryteria akceptacji	1. Administrator wejdzie do pokoju. 2. Administrator najedzie na nazwę użytkownika którego chce usunąć z pokoju i kliknie na przycisk z nazwą „Usuń z serwera". 3. Administrator po kliknięciu przycisku zobaczy monit z potwierdzeniem działania. 4. Administrator potwierdzi decyzję w monicie. 5. Po potwierdzeniu decyzji o usunięciu, administrator (tak samo jak każdy inny użytkownik podpięty do pokoju) zobaczy wiadomość systemową o usunięciu z serwera. 6. Usunięty użytkownik zostanie niezwłocznie wypięty z pokoju i jego sesja zostanie zakończona, a także pokazany zostanie monit o przyczynie tych zdarzeń (usunięcie z serwera czatu).
Nakład godzinowy (planowany / włożony)	2h / 5h
Ukończono?	Tak.

Identyfikator	HN-03
Treść	Loginy administratorów w oknach czatu są pogrubione i pokolorowane na czerwono.
Kryteria akceptacji	1. Nazwy administratorów w oknach czatu są pogrubione i pokolorowane na czerowono.
Nakład godzinowy (planowany / włożony)	1h / 1h
Ukończono?	Tak.

Identyfikator	HN-04
Treść	Nazwy pokojów mają od 3 do 32 znaków alfanumerycznych długości, przy czym nigdy dwa pokoje nie mają tej samej nazwy.
Powiązane zasady biznesowe	ZP-02 Każdy pokój ma unikalną nazwę będącą ciągiem alfanumerycznym od 3 do 32 znaków.
Kryteria akceptacji	1. Nie jest możlwe utworzenie pokoju o nazwie, która już wcześniej się pojawiała 2. Nie jest możliwe utworzenie pokoju o nazwie krótszej niż 3 znaki i dłuższej niż 32 znaki. 3. Nie jest możliwe utworzenie pokoju o nazwie zawierającej znaki inne niż litery alfabetu łacińskiego, cyfry i znak podkreślenia.
Nakład godzinowy (planowany / włożony)	Nakład czasowy ujęto w ramach zadania WF-13.
Ukończono?	Tak.

3.3.6.3 Zadania wykraczające poza pierwotne wymagania

Brak.

3.3.7 Sprint 7

Termin realizacji: 27 maja – 1 czerwca 2019 r.

3.3.7.1 Cel sprintu

Celem sprintu jest wykonanie testów całego systemu, a także wykrycie i poprawienie niedoróbek.

3.3.7.2 Zadania oparte o pierwotne wymagania

Identyfikator	WF-05
Treść	Jako użytkownik serwera czatu, chcę chcę zobaczyć powiadomienie o wpięciu się nowego użytkownika do pokoju w którym sam jestem obecnie wpięty, aby powitać nowego dyskutanta
Kryteria akceptacji	1. Niezwłocznie po wpięciu się użytkownika do pokoju, serwer wyśle wiadomość systemową o treści „Użytkownik ... dołączył do pokoju", widoczną dla wszystkich użytkowników wpiętych do tego pokoju
Nakład godzinowy (planowany / włożony)	3h / 3h
Ukończono?	Tak.

Identyfikator	WF-06
Treść	Jako użytkownik serwera czatu, chcę zobaczyć powiadomienie o opuszczeniu pokoju przez użytkownika, aby łatwo zorientować się, że nie bierze już udziału w dyskusji.
Kryteria akceptacji	1. Niezwłocznie po wypięciu się użytkownika z pokoju, serwer wyśle wiadomość systemową, widoczną dla wszystkich użytkowników wpiętych do tego pokoju, o treści: (a) „Użytkownik ... opuścił pokój", gdy użytkownik samodzielnie wypiął się z pokoju (b) „Użytkownik ... stracił połączenie", gdy użytkownik został wypięty z pokoju na skutek przerwania sesji z uwagi na zerwanie połączenia (c) „Użytkownik ... został wyrzucony", gdy użytkownik został wypięty wskutek interwencji administratora
Nakład godzinowy (planowany / włożony)	2h / 5h
Ukończono?	Tak.

Identyfikator	WF-16
Treść	Jako administrator, chcę zmienić swoje hasło, aby zabezpieczyć swoje hasło w razie ujawnienia go osobie niepowołanej, bez zmiany plików konfiguracyjnych i restartowania całego serwera.
Kryteria akceptacji	1. Administrator wejdzie na kartę „Moje konto”. 2. Administrator kliknie na przycisk „Zmień hasło”, widoczny pod nazwą użytkownika. 3. Administrator zobaczy monit zmiany hasła, zawierający jedno pole tekstowe na stare hasło i dwa na nowe hasło (wszystkie trzy ukryte przed podglądaniem treści podczas ich wprowadzania). 4. Administrator potwierdzi decyzję o zmianie hasła w monicie. 5. Po potwierdzeniu decyzji, administrator zobaczy wiadomość systemową o zmianie hasła. 6. Administrator rozłączy się z serwerem. 7. Administrator spróbuje rozpocząć nową sesję z serwerem, autoryzując się nowym hasłem. 8. Nowe hasło zostanie zaakceptowane przez serwer, sesja zostanie rozpoczęta prawidłowo.
Nakład godzinowy (planowany / włożony)	2h / 2h
Ukończono?	Tak.

Identyfikator	WS-03
Treść	Aplikacja czatu będzie dostosowana przede wszystkim do obsługi z wykorzystaniem urządzeń mobilnych.
Nakład godzinowy (planowany / włożony)	2h / 0h
Ukończono?	Nie – idea zarzucona, pozostano przy wersji dla ekranów jednego typu urządzenia - smartfonu. Pozostałe ekrany pokazują przeskalowany interfejs.

3.3.7.3 Zadania wykraczające poza pierwotne wymagania

Brak.

Część II

Język ViuAct i jego kompilator

Rozdział 4

Język ViuAct i jego kompilator – założenia

W tym rozdziale prezentujemy wszelkie „formalności" związane z pierwszą częścią naszej pracy. Opis projektu języka i jego implementacji prezentujemy w rozdziale Język ViuAct i jego kompilator – implementacja na stronie 63. Zakresem tych formalności jest:

- analiza otoczenia, wraz z klientami
- wskazanie kontekstu biznesowego systemu
- określenie udziałowców

4.1 Język ViuAct i jego kompilator w kontekście

4.1.1 Kontekst biznesowy

Docelowym środowiskiem języka ViuAct są urządzenia i programy, których zachowanie ma charakteryzować się wysoką niezawodnością i stabilnością (np. urządzenia telekomunikacyjne, oprogramowanie monitorujące i diagnostyczne).

4.1.2 Charakterystyka użytkowników

Docelowym użytkownikiem języka i kompilatora ViuAct jest **programista**. Oczekuje on od języka narzędzi pozwalających na tworzenie programów odpornych na awarie, poprawnych, i prostych w utrzymaniu.

Docelowym użytkownikiem platformy, na której wdrażane są programy napisane w języku ViuAct (czyli platformy prezentowanej przez Viua VM) jest **administrator**. Oczekuje on od platformy przejrzystego modelu wdrożenia i konserwacji instalacji, oraz możliwości monitorowania zachowania wdrożonego oprogramowania.

4.1.3 Istniejąca infrastruktura

Język ViuAct wchodzi w interakcje z istniejącą infrastrukturą poprzez narzędzia, kanały, i protokoły prezentowane przez platformę Viua VM. Zakłada się, że „istniejąca infrastruktura" w *miejscu wdrożenia* działającego programu napisanego w języku ViuAct jest zgodna z wymaganiami i pozwala na działanie platformy Viua VM. Zakłada się, że „istniejąca infrastruktura" w *miejscu wytwarzania* programu w języku ViuAct jest zgodna z wymaganiami i pozwala na działanie platformy Viua VM, oraz jest zgodna z wymaganiami kompilatora języka ViuAct.

4.2 Wymagania

Wymaganie precyzują cele projektu i określają założenia tego co musi (lub nie musi) zostać wykonane, aby projekt został uznany za zakończony sukcesem. Zostały opracowane przez członków zespołu z uwzględnieniem uwag udziałowców.

4.2.1 Wymagania ogólne i dziedzinowe

Identyfikator	WD-01
Priorytet	S
Nazwa	Udowodnienie przydatności (ang. *viability*) Viua VM
Opis	Udowodnienie, że maszyna wirtualna Viua może być celem kompilacji dla języków wyższego poziomu oraz jest możliwe uruchomienie na niej nietrywialnego oprogramowania (w przypadku tego projektu będzie to ViuaChat).
Udziałowiec	Promotor, Uczelnia, członkowie zespołu
Wymagania powiązane	

Identyfikator	WD-02
Priorytet	M
Nazwa	Projekt języka ViuAct
Opis	Wymagane jest zaprojektowanie języka programowania wyższego poziomu oraz przygotowanie jego specyfikacji.
Udziałowiec	Promotor, Uczelnia, członkowie zespołu
Wymagania powiązane	WD-01

Identyfikator	WD-03
Priorytet	M
Nazwa	Implementacja kompilatora języka ViuAct
Opis	Wymagana jest implementacja kompilatora przetwarzającego kod źródłowy napisany w języku wyższego poziomu (zaprojektowanym w punkcie WD-02) na kod w języku assemblera Viua VM.
Udziałowiec	Promotor, Uczelnia, członkowie zespołu
Wymagania powiązane	WD-02

Identyfikator	WD-04
Priorytet	M
Nazwa	Chat
Opis	Musi powstać implementacja czatu napisana w języku zaprojektowanym w punkcie WD-02.
Udziałowiec	Promotor, Uczelnia, członkowie zespołu
Wymagania powiązane	WD-03 (*do wykonania tego wymagania niezbędny jest kompilator*)

4.2.2 Wymagania funkcjonalne

Część wymagań funkcjonalnych języka ViuAct jest opisana w rozdziale 5. Specyfikacja języka ViuAct na stronie 43. Poniżej opisane są wymagania, które nie mogły być zawarte bezpośrednio w specyfikacji języka (np. sposób obsługi i raportowania błędów przez kompilator). Interfejs kompilatora jest opisany w rozdziale 6.8 na stronie 81.

4.2.2.1 Funkcjonalności kompilatora

W tej sekcji opisane są wymagania dotyczące kompilatora języka ViuAct.

Identyfikator	WF-01
Priorytet	M
Nazwa	Zachowanie znaczenia programu podczas kompilacji
Opis	Kompilator musi wygenerować kod wynikowy implementujący dokładnie taki sam program jak oryginalny kod źródłowy. Niedopuszczalne jest aby po skompilowaniu program miał inne zachowanie niż to, które zostało opisane przez oryginalny kod źródłowy w języku wyższego poziomu. Całkowite zachowanie znaczenia oznacza również, że jeśli oryginalny kod źródłowy zawiera błąd logiczny to program wynikowy również będzie go zawierać.
Udziałowiec	Promotor, Uczelnia, członkowie zespołu
Wymagania powiązane	

Identyfikator	WF-02
Priorytet	W
Nazwa	Obsługa błędów
Opis	Kompilator nie będzie w żaden sposób obsługiwał błędów ani ich raportował. Strategią obsługi błędów będzie rzucenie wyjątku, ewentualnie dodając do niego informacje o tym co konkretnie się wydarzyło, i pozwolenie mu na zabicie procesu kompilatora. Na podstawie zrzutu stosu i treści wyjątku użytkownik będzie musiał się domyślić co wywołało błąd.
Udziałowiec	Członkowie zespołu
Wymagania powiązane	

Identyfikator	WF-03
Priorytet	M
Nazwa	Emisja plików pomocniczych
Opis	Kompilator podczas przetwarzania kodu źródłowego musi zebrać informacje jakie funkcje zawiera dany moduł, jakie moduły są w nim zagnieżdżone, i jakie moduły są przez niego importowane. Te informacje są potrzebne programowi łączącemu moduły do automatycznego wykonania swojej pracy. Pliki pomocnicze to: 1. `.i` pliki opisujące interfejs modułu (eksportowane funkcje) 2. `.d` pliki opisujące zależności modułu (importowane moduły)
Udziałowiec	Promotor, Uczelnia, członkowie zespołu
Wymagania powiązane	

Identyfikator	WF-04
Priorytet	C
Nazwa	Optymalizacja zużycia rejestrów
Opis	Z uwagi na prostą budowę kompilatora generuje on kod, który alokuje duże ilości rejestrów. Optymalizacja polegająca na analizie dostępu do rejestrów (odczytach i zapisach) i wyszuaniu momentów, po których rejestr `x` nie jest już więcej używany mogłaby zredukować rozmiar alokacji dzięki recyklingowi rejestrów (ponownym użyciu po udowodnieniu, że po danej instrukcji żadna inna nie będzie korzystać z wartości w danym rejestrze).
Udziałowiec	Członkowie zespołu
Wymagania powiązane	

Identyfikator	WF-05
Priorytet	M
Nazwa	Kompletność implementacji
Opis	Kompilator musi implementować wszystkie konstrukcje językowe opisane w specyfikaji języka ViuAct.
Udziałowiec	Promotor, Uczelnia, członkowie zespołu
Wymagania powiązane	

Wymaganie WF-01 czyli „Zachowanie znaczenia programu podczas kompilacji" jest trudne do spełnienia, ponieważ nie jesteśmy w stanie przetestować wszystkich programów możliwych do napisania w języku ViuAct. Z tego powodu, między innymi, testy kompilatora języka ViuAct zostały wykonane jako testy pojedynczych konstrukcji językowych – np. konstrukcji warunkowych i wyrażeń złożonych.

4.2.2.2 Interfejs z otoczeniem

Punkty styku projektowanego języka i kompilatora z innymi podsystemami (np. assembler dostarczany jako część Viua VM), sieciami (np. Internet), i operatorami (np. programista).

Identyfikator	IZO-01
Priorytet	M
Nazwa	Poprawność emitowanego kodu
Opis	Kompilator musi generować poprawny kod źródłowy w języku assemblera Viua VM. Niedopuszczalne jest aby kompilator generował kod, który będzie zawierał błędy składniowe bądź nieznane lub niedozwolone instrukcje.
Udziałowiec	Promotor, Uczelnia, członkowie zespołu
Wymagania powiązane	

Identyfikator	IZO-02
Priorytet	M
Nazwa	Biblioteki implementujące operacje I/O
Opis	Muszą być dostarczone biblioteki umożliwiające przeprowadzanie operacji I/O (*wejścia-wyjścia*). Takie biblioteki mogą być napisane w języku ViuAct, języku assemblera Viua VM, lub języku C++. W przypadku implementacji w języku assemblera Viua VM lub języku C++ muszą zostać dostarczone pliki umożliwiające kompilatorowi ViuAct wczytanie listy funkcji oferowanych przez taką bibliotekę.
Udziałowiec	Członkowie zespołu
Wymagania powiązane	

4.2.3 Wymagania dotyczące procesu wytwarzania

W tej sekcji opisane są wymagania dotyczące kompilatora języka ViuAct.

Identyfikator	PW-01
Priorytet	M
Nazwa	Testy kompilatora
Opis	Kompilator musi być dostarczony wraz z zestawem prostych do uruchomienia testów werfikujących poprawność jego pracy.
Udziałowiec	Promotor, Uczelnia, członkowie zespołu
Wymagania powiązane	

Identyfikator	PW-02
Priorytet	S
Nazwa	Kompletność testów kompilatora
Opis	Testy kompilatora powinny obejmować jak największe spektrum możliwych konstrukcji językowych i ich kombinacji. Z uwagi na to, że jest niemożliwym przetestowanie kompilatora na wszystkich możliwych do napisania programach „kompletność" jest względna, a to wymaganie zakłada jedynie, że nie powinna istnieć konstrukcja językowa, która nie będzie przetestowana. **Uwaga**: Ostatecznym testem kompilatora pozostaje kompilacja czatu ViuaChat, który jest wymagany do poprawnego zaliczenia projektu.
Udziałowiec	Promotor, Uczelnia, członkowie zespołu
Wymagania powiązane	

4.3 Kryteria akceptacji rozwiązania

Podstawowym kryterium akceptacji dla języka jest „*Da się w nim napisać czat, który spełnia wymagania opisane w rozdziale 7.4*", a dla kompilatora „*Da się nim skompilować czat*".

Rozdział 5

Specyfikacja języka ViuAct

ViuAct jest językiem kładącym silny nacisk na funkcjonalność związaną z współbieżnością opartą na modelu aktorów. Większość konstrukcji językowych jest wyrażeniami (*expressions*) zamiast „stwierdzeniami" (*statements*). Składnia języka przypomina składnię języka Lisp.

Najciekawsze funkcjonalności języka ViuAct to:

1. aktory[1]
2. przekazywanie wiadomości
3. system modułów
4. funkcje zagnieżdżone i niemutowalne domknięcia
5. dowiązania `let`

Aktory Aktory są *wewnętrznie sekwencyjnymi* (każdy aktor wykonuje sekwencyjny strumień instrukcji bez jakiejkolwiek współbieżności), izolowanymi od siebie procesami, które komunikują się za pomocą *przekazywania wiadomości*.

Przekazywanie wiadomości Przekazywanie wiadomości w ViuAct jest *bezpośrednie* (wiadomości są przekazywane bezpośrednio między dwoma aktorami, bez użycia jakiegokolwiek jawnego kanału pośredniczącego), *asynchroniczne* (wiadomość jest wysyłana, a kontrola natychmiast wraca do aktora nadawcy bez oczekiwania na odpowiedź od odbiorcy). Przekazywanie wiadomości nie daje gwarancji dostarczenia ani kolejności – implementacja protokołu zapewniającego poprawność komunikacji leży po stronie „przestrzeni użytkownika".

System modułów ViuAct zapewnia programiście system modułów. Moduły są automatycznie kompilowane i linkowane, w większości z poziomu języka, co redukuje poziom skomplikowania konfiguracji systemów budowania projektu.

Funkcje zagnieżdżone i niemutowalne domknięcia Funkcje w ViuAct mogą być zagnieżdżane, i tworzyć niemutowalne domknięcia (ang. *closures*) – funkcje, które odwołują się do zmiennych zdefiniowanych w otaczającej je przestrzeni leksykalnej.

Dowiązania `let` Zmienne w ViuAct są typowane dynamicznie (czyli zmienna może być dowiązana do wartości wielu typów wewnątrz jednej funkcji), a wartości statycznie (każda wartość ma konkretny typ, który jest śledzony w czasie kompilacji). Dowiązania `let` do jednej nazwy mogą być tworzone wiele razy, na przykład:

[1] Dlaczego używamy słowa „aktory" zamiast „aktorzy" wyjaśnione jest na stronie 7.

```
(let conf (Config.from_file "/etc/some.config"))
(let conf (Config.load_section conf "daemon"))
```

Po ponownym dowiązaniu nie jest możliwe dotarcie do poprzednio dowiązanej wartości, ale użycia zmiennej „przed" ponownym dowiązaniem zachowują swoje znaczenie (*prior art*: na takiej samej zasadzie działają dowiązania `let` w językach OCaml i Rust).

5.1 Model programowania

5.1.1 Podział programu

Najmniejszym elementem możliwym do wykonania jest wyrażenie. Wyrażenia proste (*simple expression*) są grupowane w wyrażenia złożone (*compound expression*). Funkcje parametryzują wyrażenia. Funkcje są zebrane w modułach.

5.1.2 Wykonywanie programu

Program składa się z jednego lub większej liczby „aktorów" (procesów). Aktory są izolowane od siebie nawzajem i działają równolegle (wszystkie naraz). Wewnętrznie, każdy z aktorów wykonywany jest sekwencyjnie, bez jakiejkolwiek współbieżności. Wykonanie programu rozpoczyna się od funkcji `main`.

5.1.3 Komunikacja między aktorami

Aktory komunikują się ze sobą za pomocą „wymiany wiadomości". Wymiana wiadomości ma następujące cechy:

1. jest asynchroniczna (nadawca nie jest blokowany do momentu aż odbiorca otrzyma wiadomość)
2. nie daje gwarancji dostarczenia (nadawca nie ma gwarancji, że jego wiadomość dotrze do odbiorcy)
3. nie daje gwarancji kolejności dostarczenia (nadawca nie ma gwarancji w jakiej kolejności wysłane przez niego wiadomości dotrą do odbiorcy)
4. wiadomością może być dowolna wartość, która może być reprezentowana w języku ViuAct. Typy wartości opisane są w rozdziale 5.2 na stronie 44.

Ten model wymiany wiadomości jest warunkowany faktem, że nie jest fizycznie możliwe zagwarantowanie poprawności komunikacji.

5.1.4 Obsługa błędów

Obsługa błędów jest realizowana za pomocą wyjątków. ViuAct oferuje typową składnię `try-catch`.

5.2 Typy danych

Ten rozdział opisuje typy danych jakie mogą być używane w języku ViuAct. Każda wartość użyta w programie napisanym w języku ViuAct jest wartością jednego z typów wymienionych w tym rozdziale – czasem będąc kombinacją niektórych z nich, na przykład „wskaźnik do liczby całkowitej" lub „wektor PID-ów".

5.2.1 Typy proste

5.2.1.1 Liczba całkowita

Liczba całkowita ze znakiem, o szerokości 64 bitów, kodowana z dopełnienem do dwóch. Liczba całkowita jest typem liczbowym.

Przy konwersji do typu boolowskiego przyjmuje fałsz dla wartości całkowitoliczbowej `0`, a prawdę dla pozostałych wartości.

5.2.1.2 Liczba zmiennoprzecinkowa

Liczba zmiennoprzecinkowa podwójnej precyzji o szerokości 64 bitów kodowana w standardzie IEEE 754-2008. Liczba całkowita jest typem liczbowym.

Przy konwersji do typu boolowskiego przyjmuje fałsz dla wartości zmiennoprzecinkowej `0.0`, a prawdę dla pozostałych wartości.

5.2.1.3 Wartość boolowska

Wartość logiczna o szerokości 8 bitów reprezentująca prawdę lub fałsz. Prawda jest reprezentowana jako wartość `0x01` (najmniej znaczący bit „włączony"), a fałsz jako `0x00`.

5.2.1.4 String

Ciąg znaków Unicode, kodowany w formacie UTF-8. Maksymalna (teoretyczna) długość stringu to 2^{64} znaków.

Przy konwersji do typu boolowskiego przyjmuje fałsz dla wartości reprezentujących puste napisy (napisy o długości 0), a prawdę dla pozostałych wartości.

5.2.2 Typy złożone

5.2.2.1 Wektor

Sekwencja wartości dowolnych typów danych, indeksowana od 0. Maksymalna długość wektora to 2^{64} elementów. Wektory zawierają elementy jednego typu. Program, który wykorzystuje wektory przechowujące elementy różnych typów jest niepoprawny.

Przy konwersji do typu boolowskiego przyjmuje fałsz dla wartości reprezentujących puste wektory (wektory o długości 0), a prawdę dla pozostałych wartości.

5.2.2.2 Struktura

Struktura agregująca wartości dowolnych (różnych) typów danych. Struktury ewoluują dynamicznie, a przypisanie wartości do pól struktury odbywa się w czasie wykonywania.

Przy konwersji do typu boolowskiego przyjmuje fałsz dla wartości reprezentujących puste struktury (struktury nie mające żadnego pola), a prawdę dla pozostałych wartości.

5.2.3 Typy referencyjne

5.2.3.1 Wskaźnik

Wskaźnik umożliwia niebezpośrednie przekazanie informacji o pewnej innej wartości i może być użyty do odczytania bądź zmodyfikowania wartości, na którą wskazuje.

W przeciwieństwie do wskaźników w językach takich jak C i C++ wskaźniki w języku ViuAct są bezpieczne – nie wskazują na lokalizacje w pamięci lub rejestry, a na *wartości*.

Przy konwersji do typu boolowskiego przyjmuje fałsz dla wartości reprezentujących „martwe" wskaźniki (wskaźniki wskazujące na wartość, która już nie istnieje), a prawdę dla pozostałych wartości.

5.2.4 Typy platformy

5.2.4.1 PID

Wartość identyfikująca aktora w programie. Jest to typ dostarczany przez Viua VM. Dla programów pisanych w ViuAct jest to typ nieprzezroczysty[2].

Przy konwersji do typu boolowskiego przyjmuje fałsz dla wszystkich wartości. Używanie wartości typu PID jako wartości typu boolowskiego jest niezalecane ponieważ taka konwersja nie ma sensu praktycznego, a udostępniana jest jedynie dla zachowania spójności języka.

[2]Typ nieprzezroczysty czyli tzw. *opaque type* to typ, którego wartości nie mogą być analizowane i modyfikowane bezpośrednio z poziomu języka.

5.2.4.2 `void` – typ pusty

Typ pusty jest niemożliwy do użycia w języku ViuAct. Funkcje, które go zwracają pochodzą z bibliotek „obcych" lub z grupy funkcji wbudowanych. W ViuAct niemożliwe jest napisanie funkcji, która zwraca wartość typu `void`.

5.3 Konstrukcje języka

W tym rozdziale zaprezentowane są wszystkie konstrukcje językowe dostępne w języku ViuAct. Schemat opisu każdego z wyrażeń jest następujący:

1. abstrakcyjna składnia w formacie EBNF
2. opis wyrażenia
3. przykład kodu wykorzystujący dane wyrażenie

Przykład kodu to zazwyczaj kilka przykładów jak dana konstrukcja może wyglądać, nie kawałek działającego programu.

5.3.1 Wyrażenia

```
expression := literal
            | name
            | fn-call
            | actor-call
            | operator-call
            | try-expr
            | if-expr
expression := { expression+ }
```

Wyrażenia zwracają wartość, którą reprezentują. W przypadku literałów (linie od 1 do 4 na listingu 5.1) jest to wartość, którą bezpośrednio reprezentują; w przypadku innych wyrażeń (linie od 5 do 8 na listingu 5.1) wartość ta jest określana na etapie wykonania zależnie od wcześniejszego stanu programu. Dla wyrażeń złożonych (listing 5.2) obowiązują specjalne zasady opisane w rozdziale 5.3.1.1 na stronie 46.

Większość konstrukcji jezykowych oferowanych przez ViuAct może być użyta jako wyrażenie. Wyjątki to definicje zmiennych, definicje funkcji, definicje modułów, i przypisanie do pola struktury.

Listing 5.1: Przykłady wyrażeń

```
42                                          1
true                                        2
3.14                                        3
"Hello World!"                              4
x                                           5
(print x)                                   6
(actor server_loop sock)                    7
(+ x 1)                                     8
```

5.3.1.1 Wyrażenia złożone

Wyrażenia złożone są wyrażeniami składającymi się z kilku wyrażeń ograniczonych nawiasami klamrowymi. Wyrażenia składowe wyrażenia złożonego są wykonywane po kolei, a końcowa wartość wyrażenia złożonego jest wartością ostatniego wyrażenia składowego.

Wartością zwracaną wyrażenia złożonego przedstawionego poniżej będzie `42`:

Listing 5.2: Wyrażenie złożone

```
{
    (let x 42)
    (print x)
    x    ; wartość zwracana z tego wyrażenia jest wartością
         ; przechowywaną pod nazwą x
}
```

Wyrażenia złożone mogą być dowolnie zagnieżdżane. Mimo iż składają się z potencjalnie wielu wyrażeń wyrażenia złożone są traktowane jako jedno wyrażenie – mogą być wykorzystywane wszędzie tam gdzie składnia języka wymaga użycia pojedynczego wyrażenia (definicje funkcji, wyrażenia kontrolne w konstrukcji warunkowej, wyrażenia obsługi w konstrukcji `try`, itp.). Na przykładzie poniżej prezentowane jest dowiązanie *`let`* (przedstawione dokładniej w rozdziale 5.3.2) do wartości wyrażenia złożonego:

```
(let x {
    (let z (Std.Random.random))
    (print z)
    z
})
```

5.3.2 Definicje zmiennych

`let-binding := (let` *`name expression`*`)`

Definicje zmiennych przypisują wartości wyrażeń do nazw.

```
(let i 42)
(let f -3.14)
(let t "Hello World!")
```

5.3.3 Wywołanie operatora

(*`operator lhs rhs`*)

Operatory są zapisywane w konwencji prefiksowej (tj. najpierw operator, potem operandy).

```
(+ x 1)
(* a b)
```

5.3.3.1 Operatory binarne (dwuargumentowe) logiczne

Binarne operatory logiczne pobierają dwie wartości, i produkują wartość logiczną.

`or` pobiera dwie dowolne wartości; `(or a b)`

`and` pobiera dwie dowolne wartości; `(and a b)`

`=` pobiera dwie wartości liczbowe; `(= a b)`

`!=` pobiera dwie wartości liczbowe; `(!= a b)`

`<` pobiera dwie wartości liczbowe; `(< a b)`

`<=` pobiera dwie wartości liczbowe; `(<= a b)`

`>` pobiera dwie wartości liczbowe; `(> a b)`

`>=` pobiera dwie wartości liczbowe; `(>= a b)`

5.3.3.2 Operatory binarne (dwuargumentowe) arytmetyczne

Operatory arytmetyczne pobierają dwie wartości liczbowe i produkują wartość liczbową zgodną z typem operandu lewej strony (*left-hand side*).

```
let op : Ta -> Tb -> Ta
```

Przed wykonaniem operacji wartość operandu prawej strony (*right-hand side*) jest konwertowana na wartość odpowiadającą typowi operandu lewej strony. Liczby zmiennoprzecinkowe są zaokrąglane w dół (*floor*) podczas konwersji na liczby całkowite.

`+` dodaje dwie wartości liczbowe

`-` odejmuje dwie wartości liczbowe, *`rhs`* jest odejmowany od *`lhs`*

`*` mnoży dwie wartości liczbowe

`/` dzieli dwie wartości liczbowe, *`lhs`* jest dzielony przez *`rhs`*

5.3.3.3 Operator dostępu

`id := `*`name`*
`id := `*`id . name`*

```
(let x a_struct.some_field)
(let sock (Std.Posix.Network.socket))
```

`.` operator dostępu do składowych modułów i pól struktur, w zależności od kontekstu

5.3.3.4 Operatory unarne (jednoargumentowe)

`not` negacja logiczna, pobiera dowolną wartość i produkuje jej odwrotność logiczną

`^` dereferencja wskaźnika

5.3.4 Przypisanie do pola struktury

`(:= `*`field expression`*`)`

Operator przypisania do pola struktury powoduje utworzenie żądanego pola w strukturze (jeśli wcześniej nie istniało) lub aktualizację wartości przechowywanej w tym polu (jeśli wcześniej istniało).

```
(let x (struct))
(:= x.field 42)
```

5.3.5 Definicje funkcji

```
function-definition := (let name (formal-parameters) expression)
```

O funkcjach w ViuAct można myśleć jak o parametryzowanych wyrażeniach.

```
(let f (x) (+ x 1))
(let f_with_print (x) {
    (print x)
    (+ x 1)
})
(let apply (f x) (f x))
```

5.3.5.1 Parametry formalne

```
formal-parameters := name*
```

Parametry formalne (*formal parameters*) są nazwami zmiennych widocznymi wewnątrz funkcji. Parametry faktyczne (*actual parameters*) są wartościami przypisywanymi parametrom formalnym na etapie wykonania przez wywołującego. Parametry formalne przyjęło się nazywać *parametrami*, a parametry faktyczne *argumentami*.

Powyższe definicje zostały zaczerpnięte z rozdziału 8.3 „Parameter Passing” w [4].

5.3.5.2 Przekazywanie parametrów

Przekazywanie parametrów do funkcji odbywa się przez kopiowanie (*call-by-value* w [4]). Kopia każdego argumentu (parametru faktycznego) jest dowiązywana (jak w dowiązaniach *let*) do nazwy odpowiadającego mu parametru (parametru formalnego).

Możliwe jest przekazanie „przez referencję” (*call-by-reference* w [4]) za pomocą wskaźników – w takim wypadku to wskaźnik będzie skopiowany, nie wartość, na którą on wskazuje.

5.3.5.3 Ciało funkcji

Ciało funkcji jest reprezentowane jednym wyrażeniem (może to być wyrażenie złożone). Wyrażenia są opisane w rozdziale 5.3.1 na stronie 46.

5.3.5.4 Wartość zwracana

Wartość zwracana z funkcji jest wartością, do której ewaluuje wyrażenie będące jej ciałem.

5.3.5.5 Rekurencja

W języku ViuAct nie istnieją pętle. Sposobem na uzyskanie iteracji jest rekurencja, ze szczególnym uwzględnieniem rekurencji ogonowej (opisanej w rozdziale 5.3.8 na stronie 51).

5.3.6 Definicje modułów

```
inline-module-definition := (module module-name (
                             module-definition*
                             module-import*
                             function-definition+ ))
module-definition := module-declaration
                   | inline-module-definition
module-declaration := (module module-name)
module-import := (import module-name)
```

Moduł jest zbiorem definicji modułów, deklaracji import, i definicji funkcji. Taka kolejność w kodzie źródłowym nie jest wymagana, ale kompilator w takiej kolejności przetwarza konstrukcje składające się na definicję modułu.

5.3.6.1 Definicje modułów w plikach

Moduł jest definiowany w pliku, a nazwa pliku przekłada się na nazwę modułu. Dla przykładu: plik `A_module.lisp` definiuje moduł o nazwie `A_module`.

Relację zagnieżdżania pomiędzy modułami wprowadza się poprzez umieszczenie definicji lub deklaracji modułu-dziecka w definicji modułu-rodzica. Dla modułów zdefiniowanych w osobnych plikach używa się deklaracji modułów.

Listing 5.3: Plik `src/A_module.lisp`

```
(module Nested_module)

(let fn () {
    (let s "Hello, World! (from A_module)")
    (print s)
    0
})
```

Listing 5.4: Plik `src/A_module/Nested_module.lisp`

```
(let fn () {
    (let s "Hello, World! (from A_module.Nested_module)")
    (print s)
    0
})
```

W przykładzie powyżej moduł `Nested_module` jest modułem zagnieżdżonym w module `A_module` i musi być importowany za pomocą wyrażenia `(import A_module.Nested_module)`.

5.3.6.2 Definicje modułów *inline*

Moduł jest definiowany w tym samym pliku co jego moduł-rodzic. Relację zagnieżdżenia pomiędzy modułami widać bezpośrednio – jeden moduł jest dosłownie zagnieżdżony wewnątrz innego modułu.

Listing 5.5: Plik `src/A_module.lisp`

```
(module Nested_module (
    (let fn () {
        (let s "Hello, World! (from A_module.Nested_module)")
        (print s)
        0
    })
))

(let fn () {
    (let s "Hello, World! (from A_module)")
    (print s)
    0
})
```

W przykładzie powyżej moduł `Nested_module` jest modułem zagnieżdżonym w module `A_module` i musi być importowany za pomocą wyrażenia `(import A_module.Nested_module)`.

5.3.6.3 Importowanie modułów

Importowanie modułów jest realizowane przez wyrażenie `import`. Po słowie kluczowym `import` należy umieścić pełną ścieżkę do importowanego modułu, na przykład:

Listing 5.6: Plik `src/A_module.lisp`

```
(import A_module)
(import A_module.Nested_module)
(import Std.Posix.Network)
```

Dostęp do funkcji i wyliczeń zdefiniowanych wewnątrz modułu odbywa się z użyciem pełnej ścieżki, na przykład `(Std.Posix.Network.socket "127.0.0.1" 8080)`.

5.3.7 Wywołanie funkcji

`fn-call := (`*id expression**`)`

Wywołanie funkcji jest reprezentowane przez podanie jej nazwy w nawiasach okrągłych, po której podane jest zero lub więcej wyrażeń, których wartości zostaną przypisane do odpowiednich parametrów aktualnych.

Funkcje, których definicje znajdują się w innych modułach niż aktualnie przetwarzany muszą być poprzedzone pełną ścieżką do modułu, w którym są zdefiniowane (np. `Std.Posix.Network.socket`). Jedynie funkcje znajdujące się w tym samym module muszą być wywoływane bez pełnej ścieżki.

5.3.8 Wywołanie „tailcall”

`(tailcall` *id expression** `)`

Wywołania *tailcall* powodują wywołanie funkcji „wykorzystując” ramkę obecnej funkcji. W praktyce powodują natychmiastowe zdjęcie obecnej ramki ze stosu (co powoduje destrukcję wszystkich wartości w rejestrach lokalnych obecnej ramki, oraz uruchomienie wszelkich wywołań odroczonych) i wepchnięcie na szczyt stosu ramki dla funkcji wywołanej *tailcall*.

Wywołanie *tailcall* nie ma bezpośredniej wartości zwracanej, ponieważ nie da się jej zaobserwować. Wywołanie *tailcall* „porywa” (ang. „hijacks”) punkt, do którego powinna zwrócić wartość funkcja, która wykonała takie wywołanie – i dopiero ten punkt zaobserwuje de facto wartość zwróconą przez wywołanie *tailcall*, ale z jego punktu widzenia wygląda to tak jakby tą wartość zwróciła funkcja, którą on bezpośrednio wywołał.

```
(let g (x) (+ x 1))

(let f (x) (tailcall g x))  ; ramka wywołania funkcji f()
                            ; zastępowana ramką g()
(let fa (x) (g x))          ; funkcja f() bez tail cal

(let main () {
    (let i (f 41))  ; w tym miejscu "i" jest przypisane do
                    ; wyniku g()
    0
})
```

Podmiana ramki jest niedostrzegalna dla wywołującego oryginalną funkcję.

5.3.8.1 Zastosowania

Wywołania *tailcall* są wykorzystywane do implementacji iteracji (co jest niezbędne przy przetwarzaniu wektorów), oraz przy obsłudze błędów za pomocą funkcji *watchdog* (opisanych w rozdziale 5.3.12 na stronie 54) gdzie pozwala na podmianę funkcji głównej bez zmiany PID aktora.

Wywołania *tailcall* są również przydatne w typowym wzorcu wykorzystywanym w języku ViuAct czyli *init-impl* – „inicjalizacja-implementacja". Funkcja inicjalizująca tworzy i konfiguruje nowego aktora, a następnie wykonuje wywołanie w pozycji ogonowej do funkcji implementacji. Po zakończeniu inicjalizacji funkcja przeprowadzająca ją nie jest już potrzebna więc jej ramka może zostać zastąpiona ramką innej funkcji bez żadnego problemu.

5.3.9 Wywołanie odroczone

`(defer` *`name expression*`*`)`

Wywołania odroczone nie są wywoływane od razu (jak zwykłe wywołania funkcji, bądź wywołania *tailcall*), ale dopiero w momencie, w którym ramka, w której zostały zarejestrowane jest zdejmowana ze stosu, niezależnie od powodu dla którego jest ona zdejmowana – czy będzie to zwykły powrót, odwinięcie stosu wywołane przez wyjątek, czy wywołanie *tailcall*.

Wywołania odroczone są wykonywane w odwrotnej kolejności rejestracji. Jeśli odroczone wywołanie `f()` zostanie zarejestrowane przed odroczonym wywołaniem `g()`, to zostanie wykonane po nim. Dla przykładu:

```
(let f () (print "from f()"))
(let g () (print "from g()"))
(let h () {
    (defer f)
    (defer g)
    (print "from h()")
    0
})
```

Wywołując funkcję `h()` na standardowym wyjściu pojawi się następujący tekst:

```
from h()
from g()
from f()
```

Funkcja uruchomiona w wywołaniu odroczonym może zarejestrować swoje wywołania odroczone; takie wywołania mogą być zagnieżdżane bez ograniczeń.

Wywołania odroczone są uruchamiane po zakończeniu działania funkcji, w której zostały zarejestrowane, ale przed zniszczeniem wartości, które były umieszczone w jej zmiennych lokalnych. Ten fakt może zostać wykorzystany do implementacji zarządzania zasobami – jeśli wywołanie odroczone ma jakikolwiek uchwyt (np. wskaźnik) do zasobu to może wykonać dla niego jakąś logikę finalizacji. W takim celu po utworzeniu zasobu najlepiej od razu zarejestrować wywołanie odroczone, które dokona jego „finalizacji".

```
(let f () {
    ; ...

    (let x (Some_module.reserve_resource))
    (defer Some_module.free_resource (Std.pointer x))

    ; ...
})
```

5.3.9.1 Wywołania odroczone, a *tailcall*

Wywołania odroczone są związane z ramką wywołania, w której zostały zarejestrowane. Jeśli ta ramka jest zastępowana inną z powodu wywołania *tailcall* tu tuż **przed** zastąpieniem ramki uruchamiane są wszystkie zarejestrowane wywołania odroczone.

5.3.10 Wywołanie aktora

`(actor` *`name expression*`*`)`

5.3.10.1 Fork-join

Metoda *fork-join* może być zrealizowana w ViuAct za pomocą aktorów. Jeśli wynik wywołania aktora jest przypisywany do zmiennej, to aktor będzie działał jako *aktor potomny* aktora, który go wywołał. Taki aktor potomny może być włączony (*join*) za pomocą wywołania funkcji wbudowanej `Std.Actor.join()` z biblioteki standardowej (opisanej na stronie 56).

```
(let p (actor foo x))             ; aktor stworzony z funkcji foo
                                  ; wykonuje się równolegle
(let result (Std.Actor.join p)) ; wynik pracy aktora będzie dostępny
                                  ; w zmiennej result
```

Funkcja `Std.Actor.join()` zwraca wartość, którą zwróciła funkcja uruchomiona jako aktor lub wyjątek, który tego aktora zabił. Aktor pochodny nie utworzy zrzutu stosu – w razie awarii jego wyjątek jest zachowywyany i czeka na włączenie.

5.3.10.2 Wolne aktory

Wolne aktory to aktory, których adres nie był przypisany do zmiennej w momencie ich utworzenia. Są one niezwiązane w żaden sposób z aktorem, który je utworzył. W przeciwieństwie do aktorów potomnych aktorów wolnych nie można włączyć za pomocą funkcji `Std.Actor.join()`.

Aktory wolne w razie awarii tworzą zrzuty stosu. Zrzut stosu zawiera zapis wszystkich ramek wywołań na stosie, wraz z zawartością rejestrów lokalnych w każdej ramce.

Typowym sposobem na uzyskanie adresu aktora wolnego jest takie zaprojektowanie programu żeby aktor wolny jako jeden z parametrów przyjmował PID, na który powinien wysłać *swój* PID po wystartowaniu. Dla przykładu:

```
(actor Some_module.free_actor (Std.Actor.self))
(let p (Std.Actor.receive))
```

Funkcja `Std.Actor.self()` pozwala aktoromi uzyskać swój własny PID.

5.3.11 Konstrukcja warunkowa

`(if` *`condition-expression true-arm false-arm`*`)`

Konstrukcja warunkowa składa się z trzech wyrażeń:

`condition-expression` wyrażenie, które po obliczeniu warunkuje, które wyrażenie będzie obliczane w następnym kroku

`true-arm` wyrażenie będzie obliczone jeśli `condition-expression` ewaluuje do prawdy

`false-arm` wyrażenie będzie obliczone jeśli `condition-expression` ewaluuje do fałszu

```
(if (= x 42)
    (print "Odpowiedź na sens życia, Wszechświata, i w ogóle wszystkiego.")
    (print "Nie bardzo.")
)
```

5.3.12 Wywołanie „watchdog”

`(watchdog` *name expression**`)`

Watchdog jest ostatnią linią obrony danego aktora przed niekontrolowanym zamknięciem w wyniku błędu programu. Jeśli aktor nie obsłużył błędu (reprezentowanego przez wyjątek) i jego stos wywołań został całkowicie odwinięty, ale aktor ma ustawiony watchdog to VM „w miejscu” podmieni odwinięty stos na wywołanie funkcji ustawionej jako watchdog.

Oznacza to, że aktor zamiast zostać zabity zacznie wykonywać funkcję ustawioną jako watchdog – jednocześnie zachowując PID, co może być bardzo przydatne.

Watchdog jest normalną funkcją i może korzystać ze wszystkich dozwolonych konstrukcji językowych. Jedynym ograniczeniem jest to, że watchdog musi być napisany w języku ViuAct lub w języku assemblera Viua VM.

Argumenty funkcji watchdog

```
(let use_as_watchdog (error controller) {
    (let report (struct))
    (:= report.error error)
    (:= report.pid (Std.Actor.self))
    (Std.Actor.send controller error)
})

(let some_other_function (controller) {
    (watchdog use_as_watchdog controller)
})
```

Jako pierwszy argument watchdog zawsze otrzymuje `struct` zawierający informacje o błędzie, który spowodował awarię aktora. Jako pozostałe argumenty otrzymuje to co zostało przekazane w wywołaniu. Czyli „pierwszy” argument użyty w wywołaniu będzie tak naprawdę drugi z punktu widzenia funkcji użytej jako watchdog.

Błędy wewnątrz watchdog Jeśli podczas wykonywania funkcji ustawionej jako watchdog wystąpi kolejny nieobsłużony błąd aktor zostanie natychmiast zabity, a watchdog nie będzie uruchomiony przez VM „z powodu samego siebie”.

5.3.13 Obsługa wyjątków

`(try` *expression* `((catch` *exception-tag name expression*`)+))`

Obsługa wyjątków w ViuAct jest mechanizmem bardzo prostym. W momencie, w którym wyjątek jest zgłaszany stos wywołań jest odwijany w poszukiwaniu pierwszego bloku `catch` z odpowiadającym mu tagiem (*exception-tag*).

Kiedy taki blok jest znaleziony, wyłapana wartość wyjątku jest dowiązywana do nazwy (*name*) określonej w bloku. Wyjątek jest odrzucany (jeśli nie jest potrzebny) jeśli nazwa to „_”[3]

```
(let x (try (Std.Actor.receive 1s)) (
    (catch Bad_error _ {
        (print "Something bad happened.")
        0   ; return a default value
    })
    (catch Worse_error _ {
        (print "Something worse happened!")
        0   ; return a default value
```

[3]Podobne zachowanie, czyli odrzucenie wartości jeśli nazwa dowiązania to _, można odnaleźć w języku Erlang.

```
    })
    (catch The_worst_error _ {
        (print "Apocalypse.")
        0   ; return a default value
    })
))
```

5.3.14 Wyliczenia

`(enum` *enum-name* `(`*enum-member*`+))`

Wyliczenia przydatne są do reprezentowania pewnego skończonego zbioru możliwych wartości. Ułatwiają one zachowanie porządku w programie dzięki umożliwieniu wyliczenia możliwych stanów pewnych zmiennych, oraz redukują możliwość popełnienia błędu dzięki utrzymaniu jednej definicji tych stanów.

```
(enum Good_languages (
    Cpp
    OCaml
    ViuAct
))

(enum Bad_languages (
    Perl
    Bash
    JavaScript
))
```

Wyliczenia mogą pojawić się jedynie na poziomie modułu, w tym na poziomie modułu anonimowego niejawnie dodawanego do plików z kodem źródłowym kompilowanym do plików wykonywalnych. Wyliczeń nie można tworzyć wewnątrz funkcji.

Wyliczenia są symbolicznymi wartościami liczbowymi, dlatego można porównywać je za pomocą normalnego operatora porównania:

```
(let is_viuact (x) (= x Good_languages.ViuAct))
```

5.3.15 Komentarze

`;` *treść komentarza*

Komentarz rozpoczyna się od znaku `;` i kończy wraz z końcem linii. Nie istnieją komentarze „blokowe” takie jak `/* komentarz */` z C++ czy `(* komentarz *)` z OCaml.

5.4 Biblioteka standardowa

Biblioteka standardowa języka ViuAct składa się z modułów implementujących wybrane algorytmy (np. sortowanie) i funkcjonalności (np. konwersje między typami danych), oraz umożliwia interakcję ze środowiskiem zewnętrznym gdyż ViuAct nie posiada wbudowanych w język mechanizmów takiej interakcji.

Część funkcji udostępnianych przez bibliotekę standardową jest niemożliwa do zaimplementowania w języku ViuAct. Takie funkcje są napisane albo w języku assemblera Viua VM, albo w języku C++.

Nie wszystkie funkcje biblioteki standardowej są „normalnymi" funkcjami, możliwymi do uruchomienia jako aktor lub przekazania jako argument. Niektóre z nich kompilują się do pojedynczych instrukcji maszyny (są to tzw. funkcje „wbudowane" opisane w rozdziale 5.4.1 na stronie 56).

Dla takich funkcji programista musi napisać *wrappery* jeśli chce użyć ich jak w niektórych kontekstach. Jednak w typowym programie nie będzie to potrzebne, gdyż funkcje biblioteki standardowej zazwyczaj implementują atomowe zadania, których sens uruchamiania w osobnych aktorach jest ograniczony.

Moduł `Std.Io` jest opisany na stronie 58. Moduł `Std.Posix.Network` jest opisany na stronie 59. Moduł `Std.Random` jest opisany na stronie 61.

Konwencje

Konwencją wykorzystywaną w specyfikacji funkcji biblioteki standardowej jest następujący sposób zapisu: *`nazwa-funkcji`*`(`*`typ-parametru`*`*) -> `*`typ-zwracany`* Jeśli jest ich więcej niż jeden, typy parametrów są oddzielane od siebie przecinkiem, np. `funkcja(Ta, Tb) -> Tc`.

Jeśli funkcja przyjmuje dowolny typ oznaczany jest on jako `T`. Jeśli takich dowolnych typów jest więcej oznaczane są jako `Ta`, `Tb` itd.

Jeśli ma znaczenie nazwa parametru formalnego jego nazwa jest podawana przed jego typem i oddzielona od niego znakiem dwukropka: *`nazwa`* `: T`.

5.4.1 Funkcje wbudowane

Funkcje wbudowane są funkcjami, które kompilują się do pojedynczej instrukcji Viua VM. Są one dostępne zawsze, bez potrzeby importowania modułów, w których się znajdują.

5.4.1.1 `print()`

```
print(T) -> void
```

Drukuje pojedynczą wartość na ekran. Wydruk kończy znakiem nowej linii (`\n`).

5.4.1.2 `Std.Actor.join()`

```
Std.Actor.join(Pid) -> T
```

Oczekuje na zakończenie wykonywania przez aktora o podanym PID, po czym zwraca wynik jego pracy. Może powodować zgłoszenie wyjątku jeśli aktor o podanym PID zakończył wykonywanie z powodu błędu, lub jeśli aktor o podanym PID nie istnieje.

5.4.1.3 `Std.Actor.receive()`

```
Std.Actor.receive() -> T
Std.Actor.receive(timeout-literal) -> T
```

Zwraca pierwszą dostępną wartość w kolejce wiadomości procesu wewnątrz którego ta funkcja została wywołana. Jeśli żadna wiadomość nie jest dostępna funkcja oczekuje w nieskończoność na nadejście wiadomości.

W wariancie drugim, oczekuje na nadejście wiadomości tylko przez okres czasu określony przez *timeout-literal*. Taki literał jest w postaci: *`n`*`s` (oczekuje *n* sekund), lub *`n`*`ms` (oczekuje *n* milisekund).

```
(let wait_1_second        (Std.Actor.receive 1s))
(let wait_16_milliseconds (Std.Actor.receive 16ms))
(let wait_forever         (Std.Actor.receive))
```

Parametr *`timeout-literal`* jest liczbą całkowitą podającą ile sekund (suffix `s`) lub milisekund (suffix `ms`) aktor powinien maksymalnie oczekiwać na wiadomość. Z uwagi ma nieprzewidywalność zarządzania czasem procesora realny czas oczekiwania może się wydłużyć, jednak nigdy nie będzie krótszy niż podany w parametrze.

5.4.1.4 `Std.Actor.self()`

```
Std.Actor.self() -> Pid
```

Zwraca PID procesu, w którym ta funkcja została wywołana.

5.4.1.5 `Std.Actor.send()`

```
Std.Actor.send(Pid, T) -> void
```

Wysyła do procesu określonego podanego przez PID kopię wartości przekazanej w drugim parametrze.

5.4.1.6 `Std.Pid.eq()`

```
Std.Pid.eq(Pid, Pid) -> Bool
```

Sprawdza czy dwie wartości PID są sobie równe.

5.4.1.7 `Std.copy()`

```
Std.copy(Ta) -> Ta
```

Kopiuje wartość podaną jako parametr funkcji.

5.4.1.8 `Std.String.at()`

```
Std.String.at(String, Integer) -> String
```

Pobiera znak spod indeksu w stringu.

5.4.1.9 `Std.String.concat()`

```
Std.String.concat(String, String) -> String
```

Pobiera dwa stringi i zwraca nowy string będący ich połączeniem.

5.4.1.10 `Std.String.eq()`

```
Std.String.eq(String, String) -> Bool
```

Sprawdza czy dwie wartości typu string są sobie równe.

5.4.1.11 `Std.String.size()`

```
Std.String.size(String) -> Integer
```

Zwraca długość stringu przekazanego jako parametr.

5.4.1.12 `Std.String.substr()`

```
Std.String.substr(String, a : Integer, b : Integer) -> String
```

Zwraca wycinek stringu pomiędzy `a` i `b`.

5.4.1.13 `Std.String.to_string()`

```
Std.String.to_string(T) -> String
```

Konwertuje dowolną wartość na string.

5.4.1.14 `Std.Integer.of_bytes()`

```
Std.Integer.of_bytes(Byte_string) -> Integer
```

Konwertuje string bajtów na liczbę całkowitą. Stringi bajtów są zwracane m.in. z funkcji `Std.Io`.

5.4.1.15 `Std.Vector.at()`

```
Std.Vector.at(v : Vector, i : Integer) -> T
```

Zwraca wskaźnik do wartości w wektorze `v` pod indeksem `i`. Jeśli wektor jest krótszy niż `i` to zgłaszany jest wyjątek.

5.4.1.16 `Std.Vector.push()`

```
Std.Vector.push(Vector, T) -> void
```

Dopisuje wartość do wektora.

5.4.1.17 `Std.Vector.size()`

```
Std.Vector.size(Vector) -> Integer
```

Zwraca długość wektora.

5.4.1.18 `Std.Pointer.take()`

```
Std.Pointer.take(T) -> Tp
```

Zwraca wskaźnik do wartości podanej w parametrze.

5.4.2 `Std.Io`

Moduł udostępniający mechanizmy I/O (*wejścia-wyjścia*). Umożliwia on wykonywanie operacji I/O na plikach, oraz prostą interakcję z konsolą użytkownika (`tty`). I/O jest buforowane. Ten moduł nie jest bezpośrednim opakowaniem dla wywołań systemowych I/O definiowanych przez standard POSIX (np. `read(3)` lub `write(3)`).

Funkcje z tego modułu wykorzystują typ `Fstream` (*file stream*) umożliwiający dostęp do pliku.

5.4.2.1 `Std.Io.stdin_getline()`

```
Std.Io.stdin_getline() -> String
```

Umożliwia odczytanie pojedynczej linii ze strumienia standardowego wejścia (***`stdin`***).

5.4.2.2 `Std.Io.open()`

```
Std.Io.open(String) -> Fstream
```

Otwiera plik zdefiniowany ścieżką podaną w parametrze.

5.4.2.3 `Std.Io.peek()`

```
Std.Io.peek(Fstream) -> String
```

Zwraca pierwszy znak w strumieniu.

5.4.2.4 `Std.Io.getline()`

```
Std.Io.getline(Fstream) -> String
Std.Io.getline(Fstream, String) -> String
```

Wariant 1. odczytuje znaki ze strumienia do napotkania bajtu białej linii (`\n`). Wariant 2. odczytuje znaki ze strumienia do napotkania bajtu podanego w parametrze.

5.4.2.5 `Std.Io.read()`

```
Std.Io.read(Fstream) -> String
Std.Io.read(Fstream, Integer) -> String
```

Wariant 1. odczytuje naraz całą zawartość strumienia (cały plik). Wariant 2. odczytuje maksymalnie n bajtów.

5.4.2.6 `Std.Io.write()`

```
Std.Io.write(Fstream, String) -> void
```

Zapisuje string do strumienia.

5.4.3 `Std.Posix.Network`

Moduł udostępniający implementację „POSIX sockets". Jest to cienka abstrakcja nad API dostarczanym przez system operacyjny; w przypadku braków w tym dokumencie ich dokumentację można wydedukować ze stron manuala sekcji 3 dostarczanych przez program `man(1)` (np. dokumentację dla funkcji `Std.Posix.Network.socket` można uzyskać wykonując polecenie `man 3 socket`).

5.4.3.1 `Std.Posix.Network.socket()`

```
Std.Posix.Network.socket() -> Integer
```

Tworzy socket za pomocą wywołania funkcji `socket(3)`. Socket jest w rodzinie `AF_INET`, rodzaju `SOCK_STREAM`, i jest tworzony bez flag.

Socket zwracany przez tą funkcję jest blokujący.

5.4.3.2 `Std.Posix.Network.connect()`

```
Std.Posix.Network.connect(
      socket : Integer
    , addr   : String
    , port   : Integer
) -> void
```

Wrapper na funkcję `connect(3)`.

5.4.3.3 `Std.Posix.Network.bind()`

```
Std.Posix.Network.bind(
      socket : Integer
    , addr   : String
    , port   : Integer
) -> void
```

Wrapper na funkcję `bind(3)`.

5.4.3.4 Std.Posix.Network.listen()

```
Std.Posix.Network.listen(
      socket  : Integer
    , backlog : Integer
) -> void
```

Wrapper na funkcję `listen(3)`.

5.4.3.5 Std.Posix.Network.accept()

```
Std.Posix.Network.accept(socket : Integer) -> void
```

Wrapper na funkcję `accept(3)`.
Sockety zwracane przez tą funkcję są nieblokujące. Ich timeout jest ustawiony na 500ms.

5.4.3.6 Std.Posix.Network.write()

```
Std.Posix.Network.write(
      socket : Integer
    , value  : T
) -> Integer
```

Niezależnie od tego jaka wartość jest przekazana do funkcji, będzie najpierw przekonwertowana na `String`, a potem wpisana do socketu.

5.4.3.7 Std.Posix.Network.read()

```
Std.Posix.Network.read(
    socket  : Integer
) -> String
```

Wrapper na funkcję `read(3)`. Odczytuje z socketu maksymalnie 1024 bajty.

5.4.3.8 Std.Posix.Network.recv()

```
Std.Posix.Network.recv(
      socket        : Integer
    , buffer_length : Integer
) -> String
```

Wrapper na funkcję `recv(3)`. Odczytuje z socketu maksymalnie `buffer_length` bajtów.

5.4.3.9 Std.Posix.Network.shutdown()

```
Std.Posix.Network.listen(
    socket  : Integer
) -> void
```

Wrapper na funkcję `shutdown(3)`. Wyłącza socket z flagą `SHUT_RDWR`.

5.4.3.10 Std.Posix.Network.close()

```
Std.Posix.Network.listen(
      socket  : Integer
) -> void
```

Wrapper na funkcję `close(3)`.

5.4.4 `Std.Random`

Moduł udostępniający dostęp do liczb losowych ogólnego przeznaczenia (`/dev/urandom`) oraz zdatnych do zastosowań kryptograficznych (`/dev/random`).

5.4.4.1 `Std.Random.random()`

```
Std.Random.random() -> Float
```

Zwraca losową liczbę zmiennoprzecinkową z przedziału $[0.0, 1.0)$.

5.4.4.2 `Std.Random.randint()`

```
Std.Random.randint(lower : Integer, upper : Integer) -> Integer
```

Zwraca losową liczbę całkowitą z przedziału [*lower*, *upper*).

5.5 Interakcja z platformą

„Platformą” w przypadku ViuAct jest Viua VM. Narzuca to pewne ograniczenia związane ze środowiskiem, w którym programy napisane w języku ViuAct można skompilować i uruchomić. Wynika to z faktu, że Viua VM sama w sobie ma pewną platformę i wymagania.

5.5.1 Platforma kompilacji

Do kompilacji programów napisanych w ViuAct potrzebny jest interpreter języka Python 3 w wersji co najmniej 3.6. Jest to jedyne ograniczenie, ponieważ na etapie kompilacji ViuAct nie odwołuje się do żadnego elementu Viua VM.

5.5.2 Platforma asemblacji i linkowania

Asemblacja i linkowanie odbywa się z użyciem narzędzi dostarczanych przez Viua VM. Oprócz interpretera języka Python 3 potrzebny jest assembler dostarczany przez Viua VM. Oznacza to, że etap asemblacji i linkowania może odbywać się jedynie na platformie zgodnej z POSIX-2008, wyposażonej w kompilator obsługujący standard C++17 (są to wymagania Viua VM).

5.5.3 Platforma uruchomieniowa

Uruchomienie odbywa się w całości przy użyciu narzędzi dostarczanych przez Viua VM. Ograniczenia i zależności są na tym etapie całkowicie zależne od wymagań narzucanych przez Viua VM.

Rozdział 6

Język ViuAct i jego kompilator – implementacja

Pierwszą częścią naszej pracy jest zaprojektowanie wysokopoziomowego języka programowania i opracowanie jego implementacji. Z uwagi na to, że platforma uruchomieniowa, którą wykorzystujemy (czyli Viua VM) jest maszyną wirtualną wykonującą programy w postaci *bytecode* wybranym sposobem implementacji języka jest kompilator - program tłumaczący kod źródłowy w jednym języku na kod źródłowy w innym języku przy jednoczesnym zachowaniu zachowania programu. W przypadku naszej pracy językiem źródłowym jest język ViuAct (dokładniej opisany w rozdziale 5. Specyfikacja języka ViuAct na stronie 43), a językiem docelowym język assemblera Viua VM.

Materiałem teoretycznym przy pracy nad kompilatorem i językiem była książka Michaela Lee Scotta „Programming language pragmatics" ([4]). Jedną z książek, które wywarły wpływ na zestaw instrukcji (ISA) Viua VM była książka Davida Pattersona i Andrew Watermana „The RISC-V reader" ([5]). Pomocą w pracach nad systemem modułów była książka Johna Levine'a „Linkers and loaders" ([6]).

Sama maszyna wirtualna nie tylko implementuje określone ISA, ale też dostarcza *runtime* realizujący zadania typowo związane z systemem operacyjnym takie jak zarządzanie czasem procesora, przydzielanie numerów PID, czy nadzór na komunikacją między procesami – w tym rejonie pomocą była książka Andrew Tannenbauma „Systemy operacyjne" ([7]).

Aktory w ViuAct (tak samo jak implementujące je procesy w Viua VM) komunikują się bezpośrednio. Dla porównania, język Go (opisany w [8]), który podobnie jak ViuAct kładzie duży nacisk na współbieżność i równoległość do komunikacji między *goroutines* (jego odmiana procesów) wykorzystuje kanały – używając „pośredników" zamiast skomunikować procesy bezpośrednio.

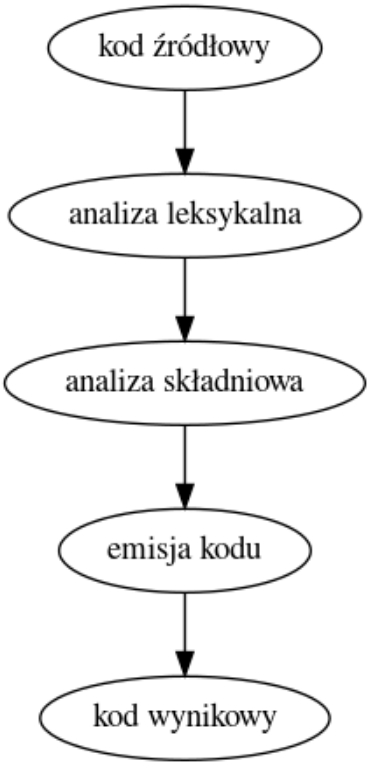
6.1 Architektura

6.1.1 Użyte wzorce projektowe – Sposób konstrukcji kompilatora

Na rysunku 6.1 przedstawiony jest uproszczony schemat budowy kompilatora. W kompilatorach „produkcyjnych" (np. GCC, Clang, czy ICC) tych faz jest więcej – przede wszystkim etap emisji kodu jest dużo bardziej rozbudowany, oraz dochodzą etapy analizy semantycznej (czy program ma sens) czy optymalizacji (prób takiego przekształcenia kodu programu żeby przy zachowaniu znaczenia działał wydajniej).

Kompilator języka ViuAct dostarczany jako element tej pracy inżynierskiej jest pozbawiony etapów analizy semantycznej oraz optymalizacji. Analiza semantyczna (oraz weryfikacja typów i wykrywanie błędów na etapie kompilacji) jest oddelegowana do assemblera dostarczanego przez platformę. Optymalizacja jest całkowicie pominięta gdyż jest to temat niezwykle rozległy; implementacja i doszlifowanie algorytmów optymalizujących kod jest sama w sobie materiałem wystarczającym na napisanie osobnej pracy inżynierskiej.

Architektura kompilatora języka ViuAct jest dokładniej opisana w rozdziale 6.1.3 (Dekompozycja systemu na podsystemy) na stronie 64. Sposób działania kompilatora jest opisany w rozdziale 6.1.4 (Przebieg procesu kompilacji) na stronie 67. Omówienie interakcji kompilatora języka ViuAct z narzę-



Rysunek 6.1: Podstawowy schemat budowy kompilatora

dziami dostarczanymi przez platformę Viua VM znajduje się w rozdziale 6.1.2 (Architektura systemu) na stronie 64.

Oprócz kompilatora (rozdział 6.1.3.1 na stronie 66) dostarczany jest również „program łączący" (rozdział 6.1.3.2 na stronie 67) dokonujący automatycznego połączenia wymaganych modułów w gotowy plik wykonywalny.

6.1.2 Architektura systemu

Rysunek 6.2 („Interakcje: od pliku źródłowego do działającego programu") prezentuje schemat interakcji jakie zachodzą w całym systemie od momentu wczytania pliku źródłowego przez kompilator do uruchomienia programu przez jądro Viua VM.

Ostatnią fazą jaką zajmuje się kompilator języka ViuAct dostarczany jako element tej pracy inżynierskiej jest emisja kodu („Assembly code emission"), której wynikiem jest plik z kodem źródłowym w języku assemblera Viua VM („`hello_world.asm`" na rysunku 6.2). Rozdział 6.1.3 (Dekompozycja systemu na podsystemy) dokładniej opisuje działanie samego kompilatora.

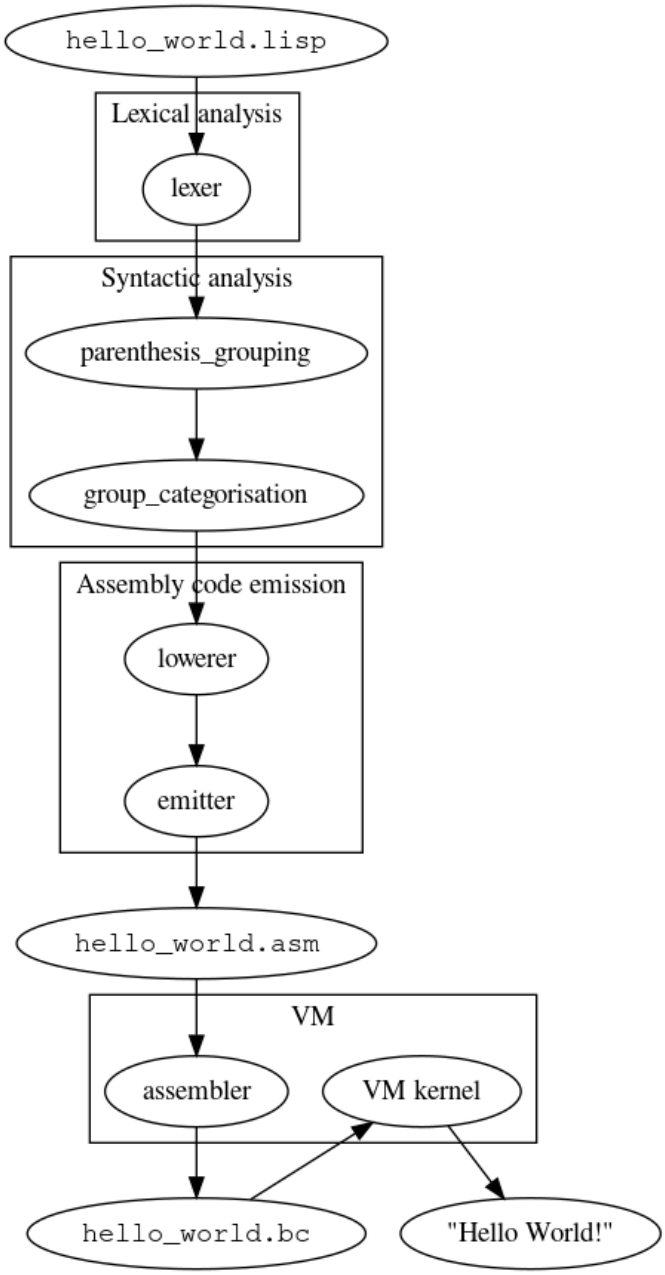
Zakres pracy inżynierskiej obejmuje wygenerowanie pliku zawierającego poprawny kod w języku assemblera Viua VM oraz plików pomocniczych (zadanie kompilatora), oraz takim pokierowaniu narzędziami dostarczanymi przez platformę, żeby wyemitowały one plik wykonywalny bądź bibliotekę (zadanie „programu łączącego"). Zakładamy, że narzędzia dostarczane przez platformę działają poprawnie.

Pliki pomocnicze są wymagane przez „program łączący" (opisany w rozdziale 6.1.3.2 na stronie 67). Ich dokładniejsze opisy znajdują się w rozdziałach „Pliki interfejsów modułów (`.i`)" na stronie 72 i „Pliki zależności modułów (`.d`)" na stronie 74

6.1.3 Dekompozycja systemu na podsystemy

Język ViuAct jest implementowany przez dwa programy:

1. **kompilator** - który przetwarza kod źródłowy w języku ViuAct na kod źródłowy w języku assemblera Viua VM



Rysunek 6.2: Interakcje: od pliku źródłowego do działającego programu

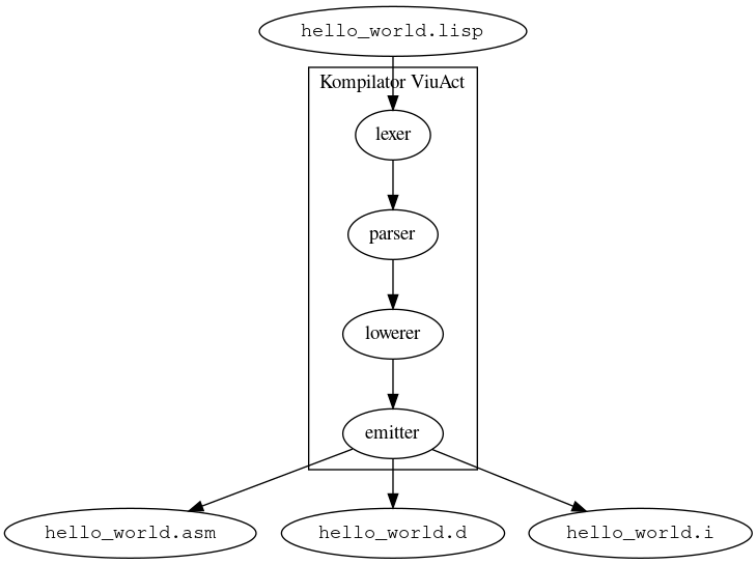
2. **linker** - który na podstawie wyników pracy kompilatora tworzy pliki wykonywalne, które mogą być uruchomione na jądrze Viua VM

Większość pracy w tym układzie wykonuje kompilator, opisany w rozdziale 6.1.3.1 na stronie 66. Generuje on pliki zawierające kod źródłowy w języku assemblera gotowe do przetworzenia przez assembler Viua VM na formę binarną, oraz pliki pomocnicze.

Z plików pomocniczych korzysta zarówno sam kompilator (do określenia interfejsów modułów importowanych przez aktualnie kompilowany moduł), ale też linker – do określenia jakie moduły powinny być dołączone do aktualnie emitowanego pliku wykonywalnego, aby zapewnić dostępność wszystkich wymaganych funkcji. Linker jest dokładniej opisany w rozdziale 6.1.3.2 na stronie 67.

6.1.3.1 Kompilator – `viuact-cc`

Kompilator składa się z kilku podsystemów, zgodnie z przedstawieniem na rysunku 6.3.



Rysunek 6.3: Podział kompilatora na podsystemy

Każdy podsystem implementuje jedną z faz kompilacji:

1. **lexer** dokonuje analizy leksykalnej wczytanego pliku źródłowego, dzieląc go na listę tokenów

2. **parser** dokonuje analizy składniowej łącząc tokeny w grupy reprezentujące większe konstrukcje językowe

3. **lowerer** mapuje grupy wyprodukowane przez *parser* do odpowiednich funkcji *emittera*; jest to dość banalny etap, ale upraszcza budowę kompilatora

4. **emitter** tłumaczy konstrukcje językowe ViuAct na równoznaczne konstrukcje w języku assemblera Viua VM

Różnica między podsystemami *lowerer* i *emitter* może być niejasna. Oba biorą udział w emisji kodu wynikowego, ale *lowerer* bezpośrednio zajmuje się tylko modułami i funkcjami, natomiast *emitter* implementuje emisję pojedynczych wyrażeń języka ViuAct – przypisań `let`, konstrukcji warunkowych `if`, wywołań funkcji, itd.

Proces kompilacji dokłaniej opisany jest w rozdziale 6.1.4 (Przebieg procesu kompilacji) na stronie 67.

6.1.3.2 Program łączący – `viuact-opt`

Program łączący (tzw. „*linker*") zajmuje się finalną fazą „kompilacji". Jest to stwierdzenie jednocześnie trafne i niepoprawne. Zazwyczaj jednak nie ma to znaczenia, ponieważ zarówno linker jak i kompilator jest ukrywany przed programistą. Popularne „kompilatory" jak np *g++* z GCC to tak naprawdę „drivery"; wywołanie polecenia *g++* powoduje wywołanie zarówno kompilatora (*cc1*), assemblera (*as*), jak i linkera (*ld*) w taki sposób aby na wyjściu uzyskać oczekiwany wynik, czyli na przykład plik wykonywalny w formacie ELF *a.out*.

W przypadku kompilatora ViuAct proces ten wygląda podobnie, ale nie jest aż tak zautomatyzowany. Rolę „drivera" pełni programista, który jest odpowiedzialny za wywołanie zarówno kompilatora jak i linkera. Przykładowo:

Listing 6.1: „Hello World!" kompilacji

```
viuact-cc --mode exec ./hello_world.lisp                    1
viuact-opt ./build/_default/hello_world.asm                 2
```

Program łączący przeprowadzi proces asemblacji pliku podanego na wejściu, oraz dołączy do niego wszelkie wymagane moduły. Zarówno asemblacja jak i łączenie będa przeprowadzone przez narzędzie dostarczane przez platformę Viua VM – `viuact-opt` zajmuje się jedynie wygenerowaniem odpowiednich poleceń dla tego narzędzia.

Informacja o tym jakie moduły muszą zostać dołączone jest tworzona w oparciu o pliki zależności (opisane w rozdziale Pliki zależności modułów (`.d`) na stronie 74). Dla uproszczenia projektu program łączący nie zbiera informacji o zależnościach rekurencyjnie.

Po zebraniu informacji o zależnościach program łączący dokonuje asemblacji wszystkich modułów, od których zależy kompilowany moduł główny. Następnie asembluje moduł główny i łączy wszystkie wyemitowane modułu bytecode'u w gotowy plik wykonywalny.

6.1.4 Przebieg procesu kompilacji

Ten rozdział zawiera dokładny opis procesu kompilacji, od momentu wczytania pliku z kodem źródłowym w języku ViuAct do momentu wyemitowania kodu wynikowego w języku assemblera Viua VM. Kompilator jest wywoływany poleceniem `viuact-cc` z opcją `--mode` określającą czy kompilowany jest moduł wykonywalny (`exec`) czy moduł biblioteki (`module`):

```
viuact-cc --mode ( 'exec' | 'module' ) file.lisp            1
```

Ten rozdział opisuje proces kompilacji „omawiając" go, bez odnoszenia się do konkretnych elementów implementacji kompilatora. Opis implementacji kompilatora znajduje się w rozdziale 6.6 na stronie 74.

6.1.4.1 Wczytanie pliku źródłowego

Kompilator wczytuje do pamięci cały plik źródłowy jako pojedynczy string.

6.1.4.2 Analiza leksykalna

Lexer patrzy na wczytany kod źródłowy i do pierwszego nieprzeanalizowanego znaku (czyli na początku analizy do znaku na indeksie 0) próbuje przypasować wzorzec określający jaki token znajduje się na tej pozycji. Po udanym przypasowaniu pozycja, która będzie rozpatrywana przez lexer jest przesuwana o tyle znaków ile wynosi długość wygenerowanego tokenu i lexer rozpoczyna pracę od nowa. Ten proces trwa do momentu aż cały string wejściowy nie zostanie przeanalizowany, albo odrzucony jako nieprawidłowy.

Algorytm przypasowania jest banalny. Lexer dysponuje listą wzorców (określonych przez wyrażenia regularne), które określają jak wygląda każdy możliwy token w języku. Lexer po kolei próbuje przypasować każdy wzorzec z listy i kończy na pierwszym trafieniu. Jeśli żaden wzorzec nie może zostać przypisany lexer odrzuca kod źródłowy jako nieprawidłowy.

Wzorce są uszeregowane w taki sposób żeby nie była możliwa pomyłka i na przykład przypasowanie początku nazwy zmiennej `letter` jako słowa kluczowego `let`.

6.1.4.3 Analiza składniowa

Składnia języka została zaprojektowana w taki sposób aby analiza składniowa mogła być uproszczona do maksimum i prosta w implementaji.

6.1.4.3.1 Grupowanie nawiasów W pierwszej fazie analizy składniowej tokeny grupowane są według nawiasów okrągłych, przy czym grupowanie jest rekurencyjne (jeśli jakaś grupa zawiera podgrupę w nawiasach to zagnieżdżona grupa będzie widoczna jako pojedynczy element w grupie zewnętrznej). Dla przykładu:

```
(let x (frobnicate 42))
```

będzie zgrupowane w następujący sposób:

```
[ "let"; "x"; [ "frobnicate"; "42" ] ]
```

6.1.4.3.2 Grupowanie id Kolejnym etapem jest grupowanie id. Id jest nazwą składającą się z kilku członów, na przykład `Std.Posix.Network.socket` składa się z 7 tokenów: trzech *nazw modułów* (`Str`, `Posix`, i `Network`), trzech *operatorów dostępu* (kropek), i jednej *nazwy* (`socket`). Taka grupa zostanie na tym etapie zredukowana do pojedynczego elementu.

6.1.4.3.3 Oznczanie wyrażeń złożonych Wyrażenia złożone składają się z kilku wyrażeń (prostych bądź złożonych). Z uwagi na fakt, że formą pośrednią wykorzystywaną na etapie grupowania są listy tokenów takie wyrażenie byłoby nieodróżnialne od listy reprezentującej wywołanie funkcji. Dlatego na etapie analizy składniowej do list reprezentujących wyrażenia złożone dodawany jest specjalny token-fantom. Dzięki temu zostaje zachowana właściwość umożliwiająca szybkie klasyfikowanie grup. Dla przykładu:

```
(let x { ... })
```

będzie zgrupowane w następujący sposób:

```
[ "let"; "x"; [ Compound_expression_marker; ... ] ]
```

6.1.4.3.4 Klasyfikacja grup Ostatnim etapem analizy składniowej jest klasyfikacja grup. W większości przypadków do klasyfikacji listy tokenów do grupy reprezentującej konkretną konstrukcję językową wystarczy spojrzeć na pierwszy token na liście. W niektórych przypadkach algorytm musi się posiłkować długością listy.

Dla przykładu:

```
[ "let"; "x";           ... ]        -> let-binding
[ "let"; "x"; [ ... ]; ... ]        -> function-definition
[ ... ]                             -> function-call
[ "actor"; ... ]                    -> actor-call
[ Compound_expression_marker; ... ] -> compound-call
```

Różnicą między definicją zmiennej (`let-binding`), a definicją funkcji (`function-definition`) jest długość listy - definicja zmiennej zawiera trzy elementy (słowo kluczowe `let`, nazwę, i wyrażenie), a definicja funkcji cztery elementy (słowo kluczowe `let`, nazwę, listę parametrów formalnych, i wyrażenie).

6.1.4.4 Emisja modułów

W następnej kolejności emitowane są wszystkie moduły zagnieżdżone w aktualnie kompilowanym module, przy czym ten etap postępuje rekurencyjnie. Moduły zagnieżdżone muszą być wyemitowane przed modułem głównym, aby kompilator miał dostęp do ich plików interfejsów i umożliwić ich importowanie.

6.1.4.5 Analiza importu modułów

Kolejnym krokiem jest analiza modułów importowanych przez aktualnie kompilowany moduł i wczytanie ich interfejsów. Kompilator ładuje listy sygnatur funkcji i wyliczenia z każdego zaimportowanego modułu.

Jeśli kompilator nie może znaleźć pliku interfejsu danego modułu to kończy kompilację informując o błędzie. Kompilator szuka plików interfejsów i modułów w ścieżkach podanych w zmiennej środowiskowej `VIUAC_LIBRARY_PATH` (opisanej na stronie 82).

6.1.4.6 Emisja kodu wynikowego

Na końcu następuje emisja kodu wynikowego w języku assemblera Viua VM. Ten etap jest wykonywany osobno dla każdej funkcji zdefiniowanej w kompilowanym module.

6.1.4.6.1 Redukcja poziomu wyrażeń Najpierw następuje redukcja poziomu wyrażeń. Tym etapem zajmuje się *lowerer*. Jest to mechaniczny proces mapujący sklasyfikowane grupy reprezentujące konretne konstrukcje językowe do funkcji udostępnianych przez *emitter*, opakowanie wyniku w sposób jakiego wymagają zasady języka assemblera Viua VM, oraz serializacja wyników do stringów.

6.1.4.6.2 Emisja instrukcji języka assemblera Emisja instrukcji języka assemblera jest wykonywana per-wyrażenie. Ten etap przeplata się z redukcją poziomu wyrażeń i jest implementowany przez *emitter*. *Emitter* emituje sekwencje instrukcji języka assemblera Viua VM odpowiadające zadanym konstrukcjom językowym ViuAct.

Dla przykładu `(let x 42)` zostanie wyemitowane jako pojedyncza instrukcja: `integer %x local 42`. Natomiast `(Some_module.frobnicate 42)` zostanie wyemitowane jako sekwencja instrukcji:

```
integer %3 local 42
frame %1 arguments
copy %0 arguments %3 local
call void Some_module::frobnicate/1
```

6.1.4.7 Zapis pliku `.asm`

Dla każdego wyemitowanego modułu kompilator zapisze plik *`nazwa_modulu`*`.asm` zawierający kod wynikowy w języku assemblera Viua VM.

6.1.4.8 Zapis pliku `.i`

Dla każdego wyemitowanego modułu biblioteki kompilator zapisze plik *`nazwa_modulu`*`.i` zawierający definicję interfejsu tego modułu. Pliki interfejsów są opisane w rozdziale Pliki interfejsów modułów (`.i`) na stronie 72.

6.1.4.9 Zapis pliku `.d`

Dla każdego wyemitowanego modułu kompilator zapisze plik *`nazwa_modulu`*`.d` zawierający definicję zależności tego modułu. Pliki zależności są opisane w rozdziale Pliki zależności modułów (`.d`) na stronie 74.

6.2 Projekt struktury

6.2.1 Wykorzystywane struktury danych

Brak jest rozbudowanego diagramu klas, ponieważ program nie jest pisany w stylu obiektowym. W programie istnieją dwie główne grupy struktur opisujące elementy języka – typy tokenów i typy grup (reprezentujących konstrukcje językowe), struktura reprezentująca adres rejestru (abstrakcyjny „slot" na wartości), struktura reprezentująca stan kompilowanego programu, oraz cztery struktury reprezentujące niskopoziomowe abstrakcje linii programu w języku assemblera – konstruktor, przeniesienie, wywołanie, i *verbatim*.

Konwencje

Listingi przedstawiające wykorzystywane struktury danych są zapisane w języku OCaml. Kompilator jest napisany w języku Python, który nie pozwala na tak łatwe i czytelne definiowanie nowych struktur danych – stąd decyzja o użyciu innego języka w pracy.

Zachowane zostały typy danych, nazwy pól i całych struktur. OCaml pozwala na dużo bardziej przejrzyste i czytelne opisanie typów danych poszczególnych pól niż Python, co ma dużą wartość dokumentacyjną.

6.2.1.1 Tokeny

Każdy typ tokenu jest reprezentowany przez osobną strukturę. Tokeny, oprócz swojego typu mają atrybuty określające ich lokalizację w pliku (wiersz i kolumna), oraz pole z leksemem. Typy tokenów wymagane do reprezentacji języka ViuAct są opisane w specyfikacji języka.

Definicje struktur reprezentujących tokeny są zawarte w pliku `viuact/token_types.py`.

6.2.1.2 Grupy

Każda konstrukcja językowa jest reprezentowana przez osobną strukturę. Konstrukcje wymagane do reprezentacji języka ViuAct są opisane w specyfikacji języka.

Definicje struktur reprezentujących grupy są zawarte w pliku `viuact/group_types.py`.

6.2.1.3 Slot

```
type register_set =
    | Local
    | Parameters
    | Arguments
    | Closure_local

type slot = {
    name         : string ;
    index        : int ;
    register_set : register_set ;
}
```

Struktura `slot` reprezentuje adres rejestru. Z punktu widzenia języka ViuAct istotne jest pole `name` (określające nazwę slotu jaką posługuje się programista); z punktu widzenia emitera kodu istotne są pola `index` i `register_set` określające adres rejestru jakim posługuje się Viua VM.

6.2.1.4 Stan programu

Stan programu (struktura `State`) zawiera pola śledzące ilość zaalokowanych rejestrów, widoczne funkcje, a w przypadkach śledzenia stanu funkcji zagnieżdżonej – także wykorzystywane sloty z otaczającego zakresu leksykalnego.

6.2.1.5 Konstruktor

```
type ctor = {
    of_type : string ;
    slot    : slot ;
    value   : string ;
}
```

Konstruktor reprezentuje instrukcję bezpośrednio tworzącą wartość w rejestrze. Przykładowo, aby wyemitować instrukcję `integer %1 local 42` utworzony zostanie:

```
{
    of_type = "integer" ;
    slot = { name = "x"; index = 1 ; register_set = Local } ;
    value = "42"
}
```

6.2.1.6 Przeniesienie

```
type move_kind =
    | Move
    | Copy

type move = {
    kind   : move_kind ;
    source : slot ;
    dest   : slot option ;
}
```

Przeniesienie opisuje przesunięcie (instrukcja *`move`*) lub kopię wartości (instrukcja *`copy`*). Slot docelowy nie musi być obecny - np. wtedy wartość jest przenoszona do slotu `void`.

6.2.1.7 Wywołanie

```
type call_kind =
    | Synchronous
    | Actor
    | Tail
    | Deferred

type call = {
    kind : call_kind ;
    slot : slot option ;
    to   : string ;
}
```

Wywołanie opisuje wywołanie funkcji w każdy sposób dostępny w języku ViuAct: zwykłe wywołanie funkcji, wywołanie tworzące aktora, wywołanie *tail call*, i wywołanie „odroczone”.

Typy wywołań opisane są w specyfikacji języka; `Synchronous` w 5.3.7 (strona 51), `Actor` w 5.3.10 (strona 53), `Tail` w 5.3.8 (strona 51), `Deferred` w 5.3.9 (strona 52).

6.2.1.8 Linia *verbatim*

```
type verbatim = {
    text : string ;
}
```

Linia *verbatim* opisuje dowolną linię języka assemblera (m.in. dyrektywy `.import:` czy `.function:`).

Emitter (rozdział 6.1.4.6 na stronie 69) większość instrukcji tworzy za pomocą linii *verbatim*. Jest to zabieg o tyle „brzydki" co efektywny; na etapie prototypowania bardzo szybko można w ten sposób wyemitować spory zakres instrukcji bez potrzeby projektowania struktury dla każdego typu instrukcji.

6.3 Decyzje projektowe

6.3.1 Środowisko docelowe

Środowiskiem docelowym, na którym będą uruchamiane programy napisane w języku ViuAct jest maszyna wirtualna Viua VM. Środowisko docelowe musi spełniać wymagania jakie ma Viua VM (m.in. musi to być system zgodny ze standardem POSIX).

6.3.2 Środowisko implementacji

Środowiskiem implmentacji jest Linux z dostępnymi standardowymi narzędziami GNU, interpreterem języka Python 3, i umożliwiający uruchomienie assemblera dostarczanego przez Viua VM.

6.3.3 Priorytety implementacyjne

Maksymalizacja prostoty budowy kompilatora i języka. Marginalizacja obsługi błędów w kompilatorze z uwagi na brak czasu. Marginalizacja optymalizacji z uwagi na brak czasu.

6.4 Projekt algorytmów i przyjętych protokołów

Dyskusja na temat algorytmów i sposobu implementacji jest częściowo przeprowadzona w rozdziale 6.1.4 na stronie 67, szczególnie w rozdziale 6.1.4.3.

6.5 Projekt interfejsu

6.5.1 Interfejs kompilatora

Kompilator składa się z dwóch programów: `viuact-cc` (kompilatora właściwego) i `viuact-opt` (programu łączącego). Programy te są konfigurowane za pomocą zmiennych środowiskowych, które kontrolują poziom „głośności" logów, położenie assemblera Viua VM, a także włączają bądź wyłączają serializację formy pośredniej.

Interfejs użytkownika opisany jest w rozdziale 6.8 „Instrukcja użytkownika kompilatora języka ViuAct" na stronie 81.

6.5.2 Interfejs języka

Interfejsem języka jest jego składnia. Jest ona opisana w rozdziale 5 na stronie 43.

6.5.3 Inne interfejsy

6.5.3.1 Pliki interfejsów modułów (`.i`)

Pliki interfejsów modułów wyliczają funkcje eksportowane przez dany moduł, oraz prezentują metadane wymagane do połączenia plików w sposób, który będzie mógł działać na Viua VM. Pliki interfejsu dla modułów „własnych" nie różnią się zasadniczą strukturą od plików interfejsu dla modułów „obcych", ale pliki interfejsu dla modułów „obcych" muszą być uzupełnione o kilka dodatkowych pól. Jest to dokładniej opisane w rozdziale 6.5.3.1.2 na stronie 73.

Pliki interfejsów są zapisywane w formacie JSON.

6.5.3.1.1 Plik interfejsu dla modułów „własnych” Moduły „własne” języka ViuAct to moduły napisane w języku ViuAct.

```
{
    "foreign": false,
    "real_name": "A_module",
    "fns": [
        {
            "arity": 1,
            "name": "f",
            "real_name": "A_module::f",
            "from_module": "A_module"
        }
    ]
}
```

Atrybut `foreign` określa czy moduł jest „obcy” (`true`) czy „własny” (`false`). Atrybut `real_name` określa nazwę modułu tak jak będzie prezentowana na poziomie bytecode'u. Atrybut `fns` jest listą funkcji, które są przez dany moduł eksportowane.

W elementach listy `fns` atrybuty mają następujące znaczenie:

1. `arity` określa liczbę parametrów formalnych funkcji
2. `name` określa nazwę funkcji widoczną z poziomu języka ViuAct
3. `real_name` określa pełną nazwę funkcji widoczną z poziomu bytecode'u
4. `from_module` określa pełną nazwę modułu, z którego pochodzi dana funkcja

6.5.3.1.2 Plik interfejsu dla modułów „obcych” Moduły „obce” języka ViuAct to moduły napisane w języku assemblera Viua VM lub w C++.

```
{
    "foreign": true,
    "real_name": "std::posix::network",
    "fns": [
        {
            "arity": 0,
            "name": "socket",
            "real_name": "Std::Posix::Network::socket",
            "bytecode_name": "std::posix::network::socket",
            "from_module": "Std::Posix::Network"
        }
    ]
}
```

Różnicą w stosunku do plików interfejsu dla modułow „własnych” jest dodatkowy atrybut w deklaracji eksportowanej funkcji – `bytecode_name` określający nazwę funkcji na poziomie bytecode'u. Nazwa ta nie musi pokrywać się z nazwą w atrybucie `real_name`, który określa pełną nazwę funkcji widoczną na poziomie języka ViuAct.

Dwa atrybuty są potrzebne ponieważ na tym poziomie następuje łączenie dwóch „światów”; języka ViuAct, który narzuca reguły tego jak muszą wyglądać nazwy modułów i funkcji, oraz języka assemblera Viua VM, który takich reguł nie narzuca. Dla modułów „własnych” atrybuty `bytecode_name` jest zbędny ponieważ dla nich nazwa widoczna na poziomie ViuAct i języka assemblera Viua VM jest taka sama.

6.5.3.2 Pliki zależności modułów (`.d`)

Pliki zależności są kodowane w formacie JSON.

```
{
    "imports": [
        {
            "module_name": "Std::Random",
            "real_name": "std::random",
            "foreign": true
        }
    ]
}
```

Atrybut `imports` przechowuje listę modułów importowanych przez moduł, którego zależności definiuje dany plik (co jest określane na podstawie nazwy pliku).

W elementach listy `imports` atrybuty mają następujące znaczenie:

1. `module_name` określa pełną nazwę modułu z punktu widzenia języka ViuAct
2. `real_name` określa pełną nazwę modułu widoczną z poziomu bytecode'u
3. `foreign` określa czy moduł jest „własny" (`false`) czy „obcy" (`true`)

6.6 Opis implementacji

Implementacja kompilatora jest bardzo prosta, jak na standardy tego typu projektów. Rysunek 6.1 na stronie 64 bardzo dobrze oddaje strukturę kompilatora dostarczonego jako wynik tej pracy.

Kompilacja rozpoczyna się od funkcji `main()` w pliku `cc.py`, która wywołuje funkcję `viuact.driver.compile_file()` w sposób określony przez argumenty podane w wierszu poleceń. `compile_file()` przygotowuje katalog kompilacji (domyślnie jest to `./build/_default`), wczytuje plik z kodem źródłowym przekazany do kompilacji i wywołuje funkcję `compile_text()` z tego samego modułu.

6.6.1 Analiza leksykalna

Funkcja `viuact.driver.compile_text()` na początek dokonuje analizy leksykalnej kodu źródłowego. Ta pierwsza faza przygotowania tekstu programu „do obróbki" jest wykonywana przez funkcję `lex()` z modułu `viuact.lexer`.

6.6.2 Analiza składniowa

6.6.2.1 Grupowanie

Po przeprowadzeniu analizy leksykalnej, funkcja `compile_text()` oddaje jej wynik (czyli listę tokenów) do funkcji `viuact.parser.group()`, która przeprowadza grupowanie tokenów odpowiadających różnym konstrukcjom językowym. Algorytm sterujący tym procesem jest trywialny:

1. utwórz pustą listę
2. rozważ pierwszy nieprzenalizowany token
3. jeśli ten token to `(` lub `{`, rozpocznij rekurencyjnie przetwarzanie strumienia tokenów od następnego tokenu, a wynik analizy dodaj jako pojedynczy element do listy
4. w innym przypadku dodaj token do listy
5. jeśli strumień tokenów jest pusty, zakończ algorytm

6. w innym przypadku kontynuuj od punktu 2.

W ten sposób tworzona jest lista list i pojedynczych tokenów reprezentujących konstrukcje językowe, z których składa się analizowany kod źródłowy. Na tym etapie łączone są również rozbudowane identyfikatory (np. `foo.bar.baz`) oraz wstawiane markery wyrażeń złożonych.

To, że wstępna faza analizy składniowej sprowadza się w dużej mierze do rekurencyjnego „odbijania się" od nawiasów, bez konieczności sprawdzania co jest pomiędzy nimi, i okazjonalnej konkatenacji jeśli parser napotka znak . jest zasługą wykorzystania składni wzorowanej na języku Lisp. Jest ona bardzo regularna co pozwala przeprowadzić proste parsowanie małym nakładem pracy i sprawdzając niewielką ilość informacji.

W wyniku grupowania otrzymujemy prymitywne drzewo składni abstrakcyjnej (*abstract syntax tree*, *AST*), które następnie jest „etykietowane" podczas następnej fazy: klasyfikacji grup.

6.6.2.2 Klasyfikacja grup

Klasyfikacją grup otrzymanych podczas fazy grupowania jako wynik funkcji `group()` zajmuje się funkcja `parse()` z modułu `viuact.parser`.

Również w tym przypadku algorytm jest bardzo prosty. Funkcja `parse()` sprawdza każdy element podstawowego AST i konwertuje go na pewien typ z modułu `viuact.group_types`. Jeśli jest to pojedynczy token konwersja następuje bezpośrednio; jeśli sprawdzanym elementem jest to lista algorytm determinuje co ta lista reprezentuje (na podstawie pierwszego elementu tejże, którym zawsze jest pojedynczy token) i postępuje rekurencyjnie etykietując każdy jej element, pozostawiając konwersję podstawowego elementu na koniec – po przeanalizowaniu elementów składowych.

Wykorzystanie algorytmu o tak niskim poziomie skomplikowania jest możliwe dzięki uważnemu zaprojektowaniu składni języka. Każda konstrukcja językowa jest ograniczona nawiasami – klamrowymi bądź okrągłymi – lub ma stałe miejsce w grupie. Poniżej, dla przykładu, zaprezentowane są wzorce odpowiadające wybranym konstrukcjom językowym:

```
(let    name expression)                                    1
(if     condition-expr true-expr false-expr)                2
(try    guarded-expr ...)                                   3
(actor  id expr*)                                           4
(+      lhs-expr rhs-expr)                                  5
```

Jeśli całość oprócz pierwszego tokenu zostanie „ukryta" to dalej oczywistym jest z jaką konstrukcją mamy do czynienia:

```
(let    ...)                                                1
(if     ...)                                                2
(try    ...)                                                3
(actor  ...)                                                4
(+      ...)                                                5
```

Wykorzystanie notacji polskiej (notacji prefiksowej) i takie zaprojektowanie składni, że każda konstrukcja ma stałą „szerokość" pozwala na zastosowanie chwytu „pierwszy token decyduje". Sprawia to też, że język ma rozpoznawalne tempo i styl, a konsekwencja układu nadaje mu elegancji.

W wyniku procesu „etykietowania" otrzymujemy „wzbogacone" AST składające się ze sklasyfikowanych konstrukcji językowych reprezentowanych przez wartości konkretnego typu zamiast ubogich w informację (bez dodatkowej weryfikacji) list i „samotnych" tokenów. Zamiast „listy list" otrzymujemy listę modułów, funkcji, wyliczeń itd. Takie AST znacząco upraszcza implementację kolejnej fazy kompilacji: generowania kodu.

6.6.3 Generowanie kodu wynikowego

Po przeprowadzeniu analizy składniowej i wygenerowaniu AST kompilator od razu przystępuje do generowania kodu. Jest to uwarunkowane pomienięciem w pracy etapów analizy semantycznej i optymalizacji. Te fazy są nieodzownymi elementami w kompilatorach „produkcyjnych" jednak zostały pominięte z uwagi na ograniczoną ilość czasu na wykonanie projektu gdyż zarówno optymalizacje jak i analiza semantyczna są elementami kompilatora, które mogą być rozwijane „w nieskończoność" – implementując kolejne algorytmy i metody optimalizacji, i tworząc coraz bardziej „inteligentne" systemy analizy statycznej (kończąc na systemach będących w stanie udowodnić poprawność bądź niepoprawność programu).

Analiza semantyczna (m.in. weryfikacja typów i analiza poprawności kodu) jest częściowo oddelegowana do assemblera Viua VM, który potrafi ją, w ograniczonym zakresie, przeprowadzić.

W kompilatorze języka ViuAct będącym wynikiem tej pracy generowanie kodu wynikowego jest implementowane przez dwa procesy działające naprzemiennie: *obniżanie poziomu* i *emisję instrukcji*.

Modułem implementującym fazę obniżenia poziomu jest `viuact.lowerer`, a fazę emisji instrukcji `viuact.emitter`. Funkcja `compile_text()` przekazuje AST otrzymane po etykietowaniu do funkcji `lower_file()` z modułu `viuact.lowerer` czym rozpoczyna proces generowania kodu wynikowego.

6.6.4 Obniżenie poziomu

„Obniżenie poziomu" polega na „rozbiciu" elementów AST na takie fragmenty, które mają swój bezpośredni (lub bliski bezpośredniemu) odpowiednik w języku assemblera Viua VM. Wygenerowanie kodu wynikowego dla takich fragmentów jest względnie proste.

Na tym etapie „znikają" konstrukcje istniejące w języku ViuAct, ale nie w języku assemblera Viua VM; są to na przykład wyliczenia i dowiązania *let*. Wyrażenia, które mają swoje odpowiedniki, ale są zbyt skomplikowane do prostego przetworzenia są rozbijane na prostsze formy (np. wyrażenia złożone, wywołania funkcji, instrukcje warunkowe).

To „rozbijanie" i „znikanie" konstrukcji języka daje tej fazie jej nazwę: konstrukcje języka wysokiego poziomu są tłumaczone na konstrukcje języka niskiego poziomu.

6.6.4.1 Obniżenie modułów

W pierwszej kolejności obniżane są moduły. Jest to wykonywane rekurencyjnie, a zagnieżdżone moduły są obniżane przed modułem, który je zawiera. Ten proces jest implementowany przez funkcję `viuact.lowerer.lower_module()`.

Importowanie Po obniżeniu modułów przetwarzane są importy modułów. Obniżenie musi być wykonane najpierw żeby umożliwić importowanie zagnieżdżonych modułów. Przetworzeniem importów zajmuje się funkcja `viuact.lowerer.perform_imports()`.

6.6.4.2 „Obniżenie" wyliczeń

Następnie „obniżane" są wyliczenia. Jest to jeden z najprostszych elementów kompilatora. Obniżenie wyliczeń sprowadza się do przypisania każdej wartości składowej wyliczenia pewnej wartości całkowitoliczbowej, a następnie zapisania wyniku tego przypisania i udostępnienia go do podglądu podczas dalszej kompilacji.

Przypisywanie wartości całkowitoliczbowych rozpoczyna się od 0, a każdej kolejnej wartości składowej pojedynczego wyliczenia jest przypisywana liczba całkowita o 1 większa od poprzedniej. Wyliczenie w języku ViuAct może mieć jedynie 2^{63} elementów z uwagi na użycie typu liczby całkowitej ze znakiem (rozdział 5.2.1.1 na stronie 44) jako typu bazowego dla wyliczeń.

6.6.4.3 Obniżenie funkcji

Obniżanie funkcji jest w założeniu prostym procesem podzielonym na kilka faz:

1. przesunięcia argumentów do rejestrów lokalnych

2. emisja wyrażenia będącego ciałem funkcji

3. przesunięcie wyniku do rejestru 0 (rejestru wartości zwracanych)

Skomplikowanie tego procesu jest „ukryte" przez fakt, że wyrażenie będące ciałem funkcji może być dowolnie złożone, ale zawsze jest emitowane przy użyciu tylko jednego (z punktu widzenia modułu obniżającego) wywołania funkcji `viuact.emitter.emit_expr()`. To, że wewnątrze `emit_expr()` może nastąpić „eksplozja" wywołań i działania nie jest widoczne z tego poziomu.

6.6.4.3.1 „Sklejanie" funkcji Funkcje są jedynymi elementami języka ViuAct, które po obniżeniu są wyraźnie widoczne w postaci linii kodu. Generowanie tekstu dla funkcji następuje w kilku etapach, których kolejność nie pokrywa się z tym co widać w pliku wynikowym.

Całość procesu spina stan (reprezentowany przez typ `State`) nazdorujący ilość alokowanych rejestrów, funkcje zagnieżdżone, oraz – w przypadku domknięć – informację o domykanym zakresie leksykalnym, do którego funkcja może się odwoływać.

Po utworzeniu stanu funkcji wykonywane są etapy opisane poniżej.

Przesunięcia argumentów Dla każdego argumentu emitowana jest instrukcja `move` (opisana dokładnie w A.2.1.1 na stronie 146) przesuwająca wartość przekazaną jako argument z rejestru argumentów do rejestru lokalnego.

Wiersze dopisywane są do listy instrukcji składajacych się na funkcję.

Emisja ciała funkcji Po przesunięciu argumentów emitowane jest wyrażenie będące ciałem funkcji. Wiersze dopisywane są na końcu listy instrukcji składajacych się na funkcję.

Utworzenie wiersza sygnatury funkcji i całościowej listy wierszy Po wyemitowaniu wyrażenia będącego ciałem funkcji tworzona jest całościowa lista wierszy funkcji. Jako jej pierwszy element wstawiana jest linia będąca syganturą funkcji zgodną z formatem oczekiwanym przez assembler Viua VM: „`.function:` *`nazwa_funkcji/ilość_argumentów`*". Na przykład „`.function: say_hello_to/1`".

Zapis alokacji rejestrów Po sygnaturze, jak pierwszy wiersz ciała właściwego (instrukcji składających się na funkcję) dopisywana jest instrukcja `allocate_registers` alokująca taką ilość rejestrów lokalnych jaka jest potrzebna emitowanej funkcji do działania.

Ilość potrzebnych funkcji rejestrów lokalnych jest śledzona przez obiekt reprezentujący stan funkcji. Każdy nowy potrzebny rejestr jest „oddawany" przez funkcję `State.get_slot()` co powoduje, że stan funkcji zna dokładną ilość rejestrów „zaalokowanych" podczas emisji kodu funkcji.

Zapis ciała wewnętrznego Po zapisie zaalokowanych rejestrów do całościowej listy wierszy dopisywane są wiersze wygenerowane przez wyrażenie będące ciałem funkcji.

Przesunięcie wartości zwracanej i epilog funkcji Po zapisie ciała wewnętrznego do całościowej listy wierszy dopisywana jest instrukcja `move` przesuwająca wynik wyrażenia będącego ciałem funkcji do rejestru 0 (rejestru wartości zwracanej).

Następnie do całościowej listy wierszy dopisywana jest instrukcja `return` oraz dyrektywa `.end`, a proces generowania funkcji jest zakończony.

6.6.4.3.2 Funkcje zagnieżdżone Funkcje w języku ViuAct mogą być zagnieżdżane. Z tego powodu funkcja `lower_function()` może zwrócić jeden lub więcej wyników. Aby ułatwić obsługę takich sytuacji `lower_function()` deleguje swoją pracę do funkcji `output_function_body()`, która zwraca listę wszystkich funkcji wyemitowanych jako wynik obniżenia funkcji wejściowych. Następnie każdy element (tj. każda funkcja) tej listy jest przekazywany do funkcji `lower_function_body()`, która przetwarza „surowe" wyniki na formę tekstową gotową do zapisu w pliku z kodem wynikowym (tj. programem przetłumaczonym z języka ViuAct na jezyk assemblera Viua VM).

Obniżenie funkcji jest ostatnim elementem fazy „obniżania poziomu". Jest też punktem styku modułu `viuact.lowerer` z modułem `viuact.emitter` – funkcja `output_function_body()` jest jedyną funkcją wywołującą funkcje modułu `emitter` podczas tej fazy.

Emisja instrukcji jest szczegółowo opisana w rozdziale 6.6.5.

6.6.5 Emisja instrukcji

Emisja instrukcji tłumaczy przetworzone konstrukcje języka ViuAct na ich bezpośrednie odpowiedniki w języku assemblera Viua VM. Emisją instrukcji zajmuje się moduł `viuact.emitter`.

6.6.5.1 Emisja wyrażeń prostych

Emisja wyrażeń prostych jest implementowana przez funkcję `emit_expr()`. Jest to również funkcja „ogólna" zajmująca się rozsyłaniem poszczególnych typów wyrażeń do ich dedykowanych funkcji emisji. Bezpośrednio `emit_expr()` emituje jedynie instrukcje odpowiadające za konstruktory wartości typów prostych (literały) i konstruktory wartości złożonych.

Funkcja `emit_expr()` zwraca „slot" (6.2.1.3 na stronie 70) identyfikujący rejestr, w którym jest przechowywany wynik. Ten slot może być „pusty" (tj. nie identyfikować żadnego rejestru) jeśli funkcja `emit_expr()` zostanie poinformowana, że wynik danego wyrażenia nie będzie użyty.

Nie wszystkie wyrażenia muszą generować wiersze kodu w każdym kontekście. Przykładem jest odwołanie do dowiązania *let* – jeśli jest samodzielnym wyrażeniem na końcu wyrażenia złożonego to musi wyemitować instrukcję `copy` aby zwrócić wartość, ale jeśli jest wyrażeniem przekazywanym argument to wystarczy zwrócić jego slot. To czy wymagane jest wyemitowanie instrukcji i czy wymagane jest zwrócenie slotu determinują dwa parametry funkcji `emit_expr()` (oba przyjmują wartości `bool`):

`must_emit` określa czy musi być wyemitowana instrukcja

`toplevel` określa czy wyrażenie może pominąć zwracanie „zapisywalnego" (tj. możliwego do umieszczenia w rejestrze) wyniku ponieważ nie ma nad sobą żadnego innego wyrażenia, które mogłoby ten wynik wykorzystać (przykład: wyrażenia składowe wyrażenia złożonego, poza ostatnim wyrażeniem składowym)

6.6.5.1.1 Emisja literałów Emisja literałów polega na wygenerowaniu instrukcji, które utworzą w rejestrze wartości typów prostych: string, liczbę całkowitą, liczbę zmiennoprzecinkową, lub wartość logiczną.

Odpowiadają im następujące instrukcje:

string	`text Rd ...`
liczba całkowita	`integer Rd ...`
liczba zmiennoprzecinkowa	`float Rd ...`
wartość logiczna	`izero Rd` w połączeniu z `not Rd`

Wartości proste mogą być wyemitowane jako pojedyncze instrukcje; wyjątkiem jest wartość logiczna, która nie ma bezpośredniego konstruktora i jest „składana" z dwóch (w przypadku `true`) lub trzech (w przypadku `false`) instrukcji.

6.6.5.1.2 Emisja konstruktorów Konstruktory są dostępne dla typów złożonych – wektorów i struktur. Mają składnię identyczną z wywołaniami funkcji, jednak emitowane są jako pojedyncze instrukcje `vector Rd` dla wektorów i `struct` dla struktur.

6.6.5.1.3 Emisja odwołań do dowiązania *let* Emisja odwołań do dowiązań *let* może, ale nie musi wygenerować instrukcji. Jeśli odwołanie jest emitowane jako część większego wyrażenia, na przykład:

```
(let x (+ lhs rhs))
```

to dla `lhs` i `rhs` nie zostaną wyemitowane osobne instrukcje; oba odwołania zawierają się w instrukcji `add` (jako adresy rejestrów). Jeśli jednak odwołanie jest wyrażeniem „samym w sobie" jak `x` na listingu poniżej:

```
(let foo (a b) {
    (let x ...)
    ...
    x
})
```

to wyemitowana zostanie dla niego instrukcja `copy` tworząca nową wartość będącą kopią tej, do której odwołanie się odwołuje (`x` ewaluuje do siebie samego).

6.6.5.2 Emisja wyrażeń złożonych

Emisja wyrażeń złożonych jest implementowana przez funkcję `emit_compound_expr()`.

Wyrażenie złożone jest listą n wyrażeń prostych. Każde z nich nich jest emitowane za pomocą funkcji `emit_expr()`. Dla wyrażeń od 0 do $n-1$ bezpośrednie wyniki tych emisji są ignorowane (tj. wyniki nie są przypisywane do żadnego rejestru). Zapisywane są tylko efekty uboczne, jak np. utworzenie dowiązania *let* do wartości, wypisanie wartości na ekran lub wpisanie jej do pliku). Dla każdego wyrażenia składowego są generowane są wiersze, które implementują dane wyrażenie.

Emisja wyrażeń od 0 do $n-1$ jest zaimplementowana przy pomocy pętli, w której wywoływana jest funkcja `emit_expr()` z każdym kolejnym wyrażeniem składowym na wejściu.

Zapisywany jest wynik ostatniego wyrażenia składowego wyrażenia złożonego. Wartość ostatniego wyrażenia składowego jest jednocześnie wartością zwracaną całego wyrażenia złożonego.

6.6.5.3 Emisja dowiązań *let*

Emisja dowiązań *let* jest implementowana przez funkcję `emit_let_binding()`. Wywołuje ona funkcję `emit_expr()` przekazując fałsz jako wartość argumentu `toplevel`, a następnie otrzymany slot zapisuje w stanie funkcji pod nazwą podaną w kodzie źródłowym.

6.6.5.4 Emisja wywołań funkcji wbudowanych

Emisja wywołań funkcji „wbudowanych" jest implementowana przez funkcję `emit_builtin_call()`.

6.6.5.5 Emisja wywołań funkcji

Emisja wywołań funkcji jest implementowana przez funkcję `emit_call()`.

6.6.5.6 Emisja wywołań aktorów

Emisja wywołań aktorów jest implementowana przez funkcję `emit_actor_call()`.

6.6.5.7 Emisja wywołań odroczonych

Emisja wywołań odroczonych jest implementowana przez funkcję `emit_deferred_call()`.

6.6.5.8 Emisja wywołań watchdog

Emisja wywołań watchdog jest implementowana przez funkcję `emit_watchdog_call()`.

6.6.5.9 Emisja wywołań ogonowych

Emisja wywołań ogonowych jest implementowana przez funkcję `emit_tail_call()`.

6.6.5.10 Emisja wyrażeń warunkowych

Emisja wyrażeń warunkowych jest implementowana przez funkcję `emit_if()`.

6.6.5.11 Emisja funkcji zagnieżdżonych

Emisja funkcji zagnieżdżonych jest implementowana przez funkcję `emit_function()`.

6.6.5.12 Emisja wyrażeń *try*

Emisja wyrażeń *try* jest implementowana przez funkcję `emit_try_expr()`.

6.6.5.13 Emisja przypisania do pola struktury

Emisja przypisania do pola struktury jest implementowana przez funkcję `emit_field_assignment()`.

6.6.5.14 Emisja dostępu do pola struktury

Emisja dostępu do pola struktury jest implementowana przez funkcję `emit_struct_field_access()`.

6.7 Testowanie

Testy kompilatora polegały na opracowaniu zestawu przypadków testowych (w formie przykładowych programów napisanych w języku ViuAct) pokrywających cały zakres konstrukcji językowych zawartych w specyfikacji języka ViuAct. Oprócz prostych programów sprawdzających działanie konkretnych konstrukcji (np. definicji funkcji zagnieżdżonych, wyrażeń warunkowych, wyrażeń złożonych) potrzebne były też „większe" programy sprawdzające współpracę kilku konstrukcji językowych naraz lub integrację i poprawność działania kompilatora (np. programy wielomodułowe, programy korzystające z modułów FFI).

6.7.1 Wykonanie testów

Każdy przypadek testowy składa się z trzech plików:

1. *test*`.sh`
2. *test*`.lisp`
3. *test*`.text`

Plik `.sh` zawiera polecenia potrzebne do skompilowania programu testowego. Jest to szczególnie potrzebne przy testach sprawdzających integrację wielu części języka (np. współpracy z modułami FFI).

Plik `.lisp` zawiera kod źródłowy przykładowego programu. Tych plików może być więcej niż jeden w przypadku bardziej skomplikowanych programów.

Plik `.text` zawiera spodziewany wynik działania programu. Każdy program testowy miał za zadanie ostatecznie wydrukować coś na standardowe wyjście, a framework testowy porównywał to z zawartością pliku `.text` i na tej podstawie oceniał czy test się powiódł czy nie.

6.7.2 Trudności w testowaniu

Przeszkodą w testowaniu był między innymi niedeterminizm wynikający z równoległego działania aktorów. Żeby znormalizować wyniki i umożliwić ich powtarzalną weryfikację trzeba było uciekać się do „sztuczek", np. sortowania linii na standardowym wyjściu programu testowego.

Także samo założenie, że każdy program miał wyprodukować na standardowym wyjściu wyniki swojej pracy nie zawsze było pomocne i zmuszało do pisania programów testowych w określony sposób – tak żeby zawsze miały co wypisać na ekran. Viua VM umożliwia generowanie „śladów wykonania" – zapisów wszystkich wykonanych w programie instrukcji wewnątrz wszystkich działających aktorów. Analiza tych śladów zapewniłaby niemal perfekcyjną dokładność i precyzję testów jednak z uwagi na ograniczoną ilość czasu na projekt takie podejście byłoby nierozsądne gdyż opracowanie testów zajmowałoby znacznie więcej czasu niż w przyjętym ostatecznie rozwiązaniu.

6.8 Instrukcja użytkownika kompilatora języka ViuAct

Tradycja nakazuje, aby pierwszym programem jaki pisze się w nowym języku był program, który wypisze na ekran napis „*Hello World!*". W ViuAct ten program wygląda następująco:

```
(let main () {
    (print "Hello World!")
    0
})
```

Aby skompilować ten program, należy wykonać w konsoli następujące polecenia:

```
$ viuact-cc --mode exec ./hello_world.lisp
$ viuact-opt ./build/_default/hello_world.asm
```

Kod wykonywalny (*bytecode* wykonywalny przez Viua VM) będzie umieszczony w pliku `hello_world.bc` w katalogu `./build/_default`. Aby go uruchomić należy użyć jądra Viua VM:

```
$ viua-vm ./build/_default/hello_world.bc
Hello World!
$
```

Nazwy plików pośrednich są wywodzone z nazwy pliku źródłowego:

example`.lisp` plik z kodem źródłowym w języku ViuAct

example`.asm` plik wynikowy kompilatora, z kodem źródłowym w języku assemblera Viua VM

example`.bc` plik wynikowy assemblera Viua VM, zawierający wykonywalny bytecode

Pliki `.asm` i `.bc` są umieszczane w katalogu `./build/_default`.

6.8.1 Opcje kompilatora

Jedyną opcją kompilatora jest `--mode`, która przyjmuje dwie możliwe wartości:

`exec` jeśli plik źródłowy definiuje moduł wykonywalny

`module` jeśli plik źródłowy definiuje moduł biblioteczny

6.8.2 Zmienne środowiskowe

Zachowanie kompilatora można częściowo zmodyfikować ustawiając zmienne środowiskowe.

6.8.2.1 `DEFAULT_OUTPUT_DIRECTORY`

Kontroluje katalog, w którym kompilator składuje pliki wynikowe. Domyślnie pliki wynikowe są składowane w katalogu `./build/_default`.

6.8.2.2 `VIUAC_LIBRARY_PATH`

.

Jak `LD_LIBRARY_PATH`.

6.8.2.3 `VIUA_ASM_PATH`

Kontroluje ścieżkę do assemblera Viua VM.

6.8.2.4 `VIUAC_VERBOSE`

Wartość `true` powoduje wyświetlenie komunikatów podczas kompilacji.

6.8.2.5 `VIUAC_DEBUGGING`

Wartość `true` włącza komunikaty debugowania.

6.8.2.6 `VIUAC_INFO`

Wartość `true` włącza dodatkowe komunikaty informacyjne.

6.8.2.7 `VIUAC_DUMP_INTERMEDIATE`

Wartość `tokens` powoduje zrzut strumienia tokenów do pliku *`example`*`.tokens`. Wartość `exprs` powoduje zrzut drzewa składni do pliku *`example`*`.expressions`. Można podać obie wartości, oddzielone przecinkiem.

Część III

Program ViuaChat

Rozdział 7

Program ViuaChat – założenia

7.1 Wprowadzenie

Ostatnią częścią pracy inżynierskiej jest program ViuaChat - chat internetowy o funkcjonalności podobnej do IRC[1]. Umożliwia użytkownikom logowanie się do sieci, zakładanie pokojów tematycznych do rozmów, oraz rozmowy „prywatne" (tj. wyłączne dla dwóch użytkowników).

W poniższym rozdziale zdefiniowano wymagania dla czatu ViuaChat. Ich opracowanie nastąpiło na podstawie analizy otoczenia aplikacji oraz analizy potrzeb projektu w stosunku do niej. W ramach tego procesu nastąpiły:

- analiza otoczenia, wraz z klientami;
- wskazanie kontekstu biznesowego systemu;
- określenie udziałowców;
- wyszczególnienie i uporządkowanie zasad biznesowych, jakie zostały założone w stosunku do aplikacji;
- opracowanie historyjek na podstawie ustalonych zasad biznesowych.

Uwaga: Niniejszy rozdział nie dotyczy języka ViuAct ani jego kompilatora.

7.1.1 Odbiorcy

Rozdział został pierwotnie napisany przede wszystkim dla członków zespołu, aby ułatwić im współpracę - w szczególności wówczas, gdy funkcjonalności czatu mogły pociągać za sobą modyfikację zestawu bibliotek Viua VM bądź struktury składni projektowanego języka ViuAct.

7.2 Czat w kontekście

7.2.1 Kontekst biznesowy

Niniejszy czat stanowi część szerszego kontekstu, jakim jest potrzeba zademonstrowania działania języka ViuAct oraz całego środowiska wytwórczego powiązanego z maszyną wirtualną ViuaVM.

Cel demonstracyjny był pierwszym i najważniejszym, jaki przyświecał budowie czatu. Ponadto, proces wytwórczy pozwolił przetestować wydajność całego środowiska w jego praktycznym wymiarze. Tym samym, możliwe było poprawienie konstrukcji kompilatora lub zastosowanych konstrukcji językowych ViuAct, podnosząc tym samym jego użyteczność.

Wszelcy odbiorcy dla aplikacji czatu zostaną, podobnie jak sama aplikacja, skonstruowani na cele demonstracyjne. Nie powinni oni odbiegać od modelowych odbiorców podobnych komunikatorów, tak, aby potencjalny, poczatkujący użytkownik środowiska ViuaVM mógł zrozumieć intencje stojące za rozwiązaniami zastosowanymi w ViuaChat oraz przenieść je do swoich pierwszych programów.

[1] IRC (*ang. Internet Relay Chat*) – usługa sieciowa, pozwalająca na tekstową komunikację w czasie rzeczywistym.

Rysunek 7.1: Ilustracja środowiska wytwórczego wraz zasięgiem, którym są objęte prace przewidziane projektem inżynierskim

7.2.2 Udziałowcy

Poniżej wyszczególniono udziałowców, mających wpływ na rozwój czatu.

Karta udziałowca	
Identyfikator	UN-01
Nazwa	ViuaVM
Opis	Maszyna wirtualna, oparta o przechowywanie danych w rejestrach zamiast *płaskich* tablic pamięci. Stanowi ona platformę, na której musi zostać uruchomiony serwer czatu. Ponieważ jej największym atutem jest zorientowanie na kod wykonywany współbieżnie, sam serwer czatu powinien tę cechę wykorzystywać w maksymalnym stopniu.
Typ	Nieożywiony, bezpośredni
Punkt widzenia	ViuaVM jest absolutnie nieodzownym elementem projektu, a serwer czatu stanowi przede wszystkim dowód jej użyteczności. O ile jądro maszyny nie ma być poddawane już żadnym zmianom i być wykorzystane takie, jakie było na inicjalnym etapie pracy inżynierskiej, o tyle dopuszcza się poszerzanie funkcjonalności o dodatkowe biblioteki zewnętrzne.
Ograniczenia	Maszyna wirtualna, jakkolwiek stanowi istotny czynnik dla decyzji w zakresie architektury czy konstrukcji oprogramowania, nie powinna mieć wpływu na wymagania stricte biznesowe, jest bowiem jedynie środowiskiem do uruchamiania współbieżnych programów, *przezroczystym* dla końcowego użytkownika czy zleceniodawcy zrealizowanego oprogramowania.
Wymagania	**WS-01** W rozwiązaniu należy wykorzystać środowisko Viua VM i język ViuAct.

Karta udziałowca	
Identyfikator	UO-01
Nazwa	Opiekun pracy inżynierskiej
Opis	Pracownik uczelni, wyznaczony do opieki nad całym projektem inżynierskim - nadzorowania jego postępów, wskazywania problemów oraz sugerowania decyzji podwyższających walor pracy oraz szanse na jej skuteczne obronienie. Ma również zasadniczy wpływ na decyzję o dopuszczeniu pracy do recenzji.
Typ	Ożywiony, bezpośredni
Punkt widzenia	Opiekun pracy patrzy na czat przede wszystkim przez pryzmat jego użyteczności jako efektownego przykładu implementacji modelu aktora w praktycznym, programistycznym ujęciu. Stąd, jego uwaga skupia się przede wszystkim na konstrukcjach językowych, strukturach oraz rozwiązaniach od strony kodu źródłowego. Czat stanowi jedynie pretekst do przeniesienia teoretycznych, akademickich rozważań na praktyczny grunt.
Ograniczenia	Opiekun pracy, pomimo bycia jej nadzorcą i posiadania istotnych uprawnień decyzyjnych w stosunku do jej dalszego rozwoju, nie ma możliwości bieżącego śledzenia prac oraz podejmowania decyzji w przypadku konkretnych problemów. Powinien zachować dystans, pozwalający na samodzielną realizację projektu przez zespół. Stąd, jego faktyczny udział ogranicza się do udzielania porad w przypadku strategicznych kierunków, w jakich będzie podążała grupa, a także doraźnego recenzowania ograniczonej puli zagadnień, wyłapanych w trakcie wspólnych spotkań.
Wymagania	**WF-03** Jako użytkownik serwera czatu, chcę po wpięciu do pokoju zobaczyć ostatnie wiadomości wysłane przed moim dołączeniem, aby dowiedzieć się, co się tam obecnie dzieje. **WF-12** Jako administrator, chcę utworzyć nowy pokój, aby umożliwić użytkownikom konwersację w węższym gronie. **WF-13** Jako administrator, chcę usunąć zbędny pokój, aby utrzymać podrządek na swoim serwerze. **WS-01** W rozwiązaniu należy wykorzystać środowisko Viua VM i język ViuAct. **WW-02** W procesie wytwórczym należy uprzednio przygotować specyfikację przypadków użycia – co wynika z wymagań uczelni.

Karta udziałowca	
Identyfikator	UO-02
Nazwa	Członek zespołu ds. ViuAct
Opis	Student i członek zespołu, skupiający się w pierwszej kolejności nad rozwojem języka programowania ViuAct, jego kompilatora oraz ewentualnego rozbudowania maszyny ViuaVM o kolejne, zewnętrzne biblioteki.
Typ	Ożywiony, bezpośredni
Punkt widzenia	Przede wszystkim, postrzega czat jako produkt, realizowany na końcowej platformie. Stąd, musi brać udział w formułowaniu wymagań związanych z ViuaVM oraz językiem ViuAct. Jego zadaniem jest doprowadzenia do zaprojektowania czatu w sposób, który ukaże możliwości ViuAct jako solidnego, kompletnego rozwiązania. Przy tym, musi trzymać rękę na pulsie i reagować, gdyby pojawiały się przeszkody w zaprogramowaniu czatu, wynikające z niedoskonałości środowiska wytwórczego. Podczas współudziału w definiowaniu wymagań, istotny jest dla niego zakres pracy, wiążący się z urzeczywistnianiem poszczególnych, proponowanych wymagań. Zbyt rozbudowany czat może opóźnić prace nad całym projektem, a w efekcie - zniweczyć trud włożony w rozwój języka programowania i dedykowanego mu kompilatora.
Ograniczenia	Jego udział w pracach nad czatem jest z gruntu nieograniczony. Jednakże, decydując się na podział odpowiedzialności podyktowany zespołowym charakterem projektu oraz własnymi ograniczeniami czasowymi, zrezygnował z decydowania o biznesowej części wymagań, faktycznie pozostając w roli konsultanta.
Wymagania	**WF-01** Jako użytkownik serwera czatu, chcę się do niego zalogować, aby zobaczyć listę pokojów dyskusyjnych. **WF-02** Jako użytkownik serwera czatu, chcę się wpiąć do pokoju, aby wziąć udział w dyskusji. **WF-04** Jako użytkownik serwera czatu, chcę wysłać wiadomość do pokoju w który jestem wpięty, aby zobaczyli ją inni uczestnicy dyskusji. **WF-09** Jako użytkownik serwera czatu, chcę wysłać wiadomość prywatną do jednego użytkownika, aby prowadzić z nim ciągłą konwersację. **HN-02** Loginy i hasła administratorów są gromadzone w plikach konfiguracyjnych, a po uruchomieniu serwera – w jego pamięci podręcznej. **WS-01** W rozwiązaniu należy wykorzystać środowisko Viua VM i język ViuAct.

Karta udziałowca	
Identyfikator	UO-03
Nazwa	Członek zespołu ds. Czatu
Opis	Student i członek zespołu, odpowiedzialny za prace nad czatem
Typ	Ożywiony, bezpośredni
Punkt widzenia	Czat stanowi dla niego, obok dokumentacji, najistotniejszą część przedsięwzięcia. Musi z jednej strony nauczyć się poruszać w nowym, dynamicznie zmieniającym się środowisku programistycznym, a z drugiej strony - zrealizować przy jego użyciu serwer czatu, który pokaże jego możliwości i zastosowania innym nowicjuszom. Podczas współudziału w definiowaniu wymagań, istotny jest dla niego zakres końcowych funkcjonalności czatu. Nie może być zbyt wąski. Z drugiej strony, konstrukcja programu powinna pozostać prosta i przejrzysta. Przykładowy kod nie powinien odstraszać potencjalnego programisty, dla którego cała koncepcja ViuaVM oraz modelu aktorów może wydawać się na pierwszy rzut oka nieco egzotyczna.
Ograniczenia	Nie ma w zasadzie organizacyjnych czy kompetencyjnych ograniczeń dla formułowania wymagań. Nie oznacza to jednak, że może definiować wymagań w oderwaniu od pozostałych udziałowców (ich role i punkty widzenia opisano wcześniej).
Wymagania	**Wszystkie, za wyjątkiem:** **HN-02** Loginy i hasła administratorów są gromadzone w plikach konfiguracyjnych, a po uruchomieniu serwera – w jego pamięci podręcznej. **WW-02** W procesie wytwórczym należy uprzednio przygotować specyfikację przypadków użycia – co wynika z wymagań uczelni.

7.2.3 Charakterystyka użytkowników

Na etapie analizy kontekstu, w którym ma zostać zaprojektowany i zrealizowany czat, zadecydowano o zaprojektowaniu następujących, modelowych użytkowników docelowego oprogramowania:

1. **Użytkownik tymczasowy.** Typ użytkownika, którego konto jest tworzone podczas połączenia z serwerem czatu oraz niszczone po jego zakończeniu. Podczas łączenia z czatem, nie będzie musiał się autoryzować przy użyciu hasła, a deklarować tylko unikalną nazwę, nie powtarzającą się z nazwą innego użytkownika, posiadającego konto na danym serwerze czatu.

2. **Administrator.** To użytkownik, który jest dodatkowo wyróżniony i posiada uprawienia do szeroko pojętego zarządzania serwerem (w tym - pozostałymi użytkownikami). Konto administratora jest utrzymywane przez serwer pomiędzy połączeniami do czatu. Każdorazowo, przed rozpoczęciem sesji połączenia z serwerem, muszą się dodatkowo autoryzować przy użyciu hasła. Równocześnie, ich nazwa jest zarezerwowana wyłącznie do jego użytku oraz niedostępna dla użytkowników tymczasowych. Nie wyróżnia się wśród administratorów żadnych dodatkowych, szczególnych ról (np. superadministrator, właściciel).

Poza wspomnianymi różnicami, wszyscy użytkownicy po rozpoczęciu sesji połączenia mają prawo do dołączania do pokojów oraz wysyłania sobie nawzajem wiadomości prywatnych. Łącznie, pula użytkowników przebywających na serwerze czatu w jednym momencie nie powinna przekraczać 320, zaś w jednym pokoju - nie więcej niż 32. W związku z tym można przyjąć, że czat jest przeznaczony dla niewielkich społeczności, np. szkolnych, uczelnianych czy hobbystycznych.

7.2.4 Istniejąca infrastruktura

- **Komputer A**

  - komputer przenośny z procesorem Intel Core i5 oraz systemem operacyjnym Windows 10
  - XAMPP 7.2.7, obejmujący serwer Apache 2.4 oraz interpreter języka PHP w wersji 7.2.7.
  - oprogramowanie VirtualBox z uruchomioną maszyną wirtualną z systemem operacyjnym Linux Mint 19 „Tara".

- **Komputer B**
  - komputer przenośny, na którym zainstalowano system operacyjny Linux Mint 19 „Tara"
  - GNU Compiler Collection 8.2
  - wirtualna maszyna Viua VM w wersji 0.9.0
  - *należy doinstalować serwer Nginx, odpowiedzialny za wysłanie frontendu do użytkownika łączącego się z czatem oraz za handshake Websocketu*

- **Smartfon C**
  - telefon LG K11 z systemem operacyjnym Android
  - przeglądarka mobilna Google Chrome

7.3 Zasady biznesowe

Zidentyfikowane zasady pogrupowano w 3 kategorie, biorąc pod uwagę podstawowe bloki funkcjonalności. Przydzielenie do kategorii jest sygnalizowanie literą alfabetu, będącą prefiksem identyfikatora danej zasady. Numeracja identyfikatorów może być nieciągła, gdyż część z wymagań została usunięta w trakcie prac nad dokumentacją.

7.3.1 System użytkowników [ZU]

ID	Zasada biznesowa	Priorytet
ZU-01	Podczas wejścia na czat, użytkownikowi pokazuje się monit z polem do wpisania nazwy użytkownika.	M
ZU-02	Użytkownicy bez stałego konta podczas logowania podają tylko nazwę użytkownika, pole hasła pozostaje puste.	M
ZU-03	Nazwa użytkownika to ciąg od 3 do 32 alfanumerycznych znaków.	M
ZU-04	Można rozpocząć sesję jako użytkownik, pod warunkiem, że zadeklarowana nazwa nie będzie powtarzać się z nazwami ju ż zalogowanych użytkowników.	M
ZU-05	Monit podczas wejścia na czat jest wyposażony w pole do wpisania hasła (nieobowiązkowe).	S
ZU-06	Administrator podczas logowania podają nazwę i odpowiadające mu hasło.	S
ZU-07	Konta administratorów są utrzymywane na serwerze w postaci par wartości: nazwa użytkownika i hasło.	S
ZU-08	Nie można rozpocząć sesji użytkownika o nazwie, która pasuje do istniejącego konta, jeżeli nie zostanie podan e prawidłowe hasło (nie można podszywać się pod nazwy administratorów).	S
ZU-09	Można rozpocząć sesję jako użytkownik bez podawania hasła, pod warunkiem, że zadeklarowana nazwa nie będzie powtarza ć się z nazwami kont administratorów i już zalogowanych użytkowników tymczasowych.	S
ZU-10	W okienkach czatu, loginy administratorów są pogrubione i pokolorowane na czerwono.	S
ZU-11	Administratorzy mają prawo przeglądać nazwy pokojów na serwerze.	M
ZU-12	Administratorzy mają prawo tworzyć i usuwać pokoje.	S
ZU-14	Administratorzy mają prawo wyrzucać użytkowników z pokojów.	C
ZU-15	Administratorzy mają prawo wyrzucać użytkowników z serwera.	C
ZU-16	Administratorzy mają prawo przeglądać nazwy i poziomy uprawnień użytkowników.	M
ZU-18	Administratorzy mają prawo zmieniać swoje hasła użytkowników.	C

7.3.2 Pokoje [ZP]

ID	Zasada biznesowa	Priorytet
ZP-01	Pokoje to właściwe czaty – tam użytkownicy mogą wejść i pisać do siebie nazwajem	M
ZP-02	Każdy pokój ma unikalną nazwę będącą ciągiem alfanumerycznym od 3 do 32 znaków	M
ZP-03	Lista pokojów jest widoczna dla każdego użytkownika po zalogowaniu się do serwera czatu	M
ZP-04	Użytkownik może być równocześnie wpięty do jednego pokoju	M
ZP-05	Wiadomość wysłana w pokoju jest widoczna w oknie pokoju dla wszystkich użytkowników podpiętych do tego pokoju	M
ZP-06	Użytkownik może się samodzielnie wypiąć z pokoju, do którego jest wpięty	S
ZP-07	Pokój może mieć ustanowione hasło, które użytkownik musi wpisać przed podpięciem się do niego	W
ZP-08	Nowo wpięty użytkownik widzi 10 najnowszych wiadomości, które zostały wysłane do pokoju tuż przed wpięciem	S
ZP-09	Serwer czatu automatycznie wysyła do pokoju wiadomości, zawierające powiadomienia o wydarzeniach związanych z pokojem, tzw. wiadomości systemowe	S
ZP-10	Wiadomości systemowe są niepodpisane przez żadnego użytkownika i zapisane kursywą	C
ZP-11	Wiadomość systemowa zostaje wysłana podczas wpięcia się nowego użytkownika do pokoju	S
ZP-12	Wiadomość systemowa zostaje wysłana podczas wypięcia użytkownika z pokoju	S
ZP-13	Wiadomość systemowa zostaje wysłana, gdy użytkownik wpięty do pokoju traci połączenie z serwerem czatu	S
ZP-14	Wiadomość systemowa zostaje wysłana, gdy użytkownik zostaje wyrzucony z pokoju	S

7.3.3 Prywatne wiadomości [ZW]

ID	Zasada biznesowa	Priorytet
ZW-01	Wiadomości prywatne to wiadomości, które są wysyłane do konkretnego odbiorcy, innego niż nadawca. Są one widoczne wyłącznie dla nadawcy i odbiorcy takiej wiadomości	M
ZW-06	Użytkownik dysponuje dedykowanym oknem, w którym widzi wiadomości prywatne.	M
ZW-07	Z okna wiadomości prywatnych można odbierać i wysyłać wyłącznie wiadomości prywatne, których nadawcą/odbiorcą jest wybrany użytkownik	M
ZW-08	W oknie wiadomości prywatnych można przeglądać wiadomości wysłane do i odebrane od jednego, wybranego użytkownika.	S
ZW-09	W oknie wiadomości prywatnych można wysyłać wiadomości wyłącznie do nadawcy, którego wiadomości są w danym momencie pokazywane.	S
ZW-10	Wiadomości prywatne są utrzymywane dopóki nadawca i odbiorca mają aktywną sesję na serwerze.	S
ZW-11	Dla każdej pary użytkowników, na serwerze jest gromadzone co najwyżej 100 wiadomości prywatnych.	S

7.4 Wymagania

7.4.1 Wymagania funkcjonalne

Ponieważ obraną metodologią wytwarzania aplikacji jest *mini-Scrum*, należący do kategorii metodyk zwinnych, wymagania funkcjonalne ujęto w formie historyjek (*user stories*).

Identyfikator	WF-01
Treść	Jako użytkownik serwera czatu, chcę się do niego zalogować, aby zobaczyć listę pokojów dyskusyjnych.
Powiązane zasady biznesowe	ZU-01 Podczas wejścia na czat, użytkownikowi pokazuje się monit z polem do wpisania nazwy użytkownika. ZP-03 Lista pokojów jest widoczna dla każdego użytkownika po zalogowaniu się do serwera czatu
Kryteria akceptacji	1. Po wejściu na czat bez rozpoczętej sesji, pokazuje się monit o podanie nazwy użytkownika. 2. Po wpisaniu nazwy użytkownika i zatwierdzeniu, użytkownik rozpocznie sesję na serwerze czatu. 3. Tuż po rozpoczęciu sesji czatu, użytkownik zobaczy listę pokojów.

Identyfikator	WF-02
Treść	Jako użytkownik serwera czatu, chcę wpiąć się do pokoju, aby wziąć udział w dyskusji.
Powiązane zasady biznesowe	ZP-01 Pokoje to właściwe czaty - tam użytkownicy mogą wejść i pisać do siebie nawzajem
Kryteria akceptacji	1. Użytkownik, który ma otwartą sesję połączenia z serwerem czatu i nie jest wpięty do żadnego pokoju, zobaczy listę pokojów. 2. Użytkownik, po kilknięciu w liście pokojów na nazwę pokoju, zostanie do niego podpięty 3. Użytkownik po wpięciu się do pokoju zobaczy okno pokoju 4. Użytkownik, który ma otwartą sesję połączenia z serwerem i jest wpięty do pokoju, po odświeżeniu przeglądarki zobaczy okno pokoju, do którego jest wpięty

Identyfikator	WF-03
Treść	Jako użytkownik serwera czatu, chcę po wpięciu do pokoju zobaczyć ostatnie wiadomości wysłane przed moim dołączeniem, aby dowiedzieć się, co tam się obecnie dzieje.
Powiązane zasady biznesowe	ZP-08 Nowo wpięty użytkownik widzi 10 najnowszych wiadomości, które zostały wysłane do pokoju tuż przed wpięciem
Kryteria akceptacji	1. Użytkownik po wpięciu się do pokoju zobaczy 10 najnowszych wiadomości wysłanych do pokoju przed jego dołączeniem (lub mniej, jeżeli dotychczas nie wysłano do pokoju co najmniej 10 wiadomości)

Identyfikator	WF-04
Treść	Jako użytkownik serwera czatu, chcę wysłać wiadomość do pokoju w który jestem wpięty, aby zobaczyli ją inni uczestnicy dyskusji.
Powiązane zasady biznesowe	ZP-01 Pokoje to właściwe czaty - tam użytkownicy mogą wejść i pisać do siebie nawzajem
Kryteria akceptacji	1. Użytkownik wpisze tekst wiadomości w polu tekstowym u dołu czatu 2. Wiadomość wpisana w polu tekstowym zostanie wysłana po wciśnięciu klawisza „Enter", gdy aktywne będzie pole tekstowe 3. Wiadomość wpisana w polu tekstowym zostanie wysłana po naciśnięciu przycisku „Wyślij", widocznego obok pola tekstowego 4. Po wysłaniu wiadomości, pole tekstowe zostanie wyczyszczone (niezależnie od tego czy wiadomość zostanie doręczona) 5. Wiadomość wysłana do pokoju jest pokazywana wszystkim użytkownikom podpiętym do czatu u dołu strony 6. Nowa wiadomość jest pokazywana wraz z nazwą użytkownika wysyłającego u dołu konwersacji

Identyfikator	WF-05
Treść	Jako użytkownik serwera czatu, chcę chcę zobaczyć powiadomienie o wpięciu się nowego użytkownika do pokoju w którym sam jestem obecnie wpięty, aby powitać nowego dyskutanta
Powiązane zasady biznesowe	ZP-09 Serwer czatu automatycznie wysyła do pokoju wiadomości, zawierające powiadomienia o wydarzeniach związanych z pokojem, tzw. wiadomości systemowe ZP-11 Wiadomość systemowa zostaje wysłana podczas wpięcia się nowego użytkownika do pokoju
Kryteria akceptacji	1. Niezwłocznie po wpięciu się użytkownika do pokoju, serwer wyśle wiadomość systemową o treści „Użytkownik ... dołączył do pokoju", widoczną dla wszystkich użytkowników wpiętych do tego pokoju

Identyfikator	WF-06
Treść	Jako użytkownik serwera czatu, chcę zobaczyć powiadomienie o opuszczeniu pokoju przez użytkownika, aby łatwo zorientować się, że nie bierze już udziału w dyskusji.
Powiązane zasady biznesowe	ZP-09 Serwer czatu automatycznie wysyła do pokoju wiadomości, zawierające powiadomienia o wydarzeniach związanych z pokojem, tzw. wiadomości systemowe ZP-12 Wiadomość systemowa zostaje wysłana podczas wpięcia wypięcia użytkownika z pokoju ZP-13 Wiadomość systemowa zostaje wysłana, gdy użytkownik wpięty do pokoju traci połączenie z serwerem ZP-14 Wiadomość systemowa zostaje wysłana, gdy użytkownik zostaje wyrzucony z pokoju
Kryteria akceptacji	1. Niezwłocznie po wypięciu się użytkownika z pokoju, serwer wyśle wiadomość systemową, widoczną dla wszystkich użytkowników wpiętych do tego pokoju, o treści: (a) „Użytkownik ... opuścił pokój", gdy użytkownik samodzielnie wypiął się z pokoju (b) „Użytkownik ... stracił połączenie", gdy użytkownik został wypięty z pokoju na skutek przerwania sesji z uwagi na zerwanie połączenia (c) „Użytkownik ... został wyrzucony", gdy użytkownik został wypięty wskutek interwencji administratora

Identyfikator	WF-07
Treść	Jako użytkownik serwera czatu, chcę odpiąć się od pokoju, aby wpiąć się do innego pokoju.
Powiązane zasady biznesowe	ZP-06 Użytkownik może się samodzielnie wypiąć z pokoju, do którego jest wpięty
Kryteria akceptacji	1. W oknie pokoju użytkownik zobaczy przycisk lub link „Opuść pokój". 2. Po kliknięciu w „Opuść pokój", użytkownik zobaczy listę pokojów.

Identyfikator	WF-08
Treść	Jako użytkownik serwera czatu, chcę zobaczyć okno wiadomości prywatnych, aby odczytać wiadomości, które wysłano specjalnie do mnie.
Powiązane zasady biznesowe	ZW-01 Wiadomości prywatne to wiadomości, które są wysyłane do konkretnego odbiorcy, innego niż nadawca... ZW-06 Użytkownik dysponuje dodatkowym oknem, na którym widzi wiadomości prywatne. ZW-07 Z okna wiadomości prywatnych można odbierać i wysyłać wyłącznie wiadomości prywatne, których nadawcą / odbiorcą jest wybrany użytkownik.
Kryteria akceptacji	1. Po kliknięciu w link „PW", użytkownik zobaczy okno prywatnych wiadomości 2. W oknie wiadomości prywatnych, użytkownik zobaczy listę użytkowników, od których otrzymał wiadomości prywatne. 3. Po kliknięciu w link z nazwą użytkownika, użytkownik zobaczy prywatne wiadomości, których nadawcą i odbiorcą jest wskazana osoba.

Identyfikator	WF-09
Treść	Jako użytkownik serwera czatu, chcę wysłać wiadomość prywatną do jednego użytkownika, aby prowadzić z nim ciągłą konwersację.
Powiązane zasady biznesowe	
Kryteria akceptacji	1. Użytkownik wpisze tekst wiadomości w polu tekstowym u dołu okna wiadomości prywatnych 2. Wiadomość wpisana w polu tekstowym zostanie wysłana po wciśnięciu klawisza „Enter", gdy aktywne będzie pole tekstowe 3. Wiadomość wpisana w polu tekstowym zostanie wysłana po naciśnięciu przycisku „Wyślij", widocznego obok pola tekstowego 4. Po wysłaniu wiadomości, pole tekstowe zostanie wyczyszczone (niezależnie od tego czy wiadomość zostanie doręczona) 5. Wiadomość wysłana w oknie zostanie pokazana tylko użytkownikowi, z którym trwa otwarta konwersacja 6. Nowa wiadomość jest pokazywana wraz z nazwą użytkownika wysyłającego u dołu konwersacji

Identyfikator	WF-11
Treść	Jako użytkownik serwera czatu, chcę wysłać wiadomość prywatną do innego użytkownika, z którym wcześniej nie wymieniałem takich wiadomości, aby rozpocząć z nim prywatną konwersację.
Powiązane zasady biznesowe	ZW-08 W oknie wiadomości prywatnych można przeglądać wiadomości wysłane do i odebrane od jednego, wybranego użytkownika. ZW-09 W oknie wiadomości prywatnych można wysyłać wiadomości wyłącznie do nadawcy, którego wiadomości są w danym momencie pokazywane.
Kryteria akceptacji	1. Użytkownik kliknie w oknie wiadomości prywatnych w przyciski „Nowy”. 2. Użytkownik zobaczy monit o podanie nazwy użytkownika, z którym chce rozpocząć rozmowę 3. Jeżeli użytkownik jest aktywny, wówczas 4. Wiadomość wpisana w polu tekstowym zostanie wysłana po wciśnięciu klawisza „Enter”, gdy aktywne będzie pole tekstowe 5. Wiadomość wpisana w polu tekstowym zostanie wysłana po naciśnięciu przycisku „Wyślij”, widocznego obok pola tekstowego 6. Po wysłaniu wiadomości, pole tekstowe zostanie wyczyszczone (niezależnie od tego czy wiadomość zostanie doręczona) 7. Wiadomość wysłana w oknie zostanie pokazana tylko użytkownikowi, z którym trwa otwarta konwersacja 8. Nowa wiadomość jest pokazywana wraz z nazwą użytkownika wysyłającego u dołu konwersacji

Identyfikator	WF-12
Treść	Jako administrator, chcę utworzyć nowy pokój, aby umożliwić użytkownikom konwersację w węższym gronie.
Powiązane zasady biznesowe	ZU-12 Administratorzy mają prawo tworzyć i usuwać pokoje.
Kryteria akceptacji	1. Administrator kliknie w oknie z listą pokojów w przycisk „Nowy”. 2. Administrator zobaczy monit o podanie nazwy nowego pokoju. 3. Administrator po podaniu nazwy i zaakceptowaniu, zostanie przeniesiony do listy pokojów, na której będzie widoczna nazwa dodanego pokoju. 4. Administrator i inni użytkownicy mogą wpiąć się do nowoutworzonego pokoju.

Identyfikator	WF-13
Treść	Jako administrator, chcę usunąć zbędny pokój, aby utrzymać porządek na swoim serwerze czatu.
Powiązane zasady biznesowe	ZU-12 Administratorzy mają prawo tworzyć i usuwać pokoje.
Kryteria akceptacji	1. Administrator wejdzie do pokoju, który chce usunąć. 2. Administrator zobaczy obok tytułu z nazwą pokoju przycisk „Usuń”. 3. Administrator po kliknięciu przycisku zobaczy monit z potwierdzeniem działania. 4. Administrator potwierdzi decyzję w monicie. 5. Po potwierdzeniu decyzji o usunięciu, administrator zostanie przeniesiony do listy pokojów, na której nie będzie już widniała nazwa usuniętego pokoju. 6. Pozostali użytkownicy w usuniętym pokoju zostaną niezwłocznie od niego odpięci i zobaczą monit systemowy informujący o usunięciu pokoju.

Identyfikator	WF-14
Treść	Jako administrator, chcę usunąć użytkownika z pokoju, aby utrzymać należyty poziom konwersacji.
Powiązane zasady biznesowe	ZU-14 Administratorzy mają prawo wyrzucać użytkowników z pokojów.
Kryteria akceptacji	1. Administrator wejdzie do pokoju. 2. Administrator najedzie na nazwę użytkownika którego chce usunąć z pokoju i kliknie na przycisk z nazwą „Usuń z pokoju”. 3. Administrator po kliknięciu przycisku zobaczy monit z potwierdzeniem działania. 4. Administrator potwierdzi decyzję w monicie. 5. Po potwierdzeniu decyzji o usunięciu, administrator (tak samo jak każdy inny użytkownik podpięty do pokoju) zobaczy wiadomość systemową o usunięciu z konwersacji. 6. Usunięty użytkownik zostanie niezwłocznie wypięty z pokoju, a także zobaczy monit o przyczynie wypięcia.

Identyfikator	WF-15
Treść	Jako administrator, chcę usunąć użytkownika z serwera, aby ukarać go za łamanie zasad netykiety.
Powiązane zasady biznesowe	ZU-15 Administratorzy mają prawo wyrzucać użytkowników z serwera.
Kryteria akceptacji	1. Administrator wejdzie do pokoju. 2. Administrator najedzie na nazwę użytkownika którego chce usunąć z pokoju i kliknie na przycisk z nazwą „Usuń z serwera”. 3. Administrator po kliknięciu przycisku zobaczy monit z potwierdzeniem działania. 4. Administrator potwierdzi decyzję w monicie. 5. Po potwierdzeniu decyzji o usunięciu, administrator (tak samo jak każdy inny użytkownik podpięty do pokoju) zobaczy wiadomość systemową o usunięciu z serwera. 6. Usunięty użytkownik zostanie niezwłocznie wypięty z pokoju i jego sesja zostanie zakończona, a także pokazany zostanie monit o przyczynie tych zdarzeń (usunięcie z serwera czatu).

Identyfikator	WF-16
Treść	Jako administrator, chcę zmienić swoje hasło, aby zabezpieczyć swoje hasło w razie ujawnienia go osobie niepowołanej, bez zmiany plików konfiguracyjnych i restartowania całego serwera.
Powiązane zasady biznesowe	ZU-18 Administratorzy mają prawo zmieniać swoje hasła użytkowników.
Kryteria akceptacji	1. Administrator wejdzie na kartę „Moje konto”. 2. Administrator kliknie na przycisk „Zmień hasło”, widoczny pod nazwą użytkownika. 3. Administrator zobaczy monit zmiany hasła, zawierający jedno pole tekstowe na stare hasło i dwa na nowe hasło (wszystkie trzy ukryte przed podglądaniem treści podczas ich wprowadzania). 4. Administrator potwierdzi decyzję o zmianie hasła w monicie. 5. Po potwierdzeniu decyzji, administrator zobaczy wiadomość systemową o zmianie hasła. 6. Administrator rozłączy się z serwerem. 7. Administrator spróbuje rozpocząć nową sesję z serwerem, autoryzując się nowym hasłem. 8. Nowe hasło zostanie zaakceptowane przez serwer, sesja zostanie rozpoczęta prawidłowo.

7.4.2 Wymagania niefunkcjonalne

Identyfikator	HN-01
Treść	Długość nazwy użytkownika jest ograniczona od 2 do 32 znaków alfanumerycznych, w celu uniknięcia problemów z identyfikacją użytkownika na serwerze.
Powiązane zasady biznesowe	ZU-03 Nazwa użytkownika to ciąg od 2 do 32 alfanumerycznych znaków.
Kryteria akceptacji	1. Po wpisaniu do pola użytkownika nazwy krótszej niż 2 znaki, dłużej niż 32 znaki lub zawierającej inne znaki niż alfanumeryczne, zwracany jest błąd.

Identyfikator	HN-02
Treść	Loginy i hasła administratorów są gromadzone w plikach konfiguracyjnych, a po uruchomieniu serwera – w jego pamięci operacyjnej.
Powiązane zasady biznesowe	ZU-07 Konta administratorów są utrzymywane na serwerze w postaci par wartości: nazwa użytkownika i hasło.
Kryteria akceptacji	1. Serwer jest wyposażony w pliki konfiguracyjne 2. Po załadowaniu serwera, z plików konfiguracyjnych są odczytywane dane kont administracyjnych 3. Serwer po uruchomieniu jest wyposażony w konta o nazwach i hasłach zgodnych z wpisami w plikach konfiguracyjnych.

Identyfikator	HN-03
Treść	Loginy administratorów w oknach czatu są pogrubione i pokolorowane na czerwono.
Powiązane zasady biznesowe	ZU-10 W okienkach czatu, loginy administratorów są pogrubione i pokolorowane na czerwono.
Kryteria akceptacji	1. Nazwy administratorów w oknach czatu są pogrubione i pokolorowane na czerowono.

Identyfikator	HN-04
Treść	Nazwy pokojów mają od 3 do 32 znaków alfanumerycznych długości, przy czym nigdy dwa pokoje nie mają tej samej nazwy.
Powiązane zasady biznesowe	ZP-02 Każdy pokój ma unikalną nazwę będącą ciągiem alfanumerycznym od 3 do 32 znaków.
Kryteria akceptacji	1. Nie jest możlwe utworzenie pokoju o nazwie, która już wcześniej się pojawiała 2. Nie jest możliwe utworzenie pokoju o nazwie krótszej niż 3 znaki i dłuższej niż 32 znaki. 3. Nie jest możliwe utworzenie pokoju o nazwie zawierającej znaki inne niż litery alfabetu łacińskiego, cyfry i znak podkreślenia.

Identyfikator	HN-05
Treść	Wiadomości prywatne są czyszczone niezwłocznie po rozłączeniu się przez dowolnego z rozmówców.
Powiązane zasady biznesowe	ZW-10 Wiadomości prywatne są utrzymywane dopóki nadawca i odbiorca mają aktywną sesję na serwerze.
Kryteria akceptacji	1. Po zamknięciu sesji użytkownika, wiadomości prywatne których był nadawcą lub odbiorcą ulegają usunięciu.

Identyfikator	HN-06
Treść	Bufor pokoju wiadomości prywatnych zawiera do 100 wiadomości.
Powiązane zasady biznesowe	ZW-11 Dla każdej pary użytkowników, na serwerze jest gromadzone co najwyżej 100 wiadomości prywatnych.
Kryteria akceptacji	1. Po przekroczeniu liczby 100 wiadomości prywatnych w pokoju, bufor ulega „zawinięciu", usuwając najstarsze wiadomości.

Identyfikator	HN-07
Treść	Bufor pokoju niebędącego dedykowanym do wiadomości prywatnych zawiera do 10 wiadomości.
Powiązane zasady biznesowe	ZP-08 Nowo wpięty użytkownik widzi 10 najnowszych wiadomości, które zostały wysłane do pokoju tuż przed wpięciem
Kryteria akceptacji	1. Po przekroczeniu liczby 10 wiadomości w pokoju, bufor ulega „zawinięciu", usuwając najstarsze wiadomości.

7.4.3 Wymagania na środowisko docelowe

Identyfikator	WS-01
Treść	W rozwiązaniu należy wykorzystać środowisko Viua VM i język ViuAct

Identyfikator	WS-02
Treść	Czat będzie użytkowany jako aplikacja webowa typu *single page application*

Identyfikator	WS-03
Treść	Aplikacja czatu będzie dostosowana przede wszystkim do obsługi z wykorzystaniem urządzeń mobilnych.

7.4.4 Wymagania dotyczące procesu wytwarzania

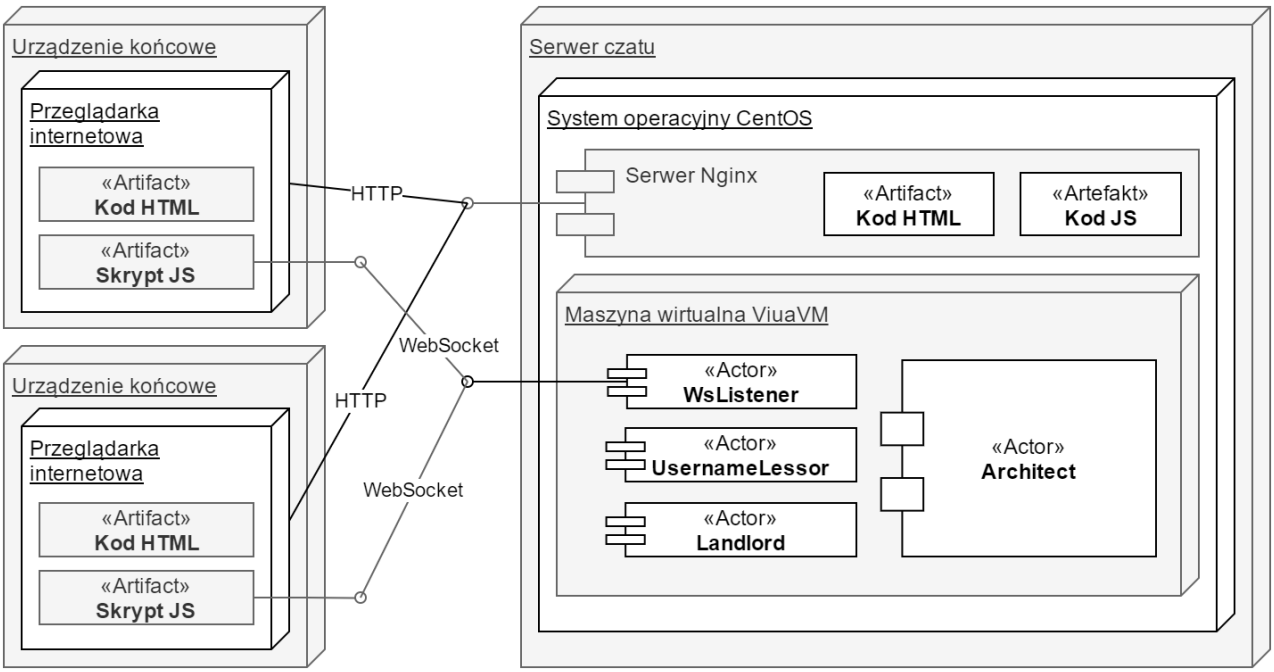
Identyfikator	WW-01
Treść	W procesie wytwórczym należy korzystać z metodologii mini-Scrum

Identyfikator	WW-02
Treść	W procesie wytwórczym należy uprzednio przygotować specyfikację przypadków użycia – co wynika z wymagań uczelni.

Rozdział 8

Program ViuaChat – implementacja

8.1 Architektura systemu



Rysunek 8.1: Diagram rozmieszczenia elementów architektury systemu ViuaChat

Architektura systemu opiera się na klasycznym modelu klient-serwer.

Klienci będą uzyskiwali dostęp do usługi czatu za pośrednictwem przeglądarki internetowej. Łącząc się z podanym adresem (IP lub domeny), na którym połączeń nasłuchuje serwer, w pierwszej kolejności przeglądarka będzie próbować połączyć się z nim przy użyciu protokołu http i standardowego portu 80, wysyłając do niego żądanie metodą GET. Wówczas, serwer będzie zawsze odpowiadał statycznym plikiem HTML, zawierającym odwołania do skryptów w języku JavaScript (JS) oraz pozostałej, statycznej treści (np. grafiki czy arkusze stylów CSS). Serwer będzie odsyłał te pliki do przeglądarki w odpowiedzi na kolejne żądania HTTP GET, wysyłane w miarę dalszego renderowania pliku HTML. W ten sposób, po stronie klienta zostanie pobrana i uruchomiona aplikacja internetowa typu Single Page Application, której interfejs będzie reagował z użytkownikiem oraz ulegał zmianom wskutek działania skryptów JS, załadowanych na pierwszym etapie uruchomienia. Po stronie serwera, dostarczaniem treści statycznych będzie zajmował się daemon HTTP - Nginx.

Gdy tylko skrypty JS wykryją pobranie wszystkich plików składowych aplikacji z serwera, podjęta zostanie próba nawiązania połączenia z tym serwerem przy użyciu protokołu WebSocket. Będzie on od tego momentu podstawowym kanałem komunikacji pomiędzy klientem a serwerem.

Zgodnie ze standardem WebSocketu, zanim zostanie nawiązane właściwe połączenie, powinno dojść

do „uścisku dłoni” (ang. *handshake*) pomiędzy klientem a serwerem. W związku z tym, pierwsza próba połączenia również zostanie podjęta przy użyciu protokołu http, jednakże tym razem pod innym, dedykowanym portem (w naszym przypadku będzie to port 8000), a także zawierać nagłówki wskazujące na żądanie zmiany używanego protokołu na WebSocket, jego wersję oraz klucz („Sec-WebSocketKey”). Serwer udzieli wówczas odpowiedzi ze swoim własnym kluczem, informując o zmianie stosowanego protokołu na WebSocket.

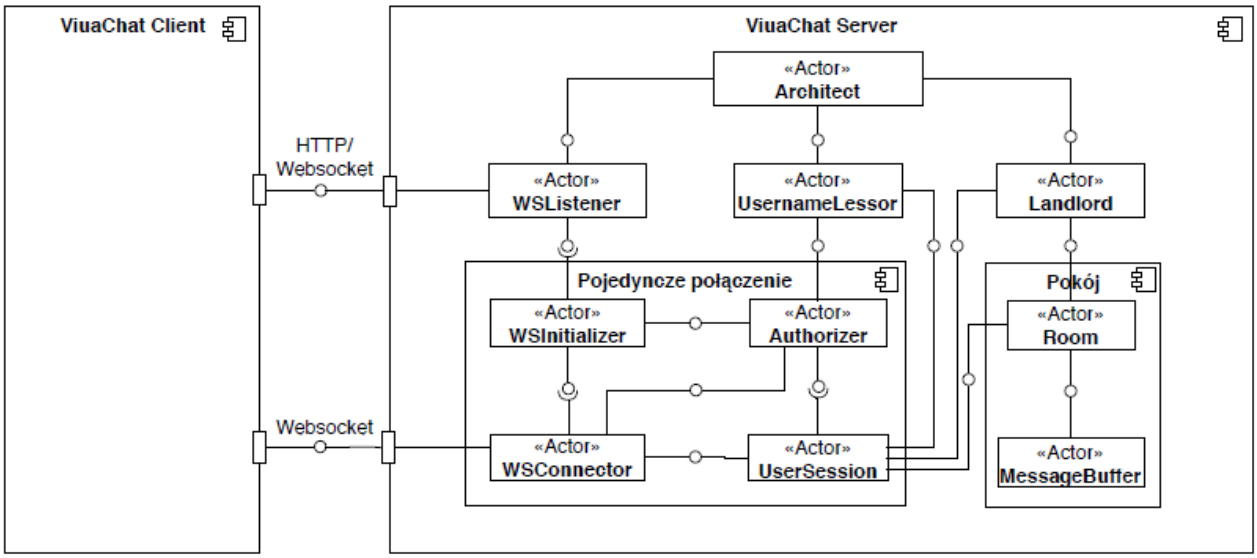
W chwili prawidłowego rozpoczęcia połączenia WebSocket, aplikacja po stronie klienta wyświetli użytkownikowi okno autoryzacyjne. Wpisane tam dane zostaną następnie przesłane do serwera. Po jego stronie, komunikat zostanie zdekodowany przez aktora `WSConnector` i przekazany powiązanemu aktorowi `Authorizer`. Jego zadaniem będzie weryfikacja przedstawionych informacji oraz podjęcie decyzji o autoryzacji lub jej odmowie. Decyzja ta jest odsyłana do `WSConnectora` i następnie przekazywana do aplikacji po stronie klienta.

Jezeli autentykacja przebiegnie pomyślnie, aktor `Authorizer` uruchamia aktora `UserSession`, spina go z aktorem `WSConnector` używanym wcześniej do komunikacji z frontendem, oraz ulega autodestrukcji.

8.1.1 Dekompozycja systemu na podsystemy

8.1.1.1 Warstwa serwera – backend

Na serwer czatu składa się grupa współdziałających, ale zupełnie odrębnych od siebie aktorów.



Rysunek 8.2: Diagram komponentów serwera ViuaChat

`Architect` Uruchamiany jako pierwszy wraz z całym serwerem, a następnie inicjalizuje i nadzoruje aktorów `WSListener`, `UsernameLessor` oraz `Landlord`. W razie nieprawidłwego działania lub wyłączenia któregokolwiek z tych trzech głównych aktorów, `Architect` automatycznie zainicjuje przeładowanie całego serwera. Ponadto, `Architect` potrafi w momencie uruchamiania serwera odczytać jego pliki konfiguracyjne i na tej podstawie należycie skonfigurować pokoje oraz administratorów. Zawsze występuje w jednym egzemplarzu.

`WSListener` Odpowiada za nasłuchiwanie na porcie 8000 po stronie serwera, a przy każdej próbie połączenia będzie tworzyć kolejnego, niezależnego aktora `WSInitializer`. Występuje zawsze w jednym egzemplarzu.

`WSInitializer` Odpowiada za realizację „uścisku dłoni” i formułowanie odpowiedzi zwrotnej po stronie serwera w odniesieniu do połączenia na konkretnym gnieździe. W razie prawidłowego nawiązania połączenia, aktor ten utworzy kolejną parę aktorów, `WSConnector` oraz `Authorizer`, zaś WSInitializer ulegnie samozniszczeniu.

`WSConnector` Odpowiada za dalszą, bezpośrednią obsługę przydzielonego gniazda. Występuje ich tylu, ile jest otwartych połącznień. Jego rolą będzie również kodowanie i dekodowanie wiadomości (ang. „messages”), czyli podstawowych logicznych jednostek informacji, które są używane przy połączeniach z użyciem protokołu WebSocket. Pilnuje również, czy połączenie nie zostało zerwane oraz inicjuje zamykanie sesji użytkownika.

`Authorizer` Wymienia wiadomości od `WSConnectora`, który został uruchomiony wraz z nim i odpowiada za należytą autentykację i/lub autoryzację użytkownika w usłudze czatu. Egzemplarz aktora tego typu jest powoływany dla każdego otwartego połączenia bez nawiązanej sesji. Aby dokonać autoryzacji, aktor `Authorizer` kontaktuje się `UsernameLessor`. Jeżeli autentykacja przebiegnie pomyślnie, aktor `Authorizer` uruchamia aktora `UserSession`, spina go z aktorem `WSConnector` używanym wcześniej do komunikacji z frontendem, oraz ulega autodestrukcji.

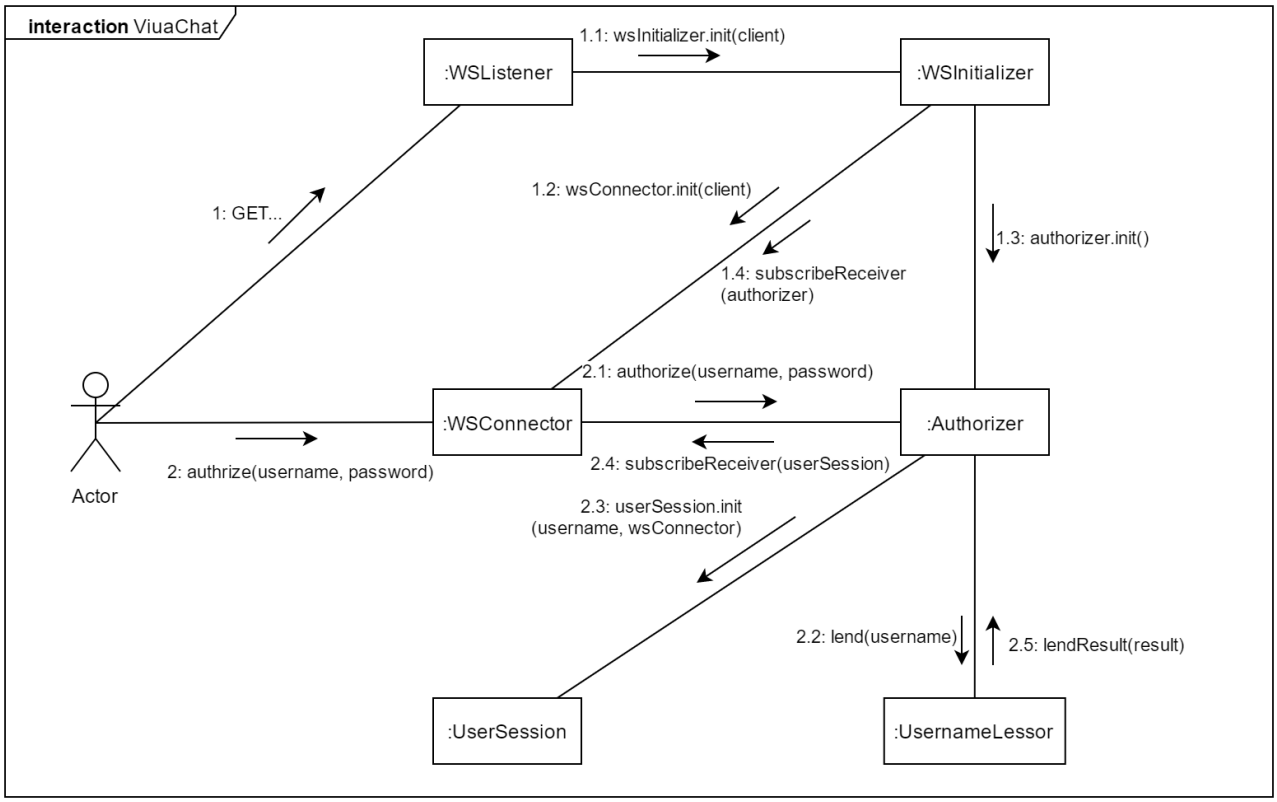
`UsernameLessor` Jego zadaniem jest zarządzanie informacjami na temat tymczasowych nazw użytkowników, należących do użytkowników bez stałych kont (ich gromadzenie, udzielanie, weryfikacja, dbanie o unikalność), a także weryfikacja tożsamości kont administratorów z dodatkowym użyciem hasła. Występuje w jednym egzemplarzu, przez cały czas istnienia serwera. Ponadto, nadzoruje działanie aktorów `Authorizer` oraz `UserSession`.

`UserSession` Przejmuje komunikację z użyciem `WSConnector`, pozwalając na zwyczajne użytkowanie czatu. Aktor „UserSession" gromadzi informacje na temat nazwy oraz poziomu uprawnień użytkownika, a także tego, z jakim pokojem jest obecnie spięty.

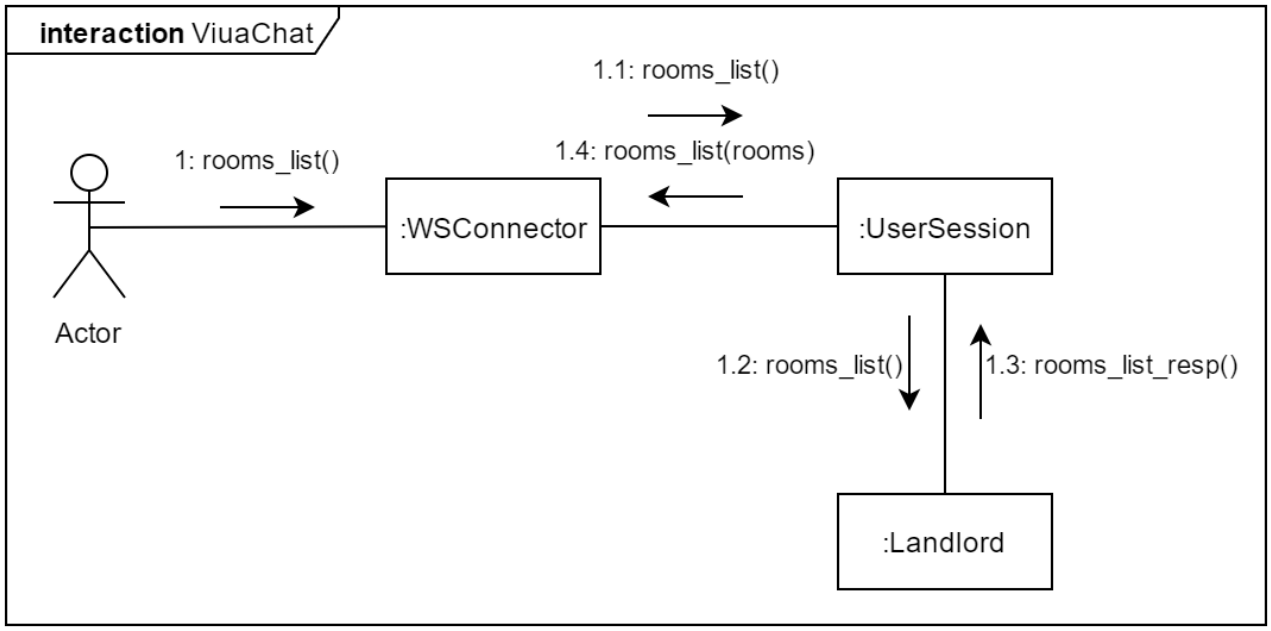
`Landlord` Jego zadaniem jest współudział w podpinaniu użytkowników do pokoju, tworzeniem nowych i usuwaniem istniejących pokojów, a także utrzymywanie i udostępnianie kompletnej listy aktywnych pokojów.

`Room` Działa jak router wiadomości i przechowuje listę użytkowników którzy są do niego wpięci. Istnieje w tylu egzemplarzach, ile jest aktywnych pokojów. Od strony backendu, jedynym odróżnieniem pomiędzy aktorami `Room` reprezentującymi klasyczny, ogólnodostępny pokój, a reprezentującymi jedynie wymianę prywatnych wiadomości pomiędzy określoną parą użytkowników, jest jego nazwa. Nazwy pokojów z wiadomościami prywatnymi zawsze zawierają znak krzyżyka (#), rozdzielający nazwy użytkowników biorących udział w konwersacji. Pokoje ogólnodostępne nigdy nie będą miały w nazwie tego znaku.

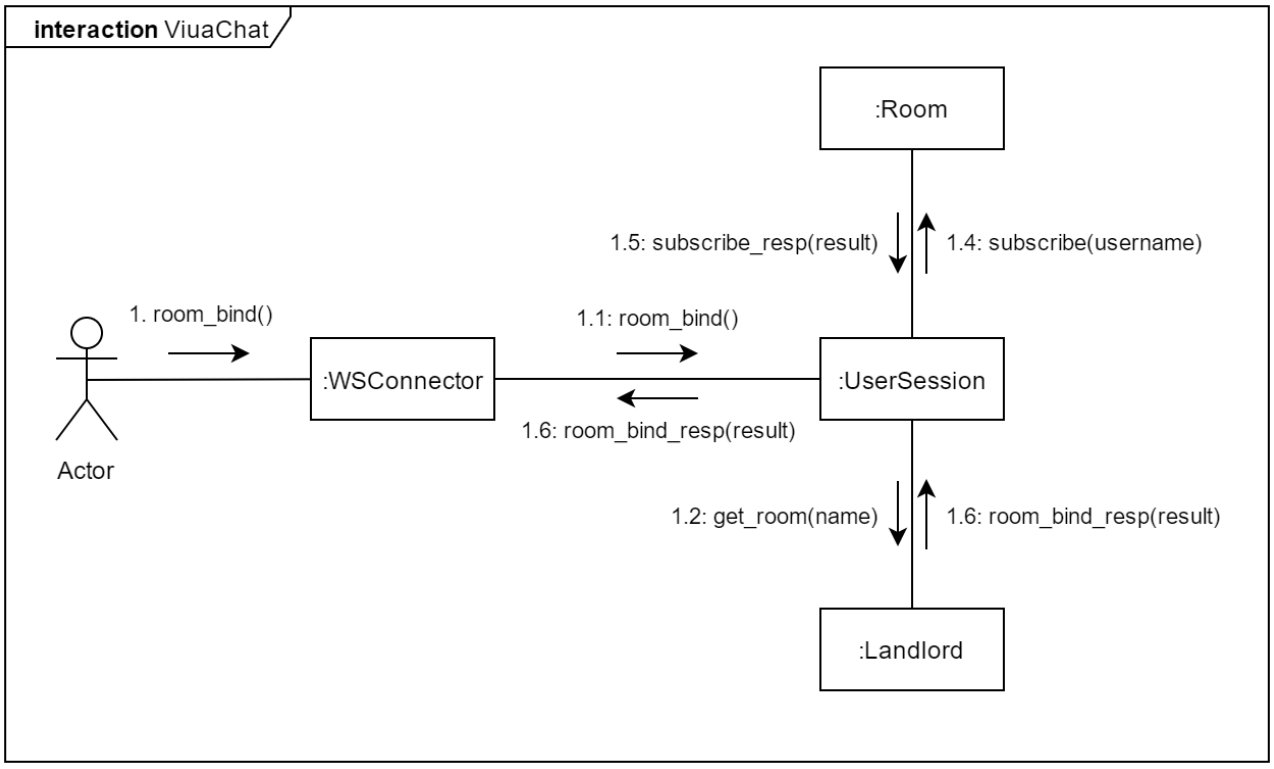
`MessageBuffer` Przechowuje i odtwarza 10 najnowszych wiadomości wysłanych do pokoju. Występuje po jednym egzemplarzu dla każdego aktywnego aktora `Room`.



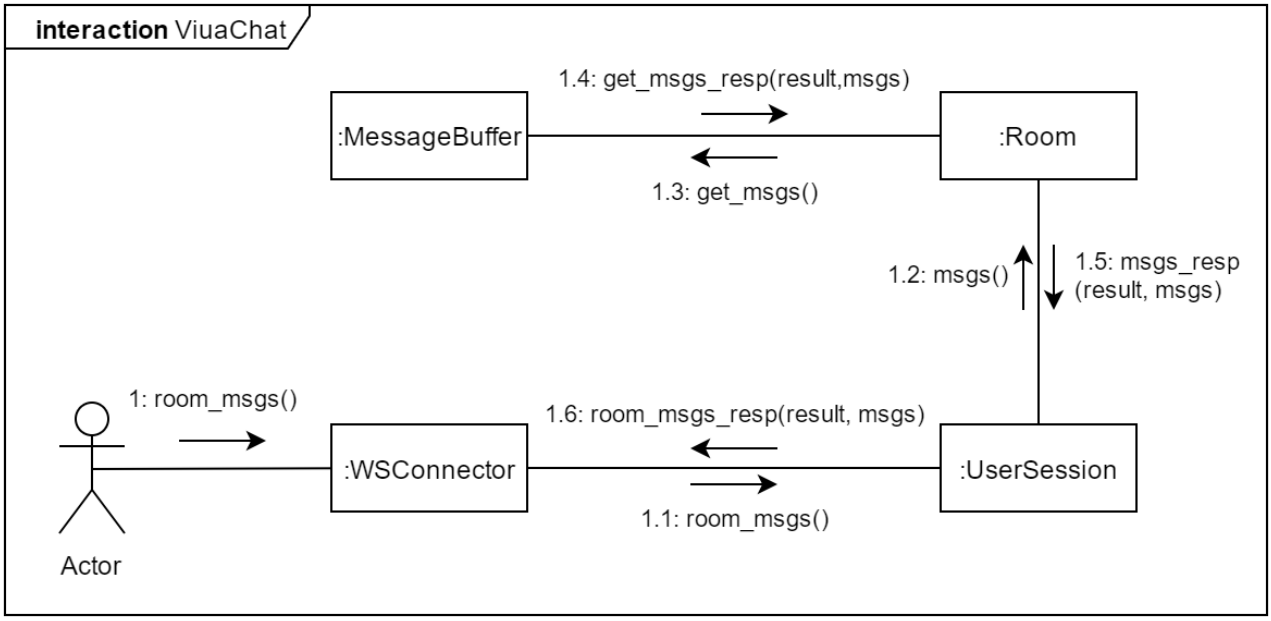
Rysunek 8.3: Diagram komunikacji pomiędzy aktorami w czasie nawiązywania połączenia



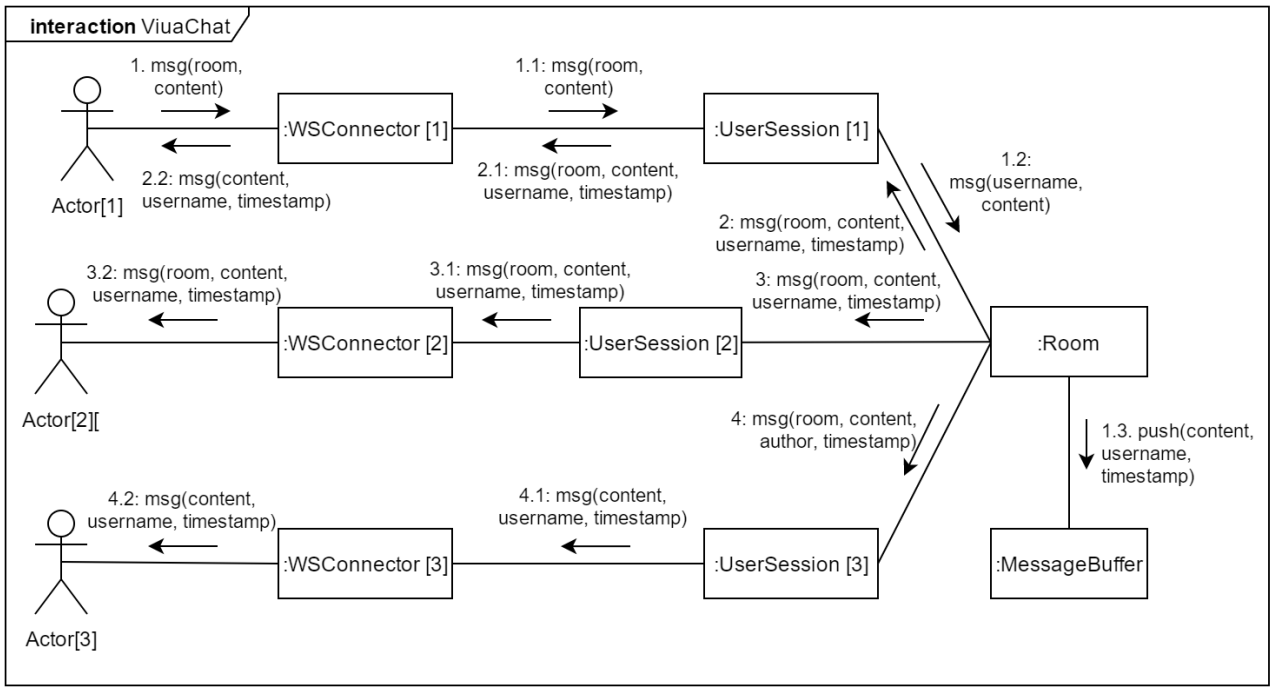
Rysunek 8.4: Diagram komunikacji pomiędzy aktorami w czasie pobierania listy pokojów



Rysunek 8.5: Diagram komunikacji pomiędzy aktorami w ramach podpinania do pokoju



Rysunek 8.6: Diagram komunikacji pomiędzy aktorami podczas pobierania listy ostatnich wiadomości



Rysunek 8.7: Diagram komunikacji pomiędzy aktorami w czasie wysyłania wiadomości do pokoju (ogólnodostępnego)

8.1.1.2 Warstwa interfejsu użytkownika – frontend

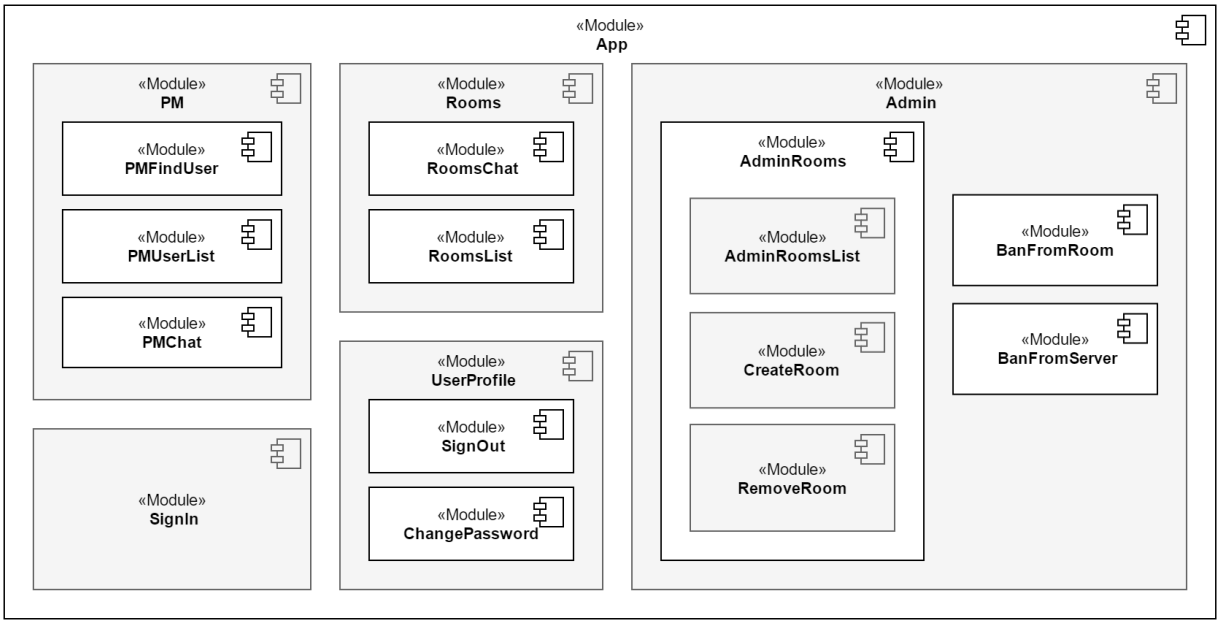
Podczas pracy nad wartwą frontendu, zastosowano framework webowy Vue.js. Jedną z przyczyn dla tej decyzji jest możliwość zdekomponowania projektowanej aplikacji na mniejsze części, nazywane modułami (ang. *modules*). Są one zorganizowane hierarchiczne. Każdy z modułów zawiera własny skrypt JavaScript, a także kod HTML i arkusz CSS. Powoduje to, że każdy z modułów jest niezależny od pozostałych i może realizować swoje zadania w pełni autonomicznie.

Każdy z modułów udostępnia swojemu rodzicowi pewne określone parametry. Ich zmiana jest podstawowym sposobem na interakcję pomiędzy nimi, co jest zgodne z paradygmatem *data driven application* (z ang. „aplikacja sterowana poprzez dane"). W podobny sposób następuje zmiana kodu HTML modułów. Zamiast zmieniać węzły DOM w sposób jawny poprzez skrypt, programista wskazuje w szablonie HTML te miejsca, które ulegają określonym przemianom wraz ze zmian wewnętrznych parametrów modułu.

Podstawowa struktura warstwy frontendowej przewiduje podział na pięć części:

- Część autoryzacyjna – służy do nawiązania połączenia z serwerem i rozpoczęcie sesji użytkownika.
- Pokoje (publiczne) – dostarcza listę ogólnodostępnych pokojów, umożliwia podpięcie się wybranych z nich oraz prowadzenie rozmów z innymi użytkownikami, którzy się do nich podpięli.
- Wiadomości prywatne – umożliwia nawiązywanie prywatnych rozmów pomiędzy użytkownikami oraz wymianę wiadomości prywatnych.
- Narzędzia administracyjne – pozwalają administratorom na dodawanie i usuwanie pokojów, wyrzucanie użytkowników z pokojów i z serwera.
- Profil użytkownika – umożliwia podejrzenie informacji na temat własnego konta, zmianę przez administratora swojego hasła użytkownika oraz rozłączenie się z serwerem.

W projekcie przewidziano zastosowanie modułów, przedstawionych na rysunku 8.8.



Rysunek 8.8: Diagram komponentów aplikacji webowej ViuaChat

`App` Nadrzędny moduł, istniejący przez cały czas użycia instancji aplikacji. Obejmuje najbardziej zewnętrzne struktury HTML, przechowuje podstawowy stan aplikacji (stan autoryzacji, nazwę użytkownika, poziom uprawnień, gniazdo WebSocket dla połączeń z

wartwą backendu), a także zapewnia routing do głównych części aplikacji - czyli modułu logowania, modułu pokojów (publicznych), pokojów prywatnych wiadomości, profilu użytkownika i narzędzi administracyjnych.

`SignIn` Moduł odpowiadający za początkową autoryzację do serwera (potocznie znane jako „logowanie").

`Rooms` Moduł reprezentuje część aplikacji poświęconą ogólnodostępnym pokojom dla wspólnych konwersacji. Podlegają mu:

- `RoomsChat` Moduł odpowiedzialny *stricte* za okno czatu - wyświetlanie konwersacji oraz wysyłanie wiadomości do pokoju.
- `RoomsList` Moduł wyświetlający listę pokojów, a także umożliwiający podpięcie się do wybranego z nich.

`PM` Moduł odwzorowujący całą część aplikacji poświęconą wymianie pomiędzy użytkownikami wiadomości prywatnych. W jego skład wchodzą:

- `PMFindUser` Moduł, którego zadaniem jest odnalezienie użytkowika, do którego ma zostać wysłana wiadomość prywatna
- `PMUserList` Moduł obsługujący listę użytkowników, którzy wcześniej otrzymali lub wysłali wiadomości prywatne.
- `PMChat` Moduł odpowiedzialny za obsługę zasadniczego okna czatu wiadomości prywatnych, pozwalającego wymieniać wiadomości prywatne z jednym, wybranym użytkownikem.

`UserProfile` Moduł pozwalający na podstawową kontrolę własnego profilu, w tym:

- `SignOut` Moduł odpowiedzialny za zamknięcie sesji użytkownika, rozłączenie z serwerem oraz powrót aplikacji do stanu sprzed autoryzacji
- `ChangePassword` Moduł mający umożliwiać administratorom zmianę swoich własnych haseł

`Admin` Moduł obejmujący pod sobą wszystkie narzędzia administracyjne. W jego skład wchodzą:

- `AdminRooms` Moduł do zarządzania ogólnodostępnymi pokojami czatu, w tym:
  - `AdminRoomsList` Moduł listy wszystkich ogólnodostępnych pokojów z narzędziami do ich edycji
  - `CreateRoom` Moduł pozwalający na dodawanie pokojów do serweru czatu
  - `RemoveRoom` Moduł pozwalający na usuwanie istniejących pokojów z serwera
- `BanFromRoom` Moduł służący do wyrzucania użytkowników z pokojów, w których obecnie się znajdują.
- `BanFromServer` Moduł pozwalający na wyrzucenie użytkowników z serwera.

8.2 Decyzje projektowe

8.2.1 Środowisko docelowe

Decyzje związane ze środowiskiem docelowym zostały zdeterminowane w głównej mierze przez wymagnia WS-01, WS-02 i WS-03. Punktem wyjścia była konieczność uruchomienia serwera czatu (backendu) na maszynie Viua VM. W związku z tym, z kręgu zaintersowań wypadły systemy operacyjne z

rodziny Windows, z uwagi na ich niekompatybilność z Viua VM. Wybór padł na system operacyjny Linux Mint, ponieważ jest dobrze utrzymywaną platformą opartą o Debiana. Ważkim argumentem była łatwość obsługi tej dystrybucji przez nowych użytkowników, którzy nie byli zaznajomieni wcześniej z rozwiązaniami linuksowymi.

W dalszej kolejności należało podjąć decyzję, w jaki sposób serwować pliki związane z warstwą frontendu. Propozycja użycia serwera Nginx została uznana za słuszną. Zespół przekonała prostota tego rozwiązania. Równocześnie, serwer Nginx był serwerem wypróbowanym i rozwijanym od wielu lat. Uznano zatem, że będzie użyteczny również w środowisku produkcyjnym.

Na cele obrony pracy dyplomowej, przygotowano i uruchomiono serwer wirtualny u dostawcy usług chmurowych, aby zapewnić niezależność od infrastruktury na miejscu prezentacji i zapewnić jej należyty poziom niezawodności. Nie jest on jedak niezbędny do uruchomienia aplikacji przez potencjalnego, przyszłego użytkownika platformy ViuAct.

Aby zademonstrować możliwości wielowątkowości w praktyce, urządzenie docelowe, na którym przyszły użytkownik będzie uruchamiał serwer czatu, powinno dysponować wielordzeniowym procesorem. Nie ma tu znaczenia, w jakiej architekturze zrealizowano określony procesor. Za uruchamianie poszczególnych wątków odpowiada bowiem Viua VM.

Urządzenia, które mają zaprezentować funkcjonowanie systemu w wykorzystaniu przez użytkowników końcowych, to komputery przenośne (laptopy), jeden z systemem operacyjnym Windows 10, a drugi – z zainstalowaną dystrybucją Linux Mint, o której wspomniano wcześniej w kontekście środowiska wytwórczego. Zaplanowano również wykorzysanie dwóch urządzeń mobilnych. Jedno z nich jest tabletem z rodziny Samsung Galaxy, a drugim – smartfon LG K11. Oba z nich są wyposażone w dystrybucje systemu operacyjnego Android oraz przeglądarkę Google Chrome. Dodanie urządzeń mobilnych do zestawu demonstracyjnego ma przede wszystkim uczynić zadość wymaganiu WS-03.

8.2.2 Środowisko implementacji

Pierwszym kryterium przy doborze składników środowiska implementacyjnego była spójność z rozwiązaniami produkcyjnymi. Miało to pozwolić uniknąć problemów z przenośnością. Na środowisko implementacyjne wybrano laptop z takim samym systemem operacyjnym, w jakim ma być uruchomiona ViuaVM wraz z oprogramowaniem serwera czatu. Szczegółowo, warstwę sprzętowo opisano w 7.2.4 Istniejąca infrastruktura.

Jak już wspomniano, nie jest możliwe skompilowanie i uruchomienie Viua VM w systemach Microsoft Windows. W związku z tym, początkowo serwer był uruchamiany po systemem operacyjnym Linux Mint, zainstalowanym na maszynie wirtualnej uruchomionej w środowisku Oracle VirtualBox. Gdy pierwsze testy uruchomieniowe zostały zakończone z powodzeniem, pracę nad projektem przeniesiono na komputer z natywnie zainstanowanym systemem Linux Mint.

Równolegle z testami współpracy Viua VM ze środowiskiem Linux Mint, Marek Marecki przygotowywał skrypt powłoki Bash oraz zestaw repozytoriów Git z podstawowymi komponentami platformy ViuAct (czyli maszyną Viua VM, kompilatorem ViuAct i kolekcją bibliotek zewnętrznych). Tak przygotowany zestaw mógł zostać pobrany i skompilowany z użyciem jednej komendy, co znacznie ułatwiło prace nad czatem.

8.3 Projekt algorytmów i przyjętych protokołów

8.3.1 Protokół frontend-backend

Komunikacja pomiędzy frontendem a backendem będzie odbywać się poprzez wymianę krótkich komunikatów tekstowych, opisujących struktury danych w standardzie JSON. Przyjęto, że komunikaty dzielą się na dwie grupy:

1. **Synchroniczne** – są wysyłane zawsze od klienta do serwera, a serwer zawsze odpowiada na nie klientowi, podając informację na temat powodzenia żądanej operacji lub też przesyłając oczekiwane dane.

2. **Asynchroniczne** – mogą być wysyłane z inicjatywy dowolnej ze stron i nie są związane z obowiązkiem jakiejkolwiek odpowiedzi.

8.3.1.1 Budowa komunikatów synchronicznych

Prawidłowe komunikaty synchroniczne, wysyłane przez klienta do serwera, mają następującą budowę:

```
{
  'fn': 'funkcja',
  'arg':
  {
    'argument1' : 123,
    'argument2' : 'abc'
  },
  'seq': 1,
              'session_key': 'abcdef123456'
}
```

Kolejne własności obiektu, wyrażonego w notacji JSON:

- `fn` – łańcuch znaków, który wskazuje, jakiej funkcjonalności (operacji) dotyczy wysłany komunikat (np. `auth` przy autoryzacji lub `bind_room` przy podpinaniu użytkownika do pokoju);
- `arg` – obiekt, który przenosi dodatkowe argumenty niezbędne do realizacji żądanej operacji (np. np. login i hasło przy komunikacie autoryzacji na serwer lub nazwę pokoju przy komunikacie podpięcia); obiekt ten może zawierać jedynie parametry z typami prostymi, takimi jak łańcuchy znaków czy liczby całkowite; parametr `arg` jest obowiązkowy ale zawarta w nim struktura może być pusta `//`;
- `seq` – numer kolejny komunikatu, generowany po stronie klienta, który pozwoli mu potem zorientowac się, na jaki komunikat została przesłana konkretna odpowiedź serwera;
- `session_key` – łańcuch znaków, będący kluczem sesji, przy użyciu którego użytkownik autoryzuje wiadomość; klucz jest uzyskiwany przy autoryzacji do serwera; parametr `session_key` nie jest obowiązkowy dla wiadomości typu `auth` przed uzyskaniem pozytywnej autoryzacjji;

Budowa komunikatu, będącego odpowiedzią serwera na komunikat synchroniczny, zależy od powodzenia żądanej operacji. W razie sukcesu wygląda następująco:

```
{
  'seq': 1,
  'result': 1,
  'par':
  {
    'parametr1': 123,
    'parametr2': 'abc'
  }
}
```

Z kolei, w przypadku błędu po stronie serwera, komunikat wygląda następująco:

```
{
  'seq': 1,
  'result': 0,
  'par': {},
  'error_code': 101,
  'error_name': 'Brak uprawnien'
}
```

Poniżej wskazano, jakie typy i znaczenia mają kolejne własności obiektu w notacji JSON:

- `seq` – liczba całkowita, przepisana z wiadomości od klienta, pozwalająca zorientować się, jakiej komendy dotyczy wysłany komunikat.
- `result` – licba całkowita 1 lub 0, która wskazuje, czy żądana operacja została zakończona powodzeniem (1 oznacza sukces, a 0 wskazuje na błąd);
- `par` – obiekt, który przenosi dodatkowe parametry z danymi, których mógł żądać użytkownik (np. listę nazw pokojów podczas logowania); dokładna struktura obiektu zależy od tego, jakiego typu była żądana operacja; parametr `par` jest obowiązkowy przy odpowiedziach, ale zawarty w nim obiekt może być pusty `{}`; schmeat
- `error_code` – liczba całkowita będąca kodem identyfikującym błąd, który wystąpił po stronie serwera podczas wykonywania żądanej operacji; parametr pojawia się wyłącznie wówczas, gdy wykonanie operacji zakończyło się niepowodzeniem;
- `error_name` – łańcuch znaków opisujący błąd, który wystąpił po stronie serwera podczas wykonywania żądanej operacji; parametr pojawia się wyłącznie wówczas, gdy wykonanie operacji zakończyło się niepowodzeniem;

8.3.1.2 Typy komunikatów synchronicznych

Lp	**Typ (`fn`)**	**Nazwa funkcjonalności (operacji)**	**Oczekiwane argumenty (`arg`)**	**Oczekiwane parametry odpowiedzi (`par`)**
1	`auth`	Autoryzacja do czatu	• `username` (nazwa użytkownika); • `password` (hasło)	• `session_key` (klucz sesji)
2	`rooms_list`	Uzyskanie listy pokojów (ogólnodostępnych)	*Brak.*	• `rooms_list` (tablica łańcuchów znaków będących nazwami pokojów)
3	`room_bind`	Wpięcie się do pokoju (ogólnodostępnego)	• `room` (nazwa pokoju)	*Brak.*
4	`room_msgs`	Uzyskanie ostatnich 10 wiadomości wysłanych do pokoju, do którego jest wpięty użytkownik	• `room` (nazwa pokoju)	• `msgs` (tablica obiektów reprezentujących ostatnie wiadomości w pokoju)
5	`unbind`	Odpięcie się od pokoju lub konwersacji prywatnej.	• `room` (nazwa pokoju)	*Brak.*
6	`users`	Lista użytkowników podpiętych do czatu, o nazwach rozpoczynających się podanym łańcuchem znaków	• `username` (pierwsze litery w nazwie użytkownika)	• `usernames` (lista łańcuchów znaków będących nazwami odnalezionych użytkowników; na liście może być maksymalnie 10 nazw, kolejne są pomijane)
7	`pm_join`	Podpięcie się pod konwersację wiadomości prywatnych z danym użytkownikiem.	• `username` (nazwa użytkownika z którym ma być nawiązana konwersacja);	*Brak.*

8	`pm_msgs`	Uzyskanie ostatnich 10 wiadomości prywatnych (z możliwością przewinięcia maksymanie 100 wiadomości wstecz)	• `offset` (liczba najnowszych wiadomości które można pominąć) • `username` (nazwa użytkownika którego dotyczy lista wiadomości)	• `msgs` (tablica obiektów reprezentujących wiadomości)
9	`room_new`	Utworzenie pokoju (dostępne tylko dla administratorów)	• `name` (nazwa pokoju)	*Brak.*
10	`room_del`	Usuwa wskazany pokój z serwera	• `name` (nazwa pokoju do usunięcia)	*Brak.*
11	`room_ban`	Wyrzuca użytkownika z pokoju w którym się znajduje (dostępne tylko dla administratorów)	• `username` (nazwa wyrzucanego użytkownika)	*Brak.*
12	`srv_ban`	Wyrzuca użytkownika z serwera (dostępne tylko dla administratora)	• `username` (nazwa wyrzucanego użytkownika)	*Brak.*
13	`status`	Zwrócenie informacji na temat bieżącego stanu połączenia	*Brak.*	• `username` (nazwa obecnego użytkownika); • `admin` (liczba 1 jeżeli obecny użytkownik jest administratorem, a w przeciwnym przypadku 0); • `current_rooms` (tablica nazw pokojów do których obecnie podpięty jest użytkownik; jeżeli nie jest podpięty, tablica jest pusta); • `current_pms` (tablica nazwa użytkowników z którymi obecnie prowadzona jest konwersacja prywatna; gdy taka konwersacja nie jest prowadzona, tablica jest pusty)
14	`change_pass`	Zmienia hasło użytkownika (dostępne tylko dla administratorów)	• `old_password` (treść starego hasła) • `new_password` (treść nowego hasła)	*Brak.*

8.3.1.3 Komunikaty asynchroniczne

Wszystkie komunikaty asynchroniczne mają tę samą budowę:

```
{                                                                    1
  'reason': 'msg',                                                   2
  'payload':                                                         3
```

```
  {                                                    4
    'content': 'Hej!'                                  5
  },                                                   6
  'session_key': 'abcdef123456'                        7
}                                                      8
```

Poniżej wskazano, jakie typy i znaczenia mają kolejne parametry:

- `reason` – łańcuch znaków wskazujący przyczynę wysłania wiadomości.
- `payload` – obiekt, który przenosi dodatkowe parametry z danymi przesyłanymi wraz z wiadomością; dokładna struktura obiektu zależy od tego, jaka jest przyczyna wysłania komunikatu; parametr `payload` jest obowiązkowy, ale zawarty w nim obiekt może być pusty `{}`;
- `session_key` – łańcuch znaków obowiązkowy przy wysyłaniu wiadomości na serwer, patrz analogiczny parametr we wiadomościach synchronicznych.

Powodów wysyłki wiadomości asynchronicznych jest znacznie mniej niż typów wiadomości synchronicznych. Poniżej wymieniono wszystkie z nich (wytłuszczono wartość parametru `reason`):

1. `msg` – nowa wiadomość w czacie:
   (a) wiadomość tego typu, wysłana od klienta do serwera, reprezentuje wiadomość wysłaną przez użytkownika do pokoju lub konwersacji prywatnej; komunikat powinien zawierać dwa obowiązkowe właściwości obiektu `payload`: właściwość `content` z łańcuchem znaków z treścią wiadomości oraz `room` z łańcuchem alfanumerycznych znaków z nazwą pokoju lub konwersacji prywatnej;
   (b) wiadomość tego typu wysłana od serwera do klienta oznacza nową wiadomość w jednym z pokojów lub prywatnych konwersacji, w których użytkownik bierze udział; wymagane są obowiązkowe własności obiektu `payload` tak jak we wcześniejszym przypadku;
2. `unbind_removed` – wysłana przez serwer, oznacza że wymuszono odpięcie się od pokoju z powodu jego usunięcia, w obiekcie `payload` obowiązkowa jest własność `room` zawierająca nazwę pokoju z którego użytkownik został wyrzucony;
3. `unbind_ban` – wysłana przez serwer, oznacza że użytkownik został wyrzucony z pokoju, w obiekcie `payload` obowiązkowa jest własność `room` zawierająca łańcuch znaków z nazwą pokoju, z któregp użytkownik został wyrzucony;
4. `disconn_ban` – wysłana przez serwer, oznacza że użytkownik został wyrzucony z serwera.

8.3.2 Protokół komunikacji pomiędzy aktorami

W celu zachowania porządku podczas opracowywania wielu współpacujących ze sobą aktorów, zadecydowano o narzuceniu pewnej ramowej struktury wszystkim komunikatom, które zostaną pomiędzy nimi wymienione. Przede wszystkim przyjęto założenie, że wszystkie komunikaty będą miały formę struktury `struct` o ściśle ustalonych, stale obecnych polach:

- `fn` – zawsze niepusty łańcuch znaków, jednoznacznie identyfikujący znaczenie komunikatu;
- `from` – zawsze wskaźnik na aktora, który nadał wiadomość;
- `sync` – zawsze liczba całkowita 1 lub 0, wskazująca, czy nadawca oczekuje na wiadomość zwrotną (1), czy też nie (0);
- `arg` – zawsze struktura `struct`, służąca do przenoszenia wraz z treścią wiadomości dodatkowych danych dla odbiorcy (np. parametrów wywołania); może być pusta;

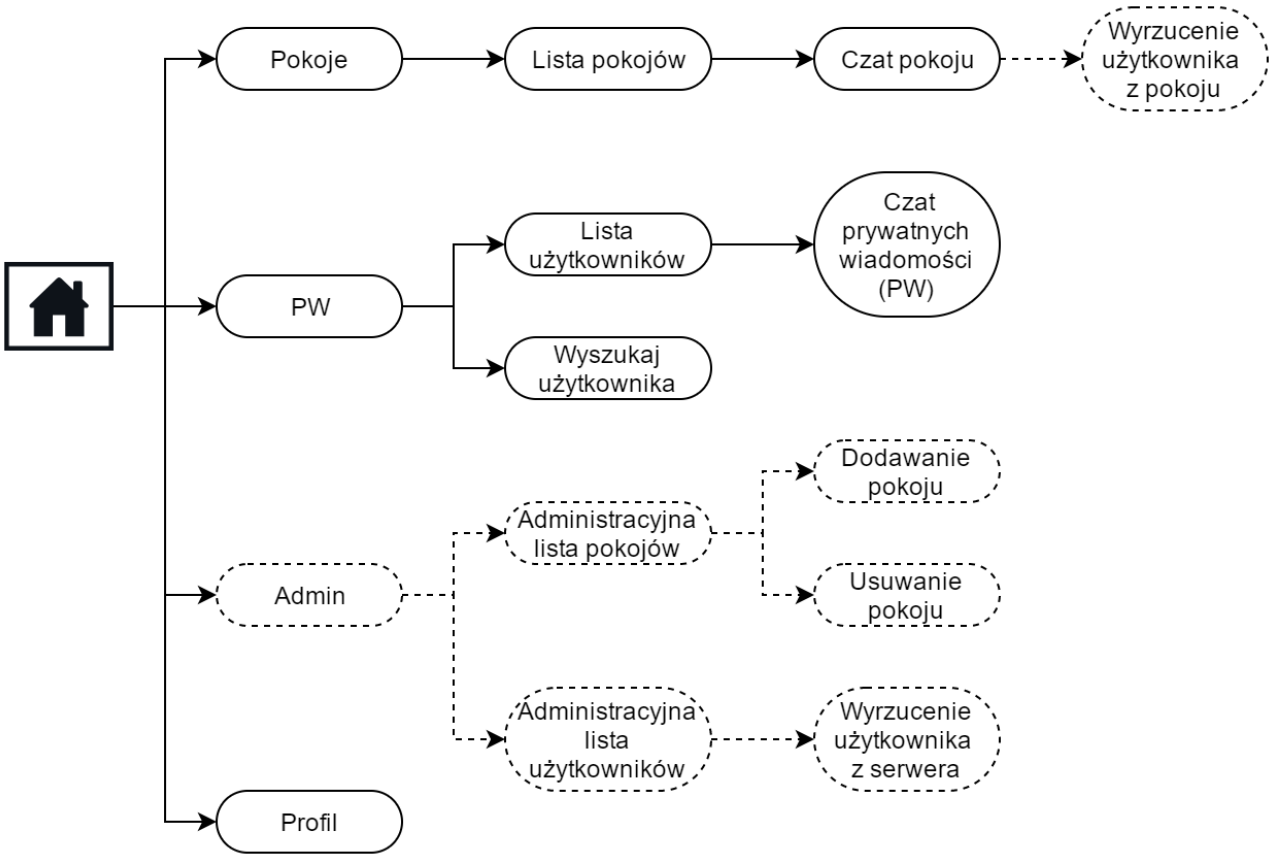
8.4 Projekt interfejsu

Interfejs graficzny czatu oparto na bibliotece Material, zapewniającej stabline i przetestowane rozwiązania z zakresu UI. Jej użycie w projekcie było możliwe dzięki temu, że twórcy udostępnili ją na licencji MIT.

Aby umożliwić szybką i łatwą nawigację po podstawowych funkcjonalnościach czatu, od początku zakładano, że w górnej części okna znajdzie się główne menu. Będzie ono obecne stale, przez cały czas korzystania z aplikacji. Pozwoli to na szybkie przechodzenie do uwidocznionych na nim opcji:

- **Pokoje** – dostęp do listy ogólnodostępnych pokojów;
- **PW** – dostęp do listy użytkowników, z którymi dotychczas wymieniono wiadomości prywatne;
- **Admin** – opcja widoczna wyłącznie po autoryzacji z uprawnieniami administracyjnymi, pozwala na przejście do narzędzi zarządzania serwerem (pokojami, użytkownikami);
- **Konto** – dostęp do informacji na temat bieżącego użytkownika i opcji związanych z bieżącą sesją.

Nawigacja po aplikacji czatu będzie odbywała się według schematu ujętego na rysunku 8.9.



Rysunek 8.9: Diagram nawigacji po interfejsie aplikacji ViuaChat (ścieżki i komponenty oznaczone przerywaną linią są dostępne wyłącznie dla administratorów)

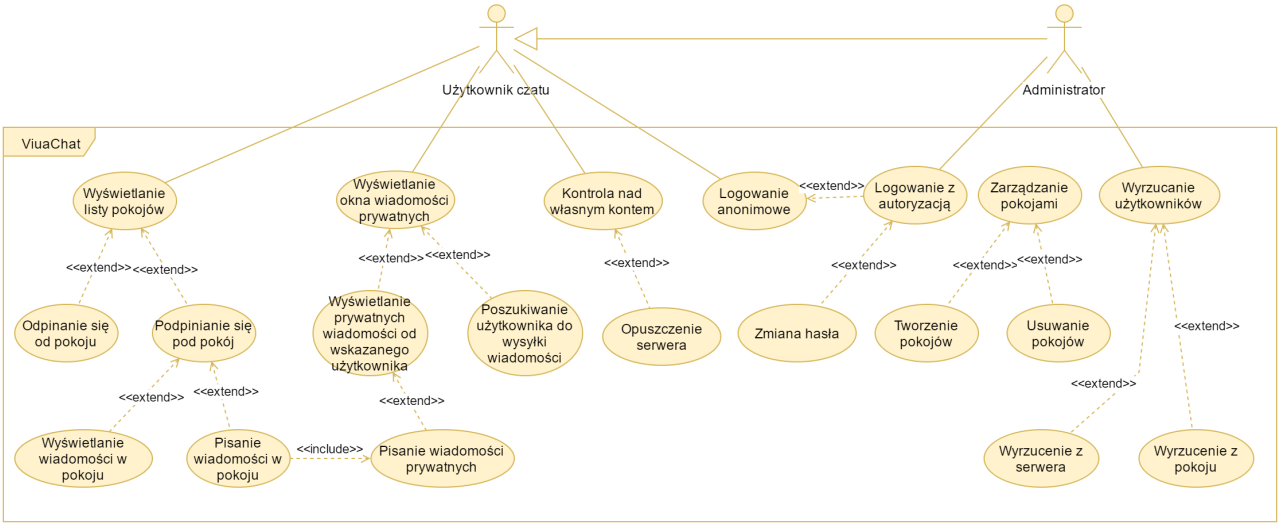
Wygląd okna czatu, z uwagi na oszczędności czasu i wąski zakres użycia czatu, zaprojektowano wyłącznie na najmniejsze urządzenia mobilne, rezygnując tym samym z przygotowania wariantów na inne rozmiary ekranów. Okno czatu ma stały, na sztywno ustalony wymiar 360 na 640 pikseli. Podejście do pozwoliło przeznaczyć zaoszczędzony czas na prace nad samą aplikacją serwera.

Rysunek 8.10: Projekt interfejsu graficznego czatu ViuaChat

8.5 Diagramy Przypadków Użycia

8.5.1 Diagram Przypadków Użycia

Diagram przypadków użycia jest pokazany na rysunku 8.11 Diagram przypadków użycia aplikacji ViuaChat.



Rysunek 8.11: Diagram przypadków użycia aplikacji ViuaChat

8.5.2 Opis aktorów

Uwaga! W niniejszej sekcji, słowo „aktor" jest używane w rozumieniu języka UML, a nie języka ViuAct.

1. **Użytkownik czatu.** Typ użytkownika, którego konto jest tworzone podczas połączenia z serwerem czatu oraz niszczone po jego zakończeniu. Podczas łączenia z czatem, nie będzie musiał się autoryzować przy użyciu hasła, a deklarować tylko unikalną nazwę, nie powtarzającą się z nazwą innego użytkownika, posiadającego konto na danym serwerze czatu. Ten typ konta jest przeznaczony dla osób, zainteresowanych dołączeniem do dyskusji na czacie bez dodatkowych zobowiązań. Jest aktorem aktywnym i głównym.

2. **Administrator.** To użytkownik, który jest dodatkowo wyróżniony i posiada uprawienia do szeroko pojętego zarządzania serwerem (w tym - pozostałymi użytkownikami). Konto administratora jest utrzymywane przez serwer pomiędzy połączeniami do czatu. Każdorazowo, przed rozpoczęciem sesji połączenia z serwerem, administratorzy muszą się dodatkowo autoryzować przyużyciu hasła. Równocześnie, ich nazwa jest zarezerwowana wyłącznie do jego użytku oraz niedostępna dla użytkowników tymczasowych. Nie wyróżnia się wśród administratorów żadnych dodatkowych, szczególnych ról (np. superadministrator, właściciel). Jest aktorem aktywnym i głównym.

8.5.3 Opis przypadków użycia

Poniżej opisano przypadki użycia ujęte na rysunku 8.11 Diagram przypadków użycia aplikacji ViuaChat.

Identyfikator	UC-01
Nazwa	Logowanie anonimowe
Aktorzy	Użytkownik czatu
Streszczenie	Użytkownik rozpoczyna korzystanie z czatu pod wybraną nazwą użytkownika, ale bez konieczności podawania hasła.
Warunek wstępny	Użytkownik nie ma rozpoczętej sesji połączenia z serwerem
Wyjątki	• Użytkownik ma już wcześniej rozpoczętą sesję z serwerem
Scenariusz podstawowy	1. Użytkownik wprowadza nazwę użytkownika i zatwierdza 2. System sprawdza czy nazwa użytkownika jest wolna 3. Gdy nazwa jest wolna, serwer rozpoczyna sesję z użytkownikiem
Scenariusze alternatywne	1. Gdy nazwa użytkownika jest zajęta (przez zalogowanego użytkownika lub stałe konto użytkownika), logowanie nie powiedzie się.
Warunek końcowy	Użytkownik ma rozpoczętą sesję z serwerem.
Komentarz	*Brak*

Identyfikator	UC-02
Nazwa	Logowanie z autoryzacją
Aktorzy	Administrator
Streszczenie	Administrator rozpoczyna korzystanie z czatu z wykorzystaniem stałego konta użytkownika. Podczas logowania, administrator, oprócz podana samej nazwy użytkownika, dodatkowo uwierzytelnia się hasłem, aby zweryfikować czy ma prawo do wykorzystywania konta.
Warunek wstępny	1. Użytkownik nie ma rozpoczętej żadnej sesji połączenia z serwerem. 2. Użytkownik posiadał wcześniej skonfigurowane konto administratora na serwerze
Wyjątki	• Użytkownik ma już wcześniej rozpoczętą sesję z serwerem
Scenariusz podstawowy	1. Użytkownik wprowadza nazwę użytkownika, hasło i zatwierdza 2. Gdy istnieje konto administratora o wskazanej nazwie, a podane hasło jest z nim zgodne, wówczas serwer rozpoczyna sesję z użytkownikiem.
Scenariusze alternatywne	1. Gdy nie istnieje konto administratora o podanej nazwie, próba zalogowania z hasłem nie powiedzie się - w przypadku logowania bez użycia hasła, patrz UC-01. 2. Gdy podane hasło nie jest zgodne z hasłem do konta administratora o podanej nazwie, logowanie nie powiedzie się.
Warunek końcowy	Użytkownik ma rozpoczętą sesję z serwerem.
Komentarz	*Brak*

Identyfikator	UC-03
Nazwa	Wyświetlanie listy pokojów
Aktorzy	Użytkownik czatu
Streszczenie	Użytkownik uzyskuje wgląd do listy pokojów z której może wybrać ten, do którego chce się podpiąć.
Warunek wstępny	1. Użytkownik ma rozpoczętą sesję z serwerem
Wyjątki	1. Użytkownik nie może być wcześniej podpięty do żadnego pokoju. *Brak*
Scenariusz podstawowy	1. Użytkownik wybiera z górnego menu opcję „Pokoje”. 2. Pokazywany jest ekran listy pokojów. 3. Użytkownik wybiera z listy nazwę pokoju, do którego chce się wpiąć.
Scenariusze alternatywne	1. Wiadomości prywatne są wysyłane z okna czatu w specyficzny sposób (patrz UC-10) 2. Tuż po zalogowaniu do serwera, scenariusz podstawowy zaczyna się od kroku 2.
Warunek końcowy	Użytkownik wybrał pokój do wpięcia się.
Komentarz	*Brak*

Identyfikator	UC-04
Nazwa	Podpinanie się pod pokój
Aktorzy	Użytkownik czatu
Streszczenie	Użytkownik czatu, po wybraniu pokoju z listy pokojów, jest do niego wpinany.
Warunek wstępny	1. Użytkownik wybrał pokój z listy pokojów.
Wyjątki	1. Użytkownik nie może być wcześniej wpięty do żadnego pokoju.
Scenariusz podstawowy	1. Użytkownik wybiera pokój z listy 2. Serwer weryfikuje, czy użytkownik nie był już wcześniej wpięty do innego pokoju 3. Jeżeli użytkownik pozostawał wcześniej niepodpięty, serwer sprawdza, czy wybrany pokój nadal istnieje, 4. Jeżeli pokój nadal istnieje, następuje podpięcie użytkownika pod wybrany pokój. 5. Uzytkownikowi, który zostaje nowo podpięty do pokoju, pokazywane jest 10 wiadomości wysłanych tuż przed jego dołączeniem do tego pokoju.
Scenariusze alternatywne	1. Gdy użytkownik był wcześniej wpięty do innego pokoju, najpierw zostaje od niego odpięty (UC-05), a dopiero później zostaje wpięty do wybranego pokoju.
Warunek końcowy	Użytkownik został podpięty pod pokój.
Komentarz	*Brak*

Identyfikator	UC-05
Nazwa	Odpinanie się od pokoju
Aktorzy	Użytkownik czatu
Streszczenie	Użytkownik, który był wcześniej wpięty do pokoju, może się od niego odpiąć, aby wpiąć się do innego pokoju lub po prostu zrezygnować z dalszej konwersacji.
Warunek wstępny	1. Użytkownik ma rozpoczętą sesję z serwerem 2. Użytkownik jest podpięty do pokoju
Wyjątki	*Brak*
Scenariusz podstawowy	1. Użytkownik wybiera przycisk „Opuść pokój”. 2. Serwer weryfikuje, czy użytkownik nadal jest podpięty pod pokój. 3. Jeżeli użytkownik jest nadal podpięty, następuje odpięcie. 4. Użytkownik zostaje przekierowany do listy pokojów (UC-03 od kroku 2)
Scenariusze alternatywne	1. Gdy użytkownik pozostawał wcześniej niepodpięty, akcja kończy się niepowodzeniem.
Warunek końcowy	Użytkownik zostaje odpięty od pokoju.
Komentarz	*Brak*

Identyfikator	UC-06
Nazwa	Pisanie wiadomości w pokoju
Aktorzy	Użytkownik czatu
Streszczenie	Użytkownik może napisać wiadomość w pokoju czatu, którą zobaczą inni użytkownicy podpięci do tego pokoju (włącznie z jej nadawcą)
Warunek wstępny	1. Użytkownik ma rozpoczętą sesję z serwerem 2. Użytkownik jest podpięty do pokoju
Wyjątki	*Brak*
Scenariusz podstawowy	1. Użytkownik pisze wiadomość w polu tekstowym pod wiadomościami czatu. 2. Po zatwierdzeniu wiadomości do wysyłki, pole tekstowe jest czyszczone. 3. Gdy użytkownik jest nadal podpięty do pokoju, wiadomość zostaje wpisana do listy wiadomości i rozesłana do wszystkich użytkowników. 4. Użytkownik widzi swoją wiadomość wyświetloną u dołu ekranu.
Scenariusze alternatywne	1. Gdy użytkownik nie był wpięty do pokoju w momencie wysyłania wiadomości, wysyłka kończy się niepowodzeniem 2. Gdy wiadomość jest poprzedzona znakiem ..#”, to jest realizowany scenariusz UC-11
Warunek końcowy	Wiadomość została zaakceptowania do rozesłania przez serwer.
Komentarz	*Brak*

Identyfikator	UC-08
Nazwa	Wyświetlanie wiadomości prywatnych w pokoju
Aktorzy	Użytkownik czatu
Streszczenie	Użytkownik widzi wiadomości prywatne, skierowane do niego przez innego użytkownika podpiętego do tego samego pokoju. Wiadomości tych nie widzi żaden inny użytkownik podpięty do pokoju, oprócz jej nadawcy i odbiorcy.
Warunek wstępny	1. Użytkownik ma rozpoczętą sesję z serwerem 2. Użytkownik jest podpięty do pokoju 3. Użytkownik wysłał wiadomość do pokoju (patrz UC-07) poprzedzoną znakiem „#”
Wyjątki	*Brak*
Scenariusz podstawowy	1. Użytkownik wysyła do pokoju wiadomość poprzedzoną znakiem „#”. 2. Serwer sprawdza, czy przed treścią wiadomości znajduje się łańcuch znaków będący prawidłową nazwą użytkownika. 3. Jeżeli forma nazwy jest prawidłowa, serwer sprawdza, czy użytkownik wskazany na początku wiadomości jest podpięty do pokoju. 4. Jeżeli użytkownik jest podpięty do pokoju, wiadomość zostaje przyjęta przez serwer jako wiadomość prywatna.
Scenariusze alternatywne	1. Gdy nazwa użytkownika odbiorcy ma nieprawidłową formę, wiadomość nie zostanie przyjęta przez serwer i wysyłka zakończy się niepowodzeniem 2. Gdy odbiorca wiadomości nie jest podpięty pod pokój, wiadomość nie zostanie przyjęta przez serwer i wysyłka zakończy się niepowodzeniem.
Warunek końcowy	Wiadomość prywatna została przyjęta przez serwer.
Komentarz	*Brak*

Identyfikator	UC-09
Nazwa	Wyświetlanie okna wiadomości prywatnych
Aktorzy	Użytkownik czatu
Streszczenie	Użytkownik może zobaczyć okno z wiadomościami prywatnymi (niezależnie od tego czy zostały wysłane z pokoju czy z okna wiadomości prywatnych), pogrupowane wg ich nadawców/odbiorców
Warunek wstępny	1. Użytkownik ma rozpoczętą sesję z serwerem.
Wyjątki	*Brak*
Scenariusz podstawowy	1. Użytkownik wybiera z menu opcję „PW” 2. Użytkownikowi zostaje pokazana lista nazw użytkowników, od których otrzymał lub którym wysyłał wiadomości prywatne.
Scenariusze alternatywne	1. Gdy użytkownik nie wysłał ani nie odebrał żadnych wiadomości prywatnych, lista nazw użytkowników będzie pusta.
Warunek końcowy	Użytkownik zobaczy listę nazw użytkowników, od których otrzymał lub którym wysłał wiadomości prywatne.
Komentarz	*Brak*

Identyfikator	UC-10
Nazwa	Wyświetlanie prywatnych wiadomości od wskazanego użytkownika
Aktorzy	Użytkownik czatu
Streszczenie	Użytkownik musi wybrać konkretnego innego użytkownika, aby zobaczyć jego wiadomości (tj. wiadomości prywatne, których jest nadawcą/odbiorcą).
Warunek wstępny	1. Użytkownik wybrał nazwę użytkownika w oknie wiadomości prywatnych.
Wyjątki	1. Użytkownik nie wysłał ani nie odebrał dotychczas żadnych wiadomości prywatnych.
Scenariusz podstawowy	1. Użytkownik wybiera jedną z nazw, którą widzi na liście w oknie wiadomości prywatnych 2. Użytkownikowi pokazywana jest lista wiadomości prywatnych, które otrzymał od tego użytkownika lub do których je skierował.
Scenariusze alternatywne	1. Gdy wybrany użytkownik nie istnieje i/lub nie jest połączony z serwerem, operacja zakończy się błędem.
Warunek końcowy	Użytkownik zobaczył wiadomości prywatne, które odebrał od lub nadał do konkretnego użytkownika.
Komentarz	*Brak*

Identyfikator	UC-11
Nazwa	Pisanie prywatnych wiadomości
Aktorzy	Użytkownik czatu
Streszczenie	Użytkownik w oknie wiadomości prywatnych pisze wiadomości, które są domyślnie uznawane za wiadomości prywatne skierowane do użytkownika, który został wcześniej wybrany.
Warunek wstępny	1. Użytkownik jest w oknie wiadomości prywatnych. 2. Użytkownik wybrał, czyje wiadomości ogląda (patrz UC-10)
Wyjątki	1. Użytkownik wybrany w oknie czatu rozłączył się z serwerem, zanim wiadomość została wysłana.
Scenariusz podstawowy	1. Użytkownik wpisuje wiadomość do pola tekstowego pod wiadomościami w oknie wiadomości prywatnych 2. Użytkownik decyduje o wysłaniu wiadomości. 3. Pole tekstowe zostaje wyczyszczone. 4. Serwer przyjmuje wiadomość jako wiadomość prywatną i rozsyła do nadawcy i odbiorcy 5. Użytkownik widzi wysłaną wiadomość u dołu pola tekstowego.
Scenariusze alternatywne	1. Gdy wybrany użytkownik odłączył się od serwera, wysłanie wiadomości kończy się niepowodzeniem.
Warunek końcowy	Wiadomość została pokazana nadawcy i odbiorcy.
Komentarz	Wiadomości wysyłane z okna wiadomości prywatych nie muszą być poprzedzane znakiem „#” i nazwą odbiorcy. Użytkownik docelowy jest wnioskowany z tego, czyje wiadomości są obecnie pokazywane w oknie wiadomości prywatnych (patrz UC-10)

Identyfikator	UC-12
Nazwa	Wyrzucanie użytkowników z serwera
Aktorzy	Administrator
Streszczenie	Administrator ma prawo w dowolnym momencie przerwać połączenie dowolnego użytkownika z serwerem.
Warunek wstępny	1. Administrator ma rozpoczętą sesję z serwerem. 2. Wybrany uzytkownik ma rozpoczetą sesję z serwerem i nadal jest z nim połączony.
Wyjątki	Nie jest możliwe wyrzucenie samego siebie.
Scenariusz podstawowy	1. Administrator klika nazwę użytkownika (niezależnie od miejsca, w którym została wyświetlona). 2. Z menu, administrator wybiera opcję „Wyrzuć z serwera”. 3. Wskazany użytkownik zostaje niezwłocznie odpięty z pokoju (o ile był podpięty do któregokolwiek). 4. Połączenie wybranego użytkownika zostaje zakończone.
Scenariusze alternatywne	1. Gdy wybrany użytkownik nie jest już połączony z serwerem, akcja kończy się niepowodzeniem.
Warunek końcowy	Wskazany użytkownik został odłączony od serwera.
Komentarz	*Brak*

Identyfikator	UC-13
Nazwa	Wyrzucanie użytkowników z pokojów
Aktorzy	Administrator
Streszczenie	Administrator może odpiąć wybranego użytkownika od pokoju, do którego jest obecnie wpięty.
Warunek wstępny	1. Administrator ma rozpoczętą sesję z serwerem. 2. Wybrany użytkownik jest wpięty do jakiegokolwiek pokoju.
Wyjątki	*Brak*
Scenariusz podstawowy	1. Administrator klika nazwę użytkownika, przebywając w oknie pokoju. 2. Z menu, administrator wybiera opcję „Wyrzuć z pokoju”. 3. Wskazany użytkownik zostaje niezwłocznie odpięty z pokoju.
Scenariusze alternatywne	1. Gdy użytkownik, przed podjęciem przez administratora decyzji o wyrzuceniu, sam wypiął się z pokoju lub zerwał połączenie z serwerem, operacja kończy się niepowodzeniem.
Warunek końcowy	Użytkownik został wypięty z pokoju.
Komentarz	*Brak*

Identyfikator	UC-14
Nazwa	Tworzenie pokojów
Aktorzy	Administrator
Streszczenie	Administrator ma prawo tworzyć i usuwać pokoje
Warunek wstępny	1. Administrator ma rozpoczętą sesję z serwerem
Wyjątki	*Brak*
Scenariusz podstawowy	1. Administrator wchodzi w listę pokojów 2. Administrator klika w ikonę plusa obok nagłówka listy 3. Pokazany zostaje okno utworzenia nowego pokoju 4. W oknie utworzenia nowego pokoju, administrator wpisuje nazwę pokoju. 5. Administrator zatwierdza utworzenie pokoju. 6. Jeżeli nazwa pokoju jest prawidłowa pod względem wymagań biznesowych, tworzony jest nowy pokój.
Scenariusze alternatywne	1. Gdy nazwa pokoju pokrywa się z istniejącym pokojem lub nie spełnia wymagań biznesowych, administrator jest poproszony o podanie takiej, która jest prawidłowa. 2. Gdy administrator wyjdzie z okna utworzenia nowego pokoju bez jego zatwierdzenia, pokój nie zostaje dodany.
Warunek końcowy	Pokój został utworzony.
Komentarz	*Brak*

Identyfikator	UC-15
Nazwa	Usuwanie pokojów.
Aktorzy	Administrator
Streszczenie	Administrator ma prawo usuwać pokoje z serwera.
Warunek wstępny	1. Administrator ma rozpoczętą sesję z serwerem
Wyjątki	*Brak*
Scenariusz podstawowy	1. Administrator przechodzi do listy pokojów. 2. Administrator klika w ikonę trzech pionowych kropek obok nazwy wybranego pokoju. 3. W rozwijanym menu zostaje pokazana opcja „Usuń”. 4. Zostaje pokazany monit z prośbą o potwierdzenie decyzji, w którym zostaje obowiązkowo wymieniona nazwa pokoju. 5. Po zatwierdzeniu decyzji, użytkownicy wpięci do usuwanego pokoju zostają usunięte, a następnie sam pokój zostaje usunięty.
Scenariusze alternatywne	1. Gdy pokój przestahe istnieć pomiędzy wybraniem a potwierdzeniem usunięcia, operacja zostanie zakończona niepowodzeniem. 2. Gdy administrator zrezygnuje z usunięcia pokoju na etapie okna potwierdzającego usunięcie, pokój pozostanie nieusunięty.
Warunek końcowy	Pokój zostaje usunięty.
Komentarz	*Brak*

Część IV

Podsumowanie

Rozdział 9

Raport końcowy

W tym rozdziale prezentujemy raport końcowy z efektów pracy inżynierskiej. Podsumowujemy odniesione sukcesy (wykonane oprogramowanie, wytworzoną dokumentację), porażki (elementy, na których wykonanie nie wystarczyło czasu, okazały się zbyt trudne do wykonania w ramach projektu inżynierskiego, bądź z innych powodów musiały zostać odrzucone), oraz zmiany, które musieliśmy wprowadzać do planu w trakcie trwania projektu (m.in. decyzja o uwzględnieniu raportowania błędów w kompilatorze).

9.1 Sukcesy

Niewątpliwym sukcesem jest doprowadzenie pracy inżynierskiej do końca. Patrząc na „zewnętrzne" efekty pracy osiągnęliśmy wszystko co było zakładane – czyli zarówno specyfikację i implementację (w formie kompilatora) języka wysokiego poziomu na platformie Viua VM, oraz nietrywialny program – ViuaChat – napisany w języku ViuAct. Przygotowano również kompletne, zintegrowane środowisko, dzięki czemu wyniki prac mogą być użyteczne dla osób spoza zespołu projektowego.

Nieoczywistym sukcesem jest również fakt, że Krzysztof Franek wykorzystał język ViuAct w trakcie zajęć na Uczelni – pisząc program zaliczeniowy z przedmiotu „Narzędzia Sztucznej Inteligencji" w języku ViuAct.

9.1.1 Ocena i poprawa stanu Viua VM

Projekt pozwolił krytycznym okiem spojrzeć na stan w jakim znajdowała się Viua VM i ocenić ją w obiektywny sposób za pomocą jasnego kryterium: albo da się oprzeć na niej implementację języka wysokiego poziomu i napisać „prawdziwy" program, albo nie.

W trakcie pracy okazało się, że odpowiedź na to pytanie jest *raczej pozytywna*. Niektóre aspekty Viua VM wymagają pracy (który to fakt był znany), jednak przeprowadzenie projektu inżynierskiego na jej podstawie pozwoliło zlokalizować największe niedociągnięcia, a nawet część z nich usunąć.

Jednym z obszarów, który został znacząco poprawiony w trakcie prac nad projektem inżynierskim było łączenie modułów w finalne programy.

9.1.2 Opracowanie systemu ViuaChat

Pomimo licznych przeciwności i problemów wynikających z początkowej niedojrzałości języka ViuAct i jego kompilatora, udało się szczęśliwie ukończyć system ViuaChat. Było to możliwe dzięki zacieśnionej współpracy obu członków zespołu, a także dzięki kreatywnemu podejściu do napotykanych utrudnień.

9.2 Wyzwania i niepowodzenia

W trakcie realizacji projektu nie wszystko przebiegło po myśli zespołu.

9.2.1 Prędkość kompilacji

Przede wszystkim, wiele do życzenia pozostawiała prędkość kompilacji, znacznie spadająca wraz z przyrastającym kodem źródłowym. Co ciekawe, w początkowej fazie prac, za źródło problemów uważano implementację kompilatora ViuAct w języku Python ponieważ programy napisane w Pythonie ustępują wydajnością programom napisanym w językach takich jak C++, czy też OCaml.

Ku zaskoczeniu studentów, wraz z kolejnymi linijkami kodu, widocznemu wydłużeniu ulegała faza asemblacji, za którą odpowiadał assembler maszyny wirtualnej Viua VM. W końcowych wersjach serwera ViuaChat, asemblacja była do 5 razy bardziej czasochłonna od kompilacji. Niestety, poprawianie wydajności asemblera pozostawało poza zakresem projektu, a wydłużony czas asemblacji musiał zostać zaakceptowany.

Dlaczego kompilacja trwa tak długo?

Całość procesu, który z plików źródłowych w języku ViuAct wyprodukuje pliki z binarną reprezentacją programu w *bytecode* Viua VM trwa długo. Przy nietrywialnych programach, takich jak ViuaChat, etap przetwarzania kodu programu w języku assemblera Viua VM na formę binarną (realizowany przez assembler dostarczany przez Viua VM) zajmuje dużą ilość czasu.

Wynika to po części z faktu, że nietrywialne programy zawierają dużą ilość wyrażeń warunkowych. Analizator statyczny wbudowany w assembler dostarczany przez Viua VM duplikuje swój stan dla każdej instrukcji skoku warunkowego (emitowanej dla każdego wyrażenia warunkowego) i rozważa wykonanie każdej ścieżki osobno. Jest to potrzebne ponieważ na podstawie niezależnej analizy każdej ścieżki wykonania analizator wykrywa między innymi martwe (tj. takie, który program nie wykorzystuje) wartości i rejestry – jednak powoduje to, że każda instrukcja skoku warunkowego w teorii podwaja koszt analizy.

9.2.2 Pozostanie przy prototypie kompilatora

Najbardziej widoczną porażką projektu jest pozostawienie implementacji języka ViuAct w fazie prototypowej – budżet czasowy jakim dysponowaliśmy okazał się niewystarczający na stworzenie prototypu kompilatora (w języku Python) w celu przeprowadzenia „rozpoznania" przestrzeni problemu, a następnie korzystając z nabytej wiedzy napisanie kompilatora „produkcyjnego" (w języku OCaml).

9.2.3 Rozbieżności między specyfikacją, a implementacją

Rozbieżności wynikają z tego, że żeby szybko uzyskać działający w dużej liczbie przypadków kompilator czasami implementacja „szła na skróty".

Spowodowało to, że kompilator, którym dysponujemy nie jest w stanie przetworzyć stu procent możliwych kombinacji konstrukcji językowych wynikających ze specyfikacji języka. Nie uniemożliwiło to co prawda napisania programu ViuaChat jednak wprowadziło pewne opóźnienia w pracach nad nim z powodu rozbieżności między teorią (specyfikacją języka), a praktyką (implementacją języka).

Dowiązania *let* i wysłuskiwanie wskaźników

Listing 9.1: Błędnie działający przykład dowiązania do wyłuskania

```
(let main () {                                  1
    (let n 42)                                  2
    (let ptr (Std.Pointer.take n))              3
    (let x (^ ptr))                             4
    (print x)                                   5
    0                                           6
})                                              7
```

Jednym z przykładów kodu, który nie daje oczekiwanych rezultatów jest przypisanie wyniku wyłuskania wskaźnika do zmiennej:

```
(let x (^ ptr))
```

W zmiennej `x` nie znajduje się, jak należałoby oczekiwać, kopia wartości, na którą wskazywał wskaźnik `ptr`. Zmiennej `x` tak naprawdę *nie ma* po skompilowaniu. Wynika to z faktu, że operator wyłuskania wskaźnika nie zwraca wartości, a jedynie wpływa na sposób adresowania wartości w rejestrach.

Aby zapisać w zmiennej wynik wyłuskania należy posłużyć się funkcją `Std.copy`:

```
(let x (Std.copy (^ ptr)))
```

Funkcja ta zwraca kopię wartości podanej jej jako parametr, czyli w powyższym przypadku skopiuje wartość, na którą wskazuje wskaźnik `ptr`.

Listing 9.2: Poprawnie działający przykład dowiązania do wyłuskania

```
(let main () {
    (let n 42)
    (let ptr (Std.Pointer.take n))
    (let x (Std.copy (^ ptr)))
    (print x)
    0
})
```

Dostęp do pól struktur

Jednym z bardziej problematycznych elementów jest również dostęp do pól struktur, do których mamy jedynie wskaźnik. Żeby uzyskać dostęp do takiego pola należy skopiować całą strukturę. Jest to szalenie niewydajne i widocznie spowalnia działanie oprogramowania.

Tworzenie kopii jest wymuszone tym, że poniższa kombinacja konstrukcji językowych jest przez kompilator przetwarzana nieprawidłowo, już na etapie parsowania kodu źródłowego:

```
(print (^ ptr).some.field)
```

W teorii, ten fragment kodu powinien wydrukować na ekran wartość przypisaną do pola `some.field` wewnątrz struktury, na którą wskazuje wskaźnik `ptr`. W praktyce, ten fragment wydrukuje na ekran strukturę, na którą wskazuje wskaźnik. Wynika to z faktu, że parser zakłada, że wszystkie wyrażenia składają się z pojedynczej grupy zamkniętej w nawiasach. W przypadku dostępów do pól to założenie jest fałszywe – dostępy do pól są jedynymi wyrażeniami w języku, które tego założenia nie spełniają (i jedynymi które nie są zapisywane w notacji prefiksowej, a w infiksowej).

Parser i specyfikacja zakładają, że dostępy do pól (czyli operator `.`) będą miały z lewej strony zawsze nazwę, nie wrażenie. Nie jest to więc „rozbieżność" między implementacją, a specyfikacją. Krzysztof Franek jednak podczas prac na programem ViuaChat spróbował wykorzystać taką konstrukcję i natknął się na tą nieścisłość w projekcie języka.

9.2.4 Typowanie dynamiczne

Mimo iż nie było to ujęte w wymaganiach, za drobną porażkę uważam również dyscyplinę typowania jaką charakteryzuje się język ViuAct. Dyscyplinę typowania można określić na dwóch głównych osiach:

1. dynamiczne—statyczne (*dynamic—static*) – określająca kiedy weryfikowane są typy danych. W typowaniu dynamicznym na etapie wykonywania (*run-time*), a w typowaniu statycznym na etapie kompilacji (*compile-time*).
2. słabe—silne (*weak—strong*) – określająca jaka jest reakcja języka na użycie niekompatybilnych typów danych. Typowanie słabe zakłada automatyczną koercję typów danych, a typowanie silne zakłada zgłoszenie błędu.

Klasyfikując język ViuAct na dwóch osiach dyscyplin typowania otrzymujemy typowanie dynamiczne—silne. Lepsze gwarancje poprawności można uzyskać kombinacją statycze—silne, jednak zbudowanie poprawnego statycznego systemu typów jest samo w sobie czasochłonne i skomplikowane, co dyskwalifikowało taką dyscyplinę typowania już na starcie projektu.

9.3 Zmiany założeń wprowadzone w trakcie trwania projektu

Większość założeń, które zostały określone na początku projektu okazała się słuszna, a ogólna koncepcja oraz kierunek rozwoju projektu nie wymagały dużych zmian.

9.3.1 Obsługa błędów kompilacji

Drobnej korekcie uległo założenie „nie implementujemy obsługi błędów w kompilatorze". Okazało się, że potrzebna jest przynajmniej minimalna obróbka błędów zgłaszanych przez kompilator ponieważ surowe zrzuty stosu nie były wystarczające do szybkiego odnajdowania błędów.

9.3.2 Obsługa protokołu WebSocket

Większą zmianą założeń była rezygnacja z wykorzystania gotowej biblioteki implementującej protokół WebSocket i zamiast tego implementacja tego protokołu „od zera". Wynikło to z bardzo prostego powodu – każda biblioteka, która była brana pod uwagę okazywała się tak naprawdę *frameworkiem* co dyskwalifikowało jej wykorzystanie w Viua VM.

Frameworki narzucają programom swój reżim i wizję tego jak ma wyglądać współpraca z nimi – narzucają współpracę o dostarczany przez siebie *event-loop*. Viua VM jednak w bardzo ostry sposób wchodzi w konflikt z takim podejściem ponieważ sama narzuca reżim i model programowania. Ta „niekompatybilność" wymusiła na nas wykonanie implementacji modułu do obsługi protokołu WebSocket w formie biblioteki – czyli modułu pozostawiającego kontrolę „w rękach" programu.

9.3.3 Wskaźniki w ViuAct

W pierwszych wersjach języka ViuAct, wskaźniki nie istniały w jawnej formie. Nie przewidziano konstrukcji do tworzenia i rozwiązywania wskaźników, ani pozwalającej na jawne skopiowanie wartości do której odwołuje się wskaźnik. Nie przeszkadzało to jednak, aby wskaźniki występowały w formie niejawnej.

Kompilator stosował je w wynikowym kodzie asemblera, np. przy odwoływaniu się do zawartości w typach złożonych. Z czasem okazało się, że brak jawnych instrukcji, które pozwoliłyby na manualne wywołanie operacji na wskaźnikach, istotnie ogranicza możliwości programisty. Problemy pojawiały się podczas iteracji po wektorach czy też przy wykorzystywaniu wartości pól w strukturach, które zostały uprzednio „wyłuskane" i zwrócone przez funkcję.

Zgłoszone niedociągnięcia doprowadziły do ubogacenia składni języka ViuAct o instrukcję wyłuskania `(^ ptr)`, kopiowania wskazywanej wartości `(Std.copy ptr)` oraz jawnego utworzenia wskaźnika `(Std.Pointer.take var)`.

9.3.4 Zakres funkcjonalności systemu ViuaChat

W początkowej fazie projektu, system ViuaChat miał zostać zrealizowany ze znacznie większym rozmachem. Z czasem, wskutek ożywionych dyskusji i ciągłych korekt harmonogramu, stawało się jasne że utrzymanie pierwotnie założonych funkcjonalności nie będzie możliwe. Stąd zrezygnowano z kilku z nich, m.in.:

1. możliwość wysyłania i odbierania wiadomości prywatnych z okien pokojów ogólnodostępnych
2. możliwość zakładania stałych kont zabezpieczonych hasłem dla użytkowników bez uprawnień administracyjnych
3. możliwość zabezpieczania dostępu do wybranych pokojów poprzez konieczność podania dodatkowego hasła

Pomimo zarzucenia tych i innych pomysłów, nie naruszono podstawowego rdzenia funkcjonalności, ustalonego jeszcze przed przystąpieniem do samego projektu.

9.4 Wpływ pracy na platformę Viua VM

W tym rozdziale omówiony jest wpływ naszej pracy inżynierskiej na platformę Viua VM; zarówno na implementację samej maszyny wirtualnej i dopracowanie jej mechanizmów, ale też na „ekosystem" dookoła niej – nowe moduły biblioteki standardowej i zewnętrzne.

9.4.1 System modułów

Umieszczenie w specyfikacji języka ViuAct systemu modułów wymusiło „ucywilizowanie" systemu modułów i linkera Viua VM. Stan w jakim oba te komponenty znajdowały się przed rozpoczęciem pracy inżynierskiej powodował, że nadawały się one do wykorzystania jedynie w najprostrzych przypadkach.

Po zakończeniu pracy inżynierskiej system modułów i linker Viua VM pozwalają na wykorzystanie w dużo szerszym wachlarzu zastosowań, są prostsze w użyciu, a linkowanie dynamiczne ma mniejszy narzut. Ujednolicone zostały również zasady linkowania modułów „własnych" (tj. modułów *bytecode* Viua VM) i „obcych" (tj. napisanych w języku C++).

9.4.2 Biblioteka standardowa

Biblioteka standardowa została powiększona o moduły do obsługi I/O na plikach, moduł dający dostęp do API systemowego *POSIX sockets*, oraz moduł do obsługi formatu serializacji danych JSON. Wszystkie te moduły zostały wykorzystane w implementacji programu ViuaChat.

9.4.3 Moduł do obsługi protokołu WebSocket

Do wykonania czatu potrzebny był moduł do obsługi protokołu WebSocket. Został on dostarczony jako biblioteka obca z plikiem interfejsu umożliwiającym jej wykorzystanie w języku ViuAct. Moduł ten jest opisany w załączniku C na stronie 175.

Rozdział 10

Wkład własny

10.1 Wspólnie zrealizowane zadania

Obaj członkowie zespołu, Marek Marecki i Krzysztof Franek, brali udział w pracach nad składnią języka ViuAct. Wspólnym ustaleniom podlegały zastosowane słowa kluczowe, sposoby budowania wyrażeń, zakres biblioteki standardowej oraz zbiór struktur jakie miały być dostarczane przez język.

Wspólnie ustalono również zakres funkcjonalności, których obecność była porządana w systemie ViuaChat. Zakres ten skodyfikowano w postaci zasad biznesowych, będących przyczynkiem do dalszych samodzielnych prac Krzysztofa Franka. Ożywionej konsultacji poddawano w szczególności te z nich, z którymi wiązałyby większe nakłady pracy na etapie szczegółowego projektowania czy implementacji.

10.2 Marek Marecki

Głównymi zadaniami Marka Mareckiego było opracowanie specyfikacji języka ViuAct oraz implementacja kompilatora i biblioteki standardowej tegoż języka. Dodatkowymi zadaniami, które wynikły w trakcie projektu były (1) implementacja bibliotek, na które zapotrzebowanie zgłosił Krzysztof Franek (2) dostosowanie Viua VM do potrzeb projektu przez drobne poprawki i przeróbki (3) stworzenie skryptu tworzącego środowisko programistyczne, który pozwolił obu członkom zespołu pracować w jednolitych warunkach.

Specyfikacja języka ViuAct obejmowała między innymi określenie ogólnego charakteru i semantyki języka, oraz skonkretyzowanie tych ogólnych założeń w formie konstrukcji językowych, typów danych i wbudowanych bezpośrednio w język mechanizmów (np. wywołania *watchdog*, komunikacja przez wymianę wiadomości) w stopniu na tyle dokładnym, aby możliwa była ich implementacja.

Marek Marecki zaprojektował, a następnie zaimplementował kompilator języka ViuAct – od podsystemów odpowiedzialnych za analizę leksykalną i składniową, do podsystemu emitującego kod wynikowy w języku assemblera Viua VM. Elementem implementacji kompilatora języka ViuAct było również opracowanie sposobów alokacji rejestrów (przydzielenia rejestru każdej wartości nazwanej - tj. będącej operandem dowiązania *let* - i anonimowej), tłumaczenia konstrukcji językowych mających swoje bezpośrednie odpowiedniki w języku assemblera Viua VM (np. dowiązania *let* czy wyrażenia warunkowe), oraz redukcji konstrukcji, które nie posiadały bezpośrednich odpowiedników (np. wyliczenia).

W gestii Marka Mareckiego była również implementacja biblioteki standardowej języka ViuAct oraz pozostałych bibliotek potrzebnych Krzysztofowi Frankowi w pracach nad czatem. Jedną z tych pozostałych bibliotek jest biblioteka `plain-websocket` (w języku ViuAct dostępna jako moduł `Websocket`) opisana w dodatku C na stronie 175.

Jako zadanie poboczne powstał skrypt `viuact-bootstrap.sh`, którego zadaniem jest tworzenie jednolitego środowiska do pracy na językiem ViuAct i programem ViuaChat. Sposób użycia skryptu jest opisany we wstępie do rozdziału 3. Przebieg prac na stronie 13.

Dodatkowym narzędziemy wytworzonym na potrzeby utrzymania jednolitego stylu pisanych programów jest `viuact-format` czyli program automatycznie formatujący kod źródłowy programów napisanych w języku ViuAct.

Bibliografia

[1] AMD, "Ryzen threadripper." `https://www.amd.com/en/products/ryzen-threadripper`, 2018. Dostęp 2019-04-11.

[2] dr hab. Jadwiga Linde-Usienkiewicz, *Wielki słownik angielsko-polski*. Wydawnictwo Naukowe PWN SA, 2011.

[3] K. Schwaber and J. Sutherland, *The Scrum Guide™. The Definitive Guide to Scrum: The Rules of the Game*. 2017.

[4] M. L. Scott, *Programming language pragmatics*. Morgan Kaufmann Publishers, 2000.

[5] D. Patterson and A. Waterman, *The RISC-V reader: an open architecture atlas*. Strawberry Canyon LLC, 2017.

[6] J. R. Levine, *Linkers and loaders*. Morgan Kaufmann Publishers, 2000.

[7] A. S. Tannenbaum and H. Bos, *Modern Operating Systems (4th Edition)*. HELION S.A., 2016.

[8] A. A. A. Donovan and B. W. Kernighan, *The Go Programming Language*. HELION S.A., 2016.

Część V

Załączniki

Dodatek A

Język assemblera Viua VM

W tym rozdziale zostanie omówiony język assemblera Viua VM – jego składnia, dostępne dyrektywy, oraz instrukcje. Wyjaśnione zostaną pojęcia takie jak „adres rejestru" i „tryby dostępu". Zostaną przedstawione istniejące i możliwe do wykorzystania zestawy rejestrów.

W dalszej części rozdziału (na stronie 146) przedstawiona zostanie dokumentacja zestawu instrukcji jakie oferuje programistom Viua VM.

A.1 Składnia języka assemblera

Składnia języka assemblera Viua VM jest prosta. Wyglądem przypomina składnię innych języków tego rodzaju (np. języki assemblera x86 czy ARM).

A.1.1 Ogólna składnia instrukcji

Instrukcje składają się z mnemoniki, zera lub więcej adresów rejestrów, i co najwyżej jednego literału. Ogólną składnię można zapisać więc tak:

```
mnemonic [<register>...] [<literal>]
```

Zanim pokazanych zostanie kilka przykładów ilustrujących różne warianty składni na konkretnych instrukcjach należy zdefiniować czym jest „adres rejestru" (lub, skrótowo, „rejestr"), a czym jest „literał".

adres rejestru identyfikator informujący kernel VM skąd ma pobrać wartości do przetworzenia i gdzie zapisać wyniki działania instrukcji, w specjalnym przypadku adresem rejestru jest tzw. „*rejestr pusty*" – `void`

literał reprezentacja wartości wpisana dosłownie w kod źródłowy lub binarny, np. `0xdeadbeef`, `"Hello World!"`, `42`; do literałów zaliczane są też nazwy funkcji, bloków i modułów

A.1.1.1 Adresy rejestrów

```
<access-operator> <index> <register-set>
void
```

`access-operator` `%` – „dostęp bezpośredni", lub `*` – „dereferencja wskaźnika"

`index` indeks rejestru wewnątrz zestawu rejestrów

`register-set` nazwa zestawu rejestrów

Dostępne zestawy rejestrów ogólnego przeznaczenia to:

`local` zawiera wartości lokalne („zmienne lokalne")

`static` zawiera wartości statyczne („zmienne statyczne”)

`arguments` zawiera wartości przekazane do aktywnej funkcji jako argumenty

`parameters` zawiera wartości przekazane do przygotowywanej ramki wywołania jako parametry

Pseudo-zestawy rejestrów specjalnego przeznaczenia to:

exception register rejestr przechowujący aktywny wyjątek; zapis do tego rejestru odbywa się za pomocą instrukcji `throw`, a odczyt z niego za pomocą instrukcji `draw`

message queue kolejka wiadomości procesu; zapis do niej odbywa się za pomocą instrukcji `send`, a odczyt z niej za pomocą instrukcji `receive` (jest to jedyny „zestaw rejestrów”, który może być modyfikowany przez procesy nie będące jego „właścicielami”)

Podczas odczytu wartości adres `%3 static` powoduje „dostęp do wartości w 3. statycznym rejestrze”; `*4 local` powoduje „dostęp do wartości, na którą wskazuje wskaźnik w 4. lokalnym rejestrze”.

Wartości w rejestrach z zestawów `parameters` i `arguments` nie mogą być modyfikowane bezpośrednio. Muszą być najpierw przeniesione lub skopiowane do rejestrów lokalnych lub statycznych.

A.1.2 Definicje funkcji, domknięć i bloków

A.1.2.1 Funkcje

Pierwszą instrukcją w funkcji zawsze *musi* być `allocate_registers`.

```
.function: <name>/<arity>
    allocate_registers
        <instruction>
        [<instruction>...]
.end
```

Przykład:

```
.function: main/0
    allocate_registers %2 local

    text %1 local "Hello, World!"
    print %1 local

    izero %0 local
    return
.end
```

A.1.2.2 Domknięcia

W przeciwieństwie do funkcji, pierwszą instrukcją w definicji domknięcia *nie może* być instrukcja `allocate_registers`. Zestaw lokalnych rejestrów dla domknięcia jest tworzony przez instrukcję `closure` w momencie konstrukcji instacji domknięcia.

```
.closure: <name>/<arity>
        <instruction>
        [<instruction>...]
.end
```

Przykład:

```
.closure: print_whatever_was_captured/0
    print %1 local
    return
.end
```

A.1.2.3 Bloki

```
.block: <name>
        <instruction>
        [<instruction>...]
.end
```

Przykład:

```
.block: handle_an_exception
    draw void
    text %1 local "An error occured."
    print %1 local
    leave
.end
```

A.1.3 Deklaracje funkcji

```
.signature: <name>/<arity>
```

A.1.4 Import modułów

```
.import: <module>[::<module>...]
```

A.1.5 Markery

```
.mark: <marker-name>
```

A.1.6 Nazywanie rejestrów

```
.name: <index> <name>
```

A.2 Instrukcje Viua VM

A.2.1 Zarządzanie wartościami

A.2.1.1 `move`

```
move Rd Rs
```

Opis Przesuwa wartości między rejestrami. Jeśli w rejestrze *`Rd`* przed wykonaniem instrukcji znajduje się jakaś wartość to będzie ona zniszczona. Po wykonaniu instrukcji wartość, która znajdowała się w rejestrze *`Rs`* znajduje się w rejestrze *`Rd`*, a rejestr *`Rs`* jest pusty.

```
move %0 local %4 local                  1
move %0 parameters %1 local             2
move %1 local %1 static                 3
```

Instrukcja z przykładu *1* przesuwa wartość z lokalnego rejestru 4 do lokalnego rejstru 0. Instrukcja z przykładu *2* przesuwa wartość z lokalnego rejestru 1 do rejestru parametrów 0 (znajdującego się w ramce wywołania). Instrukcja z przykładu *3* przesuwa wartość ze statycznego rejestru 1 do lokalnego rejestru 1.

Uwagi – przekazywanie parametrów Instrukcja `copy` jest wykorzystywana do przekazania parametrów formalnych przez przeniesienie i przy odczycie parametrów faktycznych. Przy przekazaniu parametrów wynikowym zestawem rejestrów musi być `arguments`. Przy odczycie parametrów źródłowym zestawem rejestrów musi być `parameters`.

```
move %0 arguments %1 local      ; przekazanie   1
move %1 local %0 parameters     ; odczyt        2
```

A.2.1.2 `copy`

```
copy Rd Rs
```

Opis Kopiuje wartość z rejestru *`Rs`* do rejestru *`Rd`*. Jeśli w rejestrze *`Rd`* przed wykonaniem instrukcji znajduje się jakaś wartość to będzie ona zniszczona. Po wykonaniu instrukcji wartość w rejestrze *`Rs`* pozostaje nienaruszona.

```
copy %0 local %4 local                  1
copy %0 parameters %1 local             2
copy %1 local %1 static                 3
```

Instrukcja z przykładu *1* kopiuje wartość z lokalnego rejestru 4 do lokalnego rejstru 0. Instrukcja z przykładu *2* kopiuje wartość z lokalnego rejestru 1 do rejestru parametrów 0 (znajdującego się w ramce wywołania). Instrukcja z przykładu *3* kopiuje wartość ze statycznego rejestru 1 do lokalnego rejestru 1.

Uwagi – przekazywanie parametrów Instrukcja `copy` jest wykorzystywana do przekazania parametrów formalnych przez kopię do funkcji. Przy przekazaniu parametrów wynikowym zestawem rejestrów musi być `arguments`. Przy odczycie parametrów

```
copy %0 arguments %1 local              1
```

A.2.1.3 `ptr`

```
ptr Rd Rs
```

Opis Konstruuje w rejestrze *`Rd`* wskaźnik do wartości znajdującej się w rejestrze *`Rs`*.

```
ptr %1 local %2 local
```

Uwagi – przekazywanie wiadomości Wskaźniki są ważne jedynie wewnątrz procesu, w którym zostały utworzone. Jeśli zostaną przekazane do innego procesu jako wiadomość i zostaną rozwiązane w innym procesie zgłoszony zostanie wyjątek.

A.2.1.4 `ptrlive`

```
ptrlive Rd Rs
```

Opis Sprawdza poprawność (ważność) wskaźnika w rejestrze *`Rs`*. W rejestrze *`Rd`* umieszcza wartość logiczną `true` jeśli wskaźnik wskazuje na „żywą” (niezniszczoną) wartość, lub `false` jeśli wskaźnik wskazuje na „martwą” (zniszczoną) wartość.

```
ptrlive %1 local %2 local
```

A.2.1.5 `swap`

```
swap Ra Rb
```

Opis Zamienia miejscami wartości w rejestrach *`Ra`* i *`Rb`*.

```
swap %1 local %2 local
```

A.2.1.6 `delete`

```
delete Rt
```

Opis Niszczy wartość w rejestrze *`Rt`*.

```
        izero %0 local
        return
.end
```

Wywołanie destruktora wartości.

A.2.1.7 `isnull`

```
izero Rd
```

Opis Konstruuje w rejestrze `Rd` liczbę całkowitą o wartości 0. Zwyczajowo wykorzystywana na końcu funkcji `main` do utworzenia domyślnej wartości zwracanej:

```
        izero %0 local
        return
.end
```

A.2.2 Operacje logiczne

A.2.2.1 `not`

```
not Rt
```

Opis Negacja boolowska wartości w rejestrze `Rt`.

Po wykonaniu instrukcji w rejestrze `Rt` znajduje się wartość boolowska reprezentująca negację wartości otrzymanej po konwersji wartości znajdującej się poprzednio w rejestrze `Rt` na typ boolowski.

A.2.2.2 `and`

```
and Rd Ra Rb
```

Opis Iloczyn boolowski.

Po wykonaniu instrukcji w rejestrze `Rd` znajduje się wartość boolowska reprezentująca iloczyn wartości boolowskich powstałych z konwersji wartości znajdujących się przed wykonaniem instrukcji w rejestrach `Ra` i `Rb`.

Wartości w rejestrach `Ra` i `Rb` pozostają nienaruszone.

A.2.2.3 `or`

```
or Rd Ra Rb
```

Opis Suma boolowska.

Po wykonaniu instrukcji w rejestrze `Rd` znajduje się wartość boolowska reprezentująca sumę wartości boolowskich powstałych z konwersji wartości znajdujących się przed wykonaniem instrukcji w rejestrach `Ra` i `Rb`.

Wartości w rejestrach `Ra` i `Rb` pozostają nienaruszone.

A.2.3 Operacje bitowe

A.2.3.1 `bits_of_integer`

```
bits_of_integer Rd Rs
```

Opis Konstruktor bitów z liczby całkowitej.

Dokonuje konwersji liczby całkowitej znajdującej się w rejestrze `Rs` na ciąg bitów o długości 64. Wynik konwersji jest zapisywany w rejestrze `Rt`.

Wartość w rejestrze `Rs` pozostaje niezmieniona.

```
integer %1 local 42
bits_of_integer %1 local %1 local
print %1 local
```

Powyższy przykład spowoduje wydrukowanie na ekran następującego ciągu bitów: `00000000000000000000000000000000000000000000000000000000000101010`.

A.2.3.2 `integer_of_bits`

```
integer_of_bits Rd Rs
```

Opis Konstruktor liczby całkowitej z bitów.

Dokonuje konwersji ciągu bitów znajdującej się w rejestrze `Rs` na liczbę całkowitą. Wynik konwersji jest zapisywany w rejestrze `Rt`.

Wartość w rejestrze `Rs` pozostaje niezmieniona.

```
bits %1 local 0b0101010
integer_of_bits %1 local %1 local
print %1 local
```

Powyższy przykład spowoduje wydrukowanie na ekran liczby 42.

Uwagi Konwersja ciągów dłuższych niż 64 bity powinna spowodować błąd. Jednak nie jest to weryfikowane i obecna wersja maszyny „po cichu" powoduje zawinięcie liczby.

Ciągi krótsze niż 64 bity przed konwersją są rozszerzane do długości 64 bitów. Na najbardziej znaczące pozycje wpisywane są zera, co może spowodować produkować „zaskakujące" wyniki, gdzie małe liczby ujemne są konwertowane na małe liczby dodatnie.

Jeśli wymagane są częste konwersje między ciągami bitów, a sprzętowymi liczbami całkowitymi zaleca się używanie ciągów bitów o długości 64 (odpowiadającej szerokości bitowej sprzętowej liczby całkowitej). Jeśli konwersje nie są częste, rozszerzenia ze znakiem (ang. *sign extension*) można dokonać za pomocą następującej sekwencji instrukcji (dla liczby reprezentowanej przez ciąg o długości 16 bitów):

```
.name: 1 se      ; „sign extended"
.name: 2 input   ; wejściowy ciąg bitów, do rozszerzenia
.name: 3 n       ; wartość pomocnicza

integer %se local 64
bits    %se local %se local
bitor   %se local %se local %input local

integer %n local 48
shl     void %se local %n local
ashr    void %se local %n local
```

Po wykonaniu tej sekwencji instrukcji w rejestrze `se` znajduje się rozszerzona wartość wejściowa.

A.2.3.3 `bits`

```
bits Rd 0x<hex-digits>
bits Rd 0o<octal-digits>
bits Rd 0b<binary-digits>
bits Rd Rs
```

Opis Konstruktor bitów.

W wariantach 1-3 konstruuje ciąg bitów z literału podanego jako drugi operand. W wariancie 4 konstruuje ciąg bitów o długości podanej w rejestrze `Rs`. We wszystkich wariantach wynikowy ciąg bitów zapisywany jest w rejestrze `Rd`.

```
bits %1 local 0xdeadbeef
bits %2 local 0o1337
bits %3 local 0b01

integer %4 local 8
bits %4 local %4 local
```

A.2.3.4 bitand

```
bitand Rd Ra Rb
```

Opis Bitowa operacja *and*. Długość wynikowego ciągu bitów jest równa długości ciągu, który w znadował się w rejestrze `Ra`.

A.2.3.5 bitor

```
bitor Rd Ra Rb
```

Opis Bitowa operacja *or*. Długość wynikowego ciągu bitów jest równa długości ciągu, który w znadował się w rejestrze `Ra`.

A.2.3.6 bitnot

```
bitxor Rd Rs
```

Opis Bitowa operacja *not*. Konstruuje nowy ciąg bitów będący negacją ciągu bitów znajdującego się w rejestrze `Rs`.

A.2.3.7 bitxor

```
bitxor Rd Ra Rb
```

Opis Bitowa operacja *xor*. Długość wynikowego ciągu bitów jest równa długości ciągu, który w znadował się w rejestrze `Ra`.

A.2.3.8 bitat

```
bitat Rd Rs Ri
```

Opis Sprawdzenie wartości pojedynczego bitu. Po wykonaniu instrukcji w rejestrze `Rd` znajduje się wartość boolowska określająca czy w ciągu bitów znajdującym się w rejestrze `Rs` bit pod indeksem określonym przez liczbę całkowitą znajdującą się w rejestrze `Ri` jest włączony czy wyłączony.

```
integer %1 local 0
bitat %3 local %2 local %1 local
print %3 local

integer %1 local 1
bitat %3 local %2 local %1 local
print %3 local
```

Powyższy przykład wydrukuje na ekran najpierw `true`, a potem `false`.

A.2.3.9 `bitset`

```
bitset Rt Ri <boolean-literal>
```

Opis Ustawienie wartości pojedynczego bitu. Po wykonaniu instrukcji w ciągu bitów znajdującym się w rejestrze `Rt` bit znajdujący się pod indeksem określonym przez liczbę całkowitą znajdującą się w rejestrze `Ri` jest (1) włączony jeśli *<boolean-literal>* był wartością `true` (2) wyłączony jeśli *<boolean-literal>* był wartością `fale`

A.2.3.10 `shl`

```
shl Ro Rt Rn
```

Opis Przesunięcie bitowe w lewo.

Przesuwa ciąg bitów w rejestrze `Rt` o ilość bitów określoną przez liczbę całkowitą znajdującą się w rejestrze `Rn`. Wartość w rejestrze `Rt` jest modyfikowana *w miejscu*, a w rejestrze `Ro` umieszczany jest „wysunięty" ciąg bitów.

```
bits %3 local 0x4a
print %3 local
integer %4 local 4
shl %4 local %3 local %4 local
print %3 local
print %4 local
```

Powyższy przykład spowoduje wydrukowanie na ekranie następujących wartości:

```
01001010
10100000
0100
```

A.2.3.11 `shr`

```
shr Rd Rs Rn
```

Opis Przesunięcie bitowe w prawo.

Przesuwa ciąg bitów w rejestrze `Rt` o ilość bitów określoną przez liczbę całkowitą znajdującą się w rejestrze `Rn`. Wartość w rejestrze `Rt` jest modyfikowana *w miejscu*, a w rejestrze `Ro` umieszczany jest „wysunięty" ciąg bitów.

```
bits %3 local 0x4a
print %3 local
integer %4 local 4
shr %4 local %3 local %4 local
print %3 local
print %4 local
```

Powyższy przykład spowoduje wydrukowanie na ekranie następujących wartości:

```
01001010
00000100
1010
```

A.2.3.12 `ashl`

```
ashl Rd Rs Rn
```

Opis Arytmetyczne (z zachowaniem znaku) przesunięcie bitowe w lewo.

Przesuwa ciąg bitów w rejestrze `Rt` o ilość bitów określoną przez liczbę całkowitą znajdującą się w rejestrze `Rn`. Wartość w rejestrze `Rt` jest modyfikowana *w miejscu*, a w rejestrze `Ro` umieszczany jest „wysunięty" ciąg bitów.

```
bits %3 local 0xa4
print %3 local
integer %4 local 4
ashl %4 local %3 local %4 local
print %3 local
print %4 local
```

Powyższy przykład spowoduje wydrukowanie na ekranie następujących wartości:

```
10100100
11000000
1010
```

A.2.3.13 `ashr`

```
ashr Rd Rs Rn
```

Opis Arytmetyczne przesunięcie bitowe w prawo.

Przesuwa ciąg bitów w rejestrze `Rt` o ilość bitów określoną przez liczbę całkowitą znajdującą się w rejestrze `Rn`. Wartość w rejestrze `Rt` jest modyfikowana *w miejscu*, a w rejestrze `Ro` umieszczany jest „wysunięty" ciąg bitów.

```
bits %3 local 0xa4
print %3 local
integer %4 local 4
ashr %4 local %3 local %4 local
print %3 local
print %4 local
```

Powyższy przykład spowoduje wydrukowanie na ekranie następujących wartości:

```
10100100
11111010
0100
```

A.2.3.14 `rol`

```
rol Rt Rn
```

Opis Rotacja bitowa w lewo.

„Obraca" ciąg bitów w rejestrze `Rt` o ilość bitów określoną przez liczbę całkowitą znajdującą się w rejestrze `Rn`. Wartość w rejestrze `Rt` jest modyfikowana *w miejscu*.

```
bits %3 local 0x8e
integer %4 local 3
print %3 local
rol %3 local %4 local
print %3 local
```

Powyższy przykład spowoduje wydrukowanie na ekranie następujących wartości:

```
10001110
01110100
```

A.2.3.15 `ror`

```
ror Rt Rn
```

Opis Rotacja bitowa w prawo.

„Obraca" ciąg bitów w rejestrze `Rt` o ilość bitów określoną przez liczbę całkowitą znajdującą się w rejestrze `Rn`. Wartość w rejestrze `Rt` jest modyfikowana *w miejscu*.

```
bits %3 local 0x8e
integer %4 local 3
print %3 local
ror %3 local %4 local
print %3 local
```

Powyższy przykład spowoduje wydrukowanie na ekranie następujących wartości:

```
10001110
11010001
```

A.2.4 Arytmetyka (CPU)

Arytmetyka implementowana przez fizyczne CPU, na którym działa Viua VM. Specyfika tego fizycznego CPU wpływa na wyniki działania instrukcji z tej grupy. Arytmetyka o zdefiniowanym zachowaniu niezależnym od fizycznej platformy jest opisana w rozdziale A.2.5 na stronie 156. Arytmetyka oparta o CPU jest bardziej wydajna („szybsza"), ale nie zawsze zapewnia przewidywalność, stabliność, i weryfikację wyników operacji (wykrywanie błędów).

A.2.4.1 `izero`

```
izero Rd
```

Opis Konstruuje w rejestrze `Rd` liczbę całkowitą o wartości 0. Zwyczajowo wykorzystywana na końcu funkcji `main` do utworzenia domyślnej wartości zwracanej:

```
        izero %0 local
        return
.end
```

A.2.4.2 `integer`

```
integer Rd <integer>
```

Opis Konstruuje w rejestrze `Rd` liczbę całkowitą o wartości *`integer`*.

A.2.4.3 `iinc`

```
iinc Rt
```

Opis Zwiększa wartość liczby całkowitej w rejestrze `Rt` o 1.

Uwagi Zachowanie w przypadku przepełnienia jest zdefiniowane przez zachowanie platformy sprzętowej.

A.2.4.4 `idec`

```
idec Rt
```

Opis Zmniejsza wartość liczby całkowitej w rejestrze `Rt` o 1.

Uwagi Zachowanie w przypadku przepełnienia jest zdefiniowane przez zachowanie platformy sprzętowej.

A.2.4.5 `float`

```
float Rd <float>
```

Opis Konstruuje w rejestrze `Rd` liczbę zmiennoprzecinkową o wartości *`float`*.

A.2.4.6 `itof`

```
itof Rd Rs
```

Opis Konwertuje wartość całkowitoliczbową z rejestru `Rs` na wartość zmiennoprzecinkową i umieszcza wynik w rejestrze `Rd`.

A.2.4.7 `ftoi`

```
ftoi Rd Rs
```

Opis Konwertuje wartość zmiennoprzecinkową z rejestru `Rs` na wartość całkowitoliczbową i umieszcza wynik w rejestrze `Rd`.

A.2.4.8 `stoi`

```
stoi Rd Rs
```

Opis Konwertuje string z rejestru `Rs` na wartość całkowitoliczbową i umieszcza wynik w rejestrze `Rd`.

A.2.4.9 `stof`

```
stof Rd Rs
```

Opis Konwertuje string z rejestru `Rs` na wartość zmiennoprzecinkową i umieszcza wynik w rejestrze `Rd`.

A.2.4.10 `add`

```
add Rd Ra Rb
```

Opis Dodaje dwie wartości liczbowe (całkowitoliczbową lub zmiennoprzecinkową) znajdujące się w rejestrach `Ra` i `Rb` do siebie, a wynik umieszcza w rejestrze `Rd`.

Wartość umieszczona w rejestrze `Rd` ma taki sam typ jak wartość w rejestrze `Ra`. Wartości w rejestrach `Ra` i `Rb` pozostają nienaruszone.

A.2.4.11 `sub`

```
sub Rd Ra Rb                                                    1
```

Opis Odejmuje dwie wartości liczbowe (całkowitoliczbową lub zmiennoprzecinkową) znajdujące się w rejestrach `Ra` i `Rb` do siebie, a wynik umieszcza w rejestrze `Rd`.

Wartość umieszczona w rejestrze `Rd` ma taki sam typ jak wartość w rejestrze `Ra`. Wartości w rejestrach `Ra` i `Rb` pozostają nienaruszone.

A.2.4.12 `mul`

```
mul Rd Ra Rb                                                    1
```

Opis Mnoży dwie wartości liczbowe (całkowitoliczbową lub zmiennoprzecinkową) znajdujące się w rejestrach `Ra` i `Rb`, a wynik umieszcza w rejestrze `Rd`. Działanie jest przeprowadzane zgodnie ze wzorem $a * b$.

Wartość umieszczona w rejestrze `Rd` ma taki sam typ jak wartość w rejestrze `Ra`. Wartości w rejestrach `Ra` i `Rb` pozostają nienaruszone.

A.2.4.13 `div`

```
add Rd Ra Rb                                                    1
```

Opis Dzieli dwie wartości liczbowe (całkowitoliczbową lub zmiennoprzecinkową) znajdujące się w rejestrach `Ra` i `Rb`, a wynik umieszcza w rejestrze `Rd`. Działanie jest przeprowadzane zgodnie ze wzorem a/b.

Wartość umieszczona w rejestrze `Rd` ma taki sam typ jak wartość w rejestrze `Ra`. Wartości w rejestrach `Ra` i `Rb` pozostają nienaruszone.

A.2.4.14 `lt`

```
lt Rd Ra Rb                                                     1
```

Opis Porównuje dwie wartości liczbowe (całkowitoliczbową lub zmiennoprzecinkową) znajdujące się w rejestrach `Ra` i `Rb`, a wynik umieszcza w rejestrze `Rd`. Działanie jest przeprowadzane zgodnie ze wzorem $a < b$.

Wartość wynika jest typu boolowskiego.

A.2.4.15 `lte`

```
lte Rd Ra Rb                                                    1
```

Opis Porównuje dwie wartości liczbowe (całkowitoliczbową lub zmiennoprzecinkową) znajdujące się w rejestrach `Ra` i `Rb`, a wynik umieszcza w rejestrze `Rd`. Działanie jest przeprowadzane zgodnie ze wzorem $a <= b$.

Wartość wynika jest typu boolowskiego.

A.2.4.16 `gt`

```
gt Rd Ra Rb
```

Opis Porównuje dwie wartości liczbowe (całkowitoliczbową lub zmiennoprzecinkową) znajdujące się w rejestrach `Ra` i `Rb`, a wynik umieszcza w rejestrze `Rd`. Działanie jest przeprowadzane zgodnie ze wzorem $a > b$.

Wartość wynika jest typu boolowskiego.

A.2.4.17 `gte`

```
gte Rd Ra Rb
```

Opis Porównuje dwie wartości liczbowe (całkowitoliczbową lub zmiennoprzecinkową) znajdujące się w rejestrach `Ra` i `Rb`, a wynik umieszcza w rejestrze `Rd`. Działanie jest przeprowadzane zgodnie ze wzorem $a >= b$.

Wartość wynika jest typu boolowskiego.

A.2.4.18 `eq`

```
eq Rd Ra Rb
```

Opis Porównuje dwie wartości liczbowe (całkowitoliczbową lub zmiennoprzecinkową) znajdujące się w rejestrach `Ra` i `Rb`, a wynik umieszcza w rejestrze `Rd`. Działanie jest przeprowadzane zgodnie ze wzorem $a = b$.

Wartość wynika jest typu boolowskiego.

A.2.5 Arytmetyka (Viua VM)

Arytmetyka implementowana przez Viua VM, niezależna od fizycznej platformy, na której maszyna wirtualna jest uruchomiona. Arytmetyka o zachowaniu zależnym od fizycznej platformy jest opisana w rozdziale A.2.4 na stronie 153. Arytmetyka o zachowaniu niezależnym od fizycznej platformy jest mniej wydajna („wolniesza"), ale zapewnia przewidywalność, stabliność, i weryfikację wyników operacji (wykrywanie błędów).

Operandami w instrukcjach arytmetycznych implementowanych przez VM są ciągi bitów, których konstruktory są opisane w rozdziale A.2.3 na stronie 148.

A.2.5.1 `wrapincrement`

```
wrapincrement Rt
```

Opis Inkrementacja modulo (ze znakiem).

Instrukcja interpretuje ciąg bitów jako liczbę całkowitą w kodowaniu z dopełnieniem do dwóch[1] o szerokości równej długości ciągu bitów, na którym instrukcja operuje.

[1] Kodowanie z dopełnieniem do dwóch (ang. *two's complement*)

Uwagi W przypadku wystąpienia przepełnienia największa wartość dodatnia staje się największą (tj. najbardziej oddaloną od zera) wartością ujemną.

```
bits %1 local 0b01111111
print %1 local
wrapincrement %1 local
print %1 local
```

Ma to zastosowanie jedynie jeśli ciągi bitów są interpretowane przy konwersji jako liczby ze znakiem.

A.2.5.2 `wrapdecrement`

```
wrapdecrement Rt
```

Opis Dekrementacja modulo (ze znakiem).

Instrukcja interpretuje ciąg bitów jako liczbę całkowitą w kodowaniu z dopełnieniem do dwóch o szerokości równej długości ciągu bitów, na którym instrukcja operuje.

Uwagi W przypadku wystąpienia przepełnienia największa (tj. najbardziej oddalona od zera) wartość ujemna staje się największą wartością dodatnią.

```
bits %1 local 0x80
print %1 local
wrapdecrement %1 local
print %1 local
```

Ma to zastosowanie jedynie jeśli ciągi bitów są interpretowane przy konwersji jako liczby ze znakiem.

A.2.5.3 `wrapadd`

```
wrapadd Rd Ra Rb
```

Opis Dodawanie modulo.

Instrukcja interpretuje liczby w rejestrach `Ra` i `Rb` jaki liczby całkowite ze znakiem w kodowaniu z dopełnieniem do dwóch. Wynikiem instrukcji jest ciąg bitów reprezentujący liczbe całkowitą ze znakiem w kodowaniu z dopełnieniem do dwóch, o długości odpowiadającej długości ciągu bitów znajdującemu się w rejestrze `Ra`.

Operacja jest wykonywana według wzoru $a + b$.

Uwagi W przypadku przepełnienia wartość jest „zawijana” i wynik z dodatniego może stać się ujemy i *vice versa*.

A.2.5.4 `wrapsub`

```
wrapsub Rd Ra Rb
```

Opis Odejmowanie modulo.

Instrukcja interpretuje liczby w rejestrach `Ra` i `Rb` jaki liczby całkowite ze znakiem w kodowaniu z dopełnieniem do dwóch. Wynikiem instrukcji jest ciąg bitów reprezentujący liczbe całkowitą ze znakiem w kodowaniu z dopełnieniem do dwóch, o długości odpowiadającej długości ciągu bitów znajdującemu się w rejestrze `Ra`.

Operacja jest wykonywana według wzoru $a - b$.

Uwagi W przypadku przepełnienia wartość jest „zawijana” i wynik z dodatniego może stać się ujemy i *vice versa*.

A.2.5.5 `wrapmul`

```
wrapmul Rd Ra Rb
```

Opis Mnożenie modulo.

Instrukcja interpretuje liczby w rejestrach `Ra` i `Rb` jaki liczby całkowite ze znakiem w kodowaniu z dopełnieniem do dwóch. Wynikiem instrukcji jest ciąg bitów reprezentujący liczbe całkowitą ze znakiem w kodowaniu z dopełnieniem do dwóch, o długości odpowiadającej długości ciągu bitów znajdującemu się w rejestrze `Ra`.

Operacja jest wykonywana według wzoru $a * b$.

Uwagi W przypadku przepełnienia wartość jest „zawijana” i wynik z dodatniego może stać się ujemy i *vice versa*.

A.2.5.6 `wrapdiv`

```
wrapdiv Rd Ra Rb
```

Opis Dzielenie modulo.

Instrukcja interpretuje liczby w rejestrach `Ra` i `Rb` jaki liczby całkowite ze znakiem w kodowaniu z dopełnieniem do dwóch. Wynikiem instrukcji jest ciąg bitów reprezentujący liczbe całkowitą ze znakiem w kodowaniu z dopełnieniem do dwóch, o długości odpowiadającej długości ciągu bitów znajdującemu się w rejestrze `Ra`.

Operacja jest wykonywana według wzoru a/b.

Uwagi W przypadku przepełnienia wartość jest „zawijana” i wynik z dodatniego może stać się ujemy i *vice versa*.

A.2.5.7 `checkedsincrement`

```
checkedsincrement Rt
```

Opis Inkrementacja weryfikowana ze znakiem.

Instrukcja interpretuje ciąg bitów jako liczbę całkowitą w kodowaniu z dopełnieniem do dwóch o szerokości równej długości ciągu bitów, na którym instrukcja operuje.

Uwagi W przypadku przepełnienia zgłaszany jest wyjątek.

A.2.5.8 `checkedsdecrement`

```
checkedsdecrement Rt
```

Opis Inkrementacja weryfikowana ze znakiem.

Instrukcja interpretuje ciąg bitów jako liczbę całkowitą w kodowaniu z dopełnieniem do dwóch o szerokości równej długości ciągu bitów, na którym instrukcja operuje.

Uwagi W przypadku przepełnienia zgłaszany jest wyjątek.

A.2.5.9 `checkedsadd`

```
checkedsadd Rd Ra Rb
```

Opis Dodawanie weryfikowane ze znakiem.

Instrukcja interpretuje liczby w rejestrach `Ra` i `Rb` jaki liczby całkowite ze znakiem w kodowaniu z dopełnieniem do dwóch. Wynikiem instrukcji jest ciąg bitów reprezentujący liczbe całkowitą ze znakiem w kodowaniu z dopełnieniem do dwóch, o długości odpowiadającej długości ciągu bitów znajdującemu się w rejestrze `Ra`.

Operacja jest wykonywana według wzoru $a + b$.

Uwagi W przypadku przepełnienia zgłaszany jest wyjątek.

A.2.5.10 `checkedssub`

```
checkedssub Rd Ra Rb
```

Opis Odejmowanie weryfikowane ze znakiem.

Instrukcja interpretuje liczby w rejestrach `Ra` i `Rb` jaki liczby całkowite ze znakiem w kodowaniu z dopełnieniem do dwóch. Wynikiem instrukcji jest ciąg bitów reprezentujący liczbe całkowitą ze znakiem w kodowaniu z dopełnieniem do dwóch, o długości odpowiadającej długości ciągu bitów znajdującemu się w rejestrze `Ra`.

Operacja jest wykonywana według wzoru $a - b$.

Uwagi W przypadku przepełnienia zgłaszany jest wyjątek.

A.2.5.11 `checkedsmul`

```
checkedsmul Rd Ra Rb
```

Opis Dzielenie weryfikowane ze znakiem.

Instrukcja interpretuje liczby w rejestrach `Ra` i `Rb` jaki liczby całkowite ze znakiem w kodowaniu z dopełnieniem do dwóch. Wynikiem instrukcji jest ciąg bitów reprezentujący liczbe całkowitą ze znakiem w kodowaniu z dopełnieniem do dwóch, o długości odpowiadającej długości ciągu bitów znajdującemu się w rejestrze `Ra`.

Operacja jest wykonywana według wzoru $a * b$.

Uwagi W przypadku przepełnienia zgłaszany jest wyjątek.

A.2.5.12 `checkedsdiv`

```
checkedsdiv Rd Ra Rb
```

Opis Dzielenie weryfikowane ze znakiem.

Instrukcja interpretuje liczby w rejestrach `Ra` i `Rb` jaki liczby całkowite ze znakiem w kodowaniu z dopełnieniem do dwóch. Wynikiem instrukcji jest ciąg bitów reprezentujący liczbe całkowitą ze znakiem w kodowaniu z dopełnieniem do dwóch, o długości odpowiadającej długości ciągu bitów znajdującemu się w rejestrze `Ra`.

Operacja jest wykonywana według wzoru a/b.

Uwagi W przypadku przepełnienia zgłaszany jest wyjątek.

A.2.5.13 `saturatingsincrement`

```
saturatingsincrement Rt
```

Opis Inkrementacja nasyceniowa ze znakiem.

Instrukcja interpretuje ciąg bitów jako liczbę całkowitą w kodowaniu z dopełnieniem do dwóch o szerokości równej długości ciągu bitów, na którym instrukcja operuje.

Uwagi W przypadku wykrycia przepełnienia następuje nasycenie i wynik jest wartością maksymalną dla liczby całkowitej ze znakiem w kodowaniu do dwóch o szerokości równej długości ciągu bitów użytego jako operand instrukcji.

A.2.5.14 `saturatingsdecrement`

```
saturatingsdecrement Rt
```

Opis Dekrementacja nasyceniowa ze znakiem.

Instrukcja interpretuje ciąg bitów jako liczbę całkowitą w kodowaniu z dopełnieniem do dwóch o szerokości równej długości ciągu bitów, na którym instrukcja operuje.

Uwagi W przypadku wykrycia przepełnienia następuje nasycenie i wynik jest wartością minimalną dla liczby całkowitej ze znakiem w kodowaniu do dwóch o szerokości równej długości ciągu bitów użytego jako operand instrukcji.

A.2.5.15 `saturatingsadd`

```
saturatingsadd Rd Ra Rb
```

Opis Dodawanie nasyceniowe ze znakiem.

Instrukcja interpretuje liczby w rejestrach `Ra` i `Rb` jaki liczby całkowite ze znakiem w kodowaniu z dopełnieniem do dwóch. Wynikiem instrukcji jest ciąg bitów reprezentujący liczbe całkowitą ze znakiem w kodowaniu z dopełnieniem do dwóch, o długości odpowiadającej długości ciągu bitów znajdującemu się w rejestrze `Ra`.

Operacja jest wykonywana według wzoru $a + b$.

Uwagi W przypadku wykrycia przepełnienia następuje nasycenie i

1. jeśli wynik miał być dodatni to jest wartością maksymalną dla liczby całkowitej ze znakiem w kodowaniu do dwóch o szerokości równej długości ciągu bitów użytego jako operand instrukcji
2. jeśli wynik miał być ujemny to jest wartością minimalną dla liczby całkowitej ze znakiem w kodowaniu do dwóch o szerokości równej długości ciągu bitów użytego jako operand instrukcji

A.2.5.16 `saturatingssub`

```
saturatingssub Rd Ra Rb
```

Opis Odejmowanie nasyceniowe ze znakiem.

Instrukcja interpretuje liczby w rejestrach `Ra` i `Rb` jaki liczby całkowite ze znakiem w kodowaniu z dopełnieniem do dwóch. Wynikiem instrukcji jest ciąg bitów reprezentujący liczbe całkowitą ze znakiem w kodowaniu z dopełnieniem do dwóch, o długości odpowiadającej długości ciągu bitów znajdującemu się w rejestrze `Ra`.

Operacja jest wykonywana według wzoru $a - b$.

Uwagi W przypadku wykrycia przepełnienia następuje nasycenie i

1. jeśli wynik miał być dodatni to jest wartością maksymalną dla liczby całkowitej ze znakiem w kodowaniu do dwóch o szerokości równej długości ciągu bitów użytego jako operand instrukcji
2. jeśli wynik miał być ujemny to jest wartością minimalną dla liczby całkowitej ze znakiem w kodowaniu do dwóch o szerokości równej długości ciągu bitów użytego jako operand instrukcji

A.2.5.17 `saturatingsmul`

```
saturatingsmul Rd Ra Rb
```

Opis Mnożenie nasyceniowe ze znakiem.

Instrukcja interpretuje liczby w rejestrach `Ra` i `Rb` jaki liczby całkowite ze znakiem w kodowaniu z dopełnieniem do dwóch. Wynikiem instrukcji jest ciąg bitów reprezentujący liczbe całkowitą ze znakiem w kodowaniu z dopełnieniem do dwóch, o długości odpowiadającej długości ciągu bitów znajdującemu się w rejestrze `Ra`.

Operacja jest wykonywana według wzoru $a * b$.

Uwagi W przypadku wykrycia przepełnienia następuje nasycenie i

1. jeśli wynik miał być dodatni to jest wartością maksymalną dla liczby całkowitej ze znakiem w kodowaniu do dwóch o szerokości równej długości ciągu bitów użytego jako operand instrukcji
2. jeśli wynik miał być ujemny to jest wartością minimalną dla liczby całkowitej ze znakiem w kodowaniu do dwóch o szerokości równej długości ciągu bitów użytego jako operand instrukcji

A.2.5.18 `saturatingsdiv`

```
checkedsdiv Rd Ra Rb
```

Opis Dzielenie nasyceniowe ze znakiem.

Instrukcja interpretuje liczby w rejestrach `Ra` i `Rb` jaki liczby całkowite ze znakiem w kodowaniu z dopełnieniem do dwóch. Wynikiem instrukcji jest ciąg bitów reprezentujący liczbe całkowitą ze znakiem w kodowaniu z dopełnieniem do dwóch, o długości odpowiadającej długości ciągu bitów znajdującemu się w rejestrze `Ra`.

Operacja jest wykonywana według wzoru a/b.

Uwagi W przypadku wykrycia przepełnienia następuje nasycenie i

1. jeśli wynik miał być dodatni to jest wartością maksymalną dla liczby całkowitej ze znakiem w kodowaniu do dwóch o szerokości równej długości ciągu bitów użytego jako operand instrukcji
2. jeśli wynik miał być ujemny to jest wartością minimalną dla liczby całkowitej ze znakiem w kodowaniu do dwóch o szerokości równej długości ciągu bitów użytego jako operand instrukcji

A.2.6 Obsługa tekstu

A.2.6.1 `string`

```
string Rd "<str>"
```

Opis Konstruuje w rejestrze `Rd` ciąg bajtów o wartości *`str`*.

A.2.6.2 `streq`

```
streq Rd Ra Rb
```

Opis Weryfikuje czy ciągi bajtów w rejestarch `Ra` i `Rb` są sobie równe, a wynik sprawdzenia przechowuje w rejestrze `Rd` jako wartość typu boolowskiego (1) `true` jeśli ciągi są sobie równe (2) `false` jeśli ciągi nie są sobie równe.

A.2.6.3 `text`

```
text Rd "<txt>"
text Rd Rs
```

Opis Wariant pierwszy konstruuje w rejestrze `Rd` tekst o wartości *`txt`*. Wariant drugi konwertuje wartość z rejestru `Rs` na tekst i umieszcza go w rejestrze `Rd`.

Utworzona wartość tekstowa jest ciągiem *codepoint*-ów Unicode w kodowaniu UTF-8.

A.2.6.4 `texteq`

```
texteq Rd Ra Rb
```

Opis Weryfikuje czy ciągi bajtów w rejestarch `Ra` i `Rb` są sobie równe, a wynik sprawdzenia przechowuje w rejestrze `Rd` jako wartość typu boolowskiego (1) `true` jeśli ciągi są sobie równe (2) `false` jeśli ciągi nie są sobie równe.

A.2.6.5 `textat`

```
textat Rd Rs Rn
```

Opis Z tekstu w rejestrze `Rs` pobiera *codepoint* Unicode z indeksu określonego przez liczbę całkowitą znajdującą się w rejestrze `Rn`. Wartość w rejestrze `Rs` pozostaje nienaruszona.

A.2.6.6 `textsub`

```
textsub Rd Rs Rbegin Rend
```

Opis Z tekstu w rejestrze `Rs` kopiuje tekst pomiędzy indeksem określonym przez liczbę całkowitą znajdującą się w rejestrze `Rbegin`, a indeksem określonym przez liczbę całkowitą znajdującą się w rejestrze `Rend` bez uwzględnienia znaku pod indeksem określonym przez `Rend` – czyli zakres wycinka tekstu to $[begin, end)$.

A.2.6.7 `textlength`

```
textlength Rd Rs
```

Opis Odczytuje długość tekstu znajdującego się w rejestrze `Rs` jako liczbę całkowitą, a wynik umieszcza w rejestrze `Rd`. Długość tekstu jest określona przez liczbę *codepoint*-ów Unicode, nie liczbę bajtów.

A.2.6.8 `textcommonprefix`

```
textcommonprefix Rd Ra Rb
```

Opis Określna wspólny prefiks tekstów znajdujących się w rejestrach `Ra` i `Rb`. Wynik umieszcza w rejestrze `Rd`. Wynik jest podany jako liczba całkowita określająca długość wspólnego prefiksu w *codepoint*-ach Unicode.

A.2.6.9 `textcommonsuffix`

```
textcommonsuffix Rd Ra Rb
```

Opis Określna wspólny sufiks tekstów znajdujących się w rejestrach `Ra` i `Rb`. Wynik umieszcza w rejestrze `Rd`. Wynik jest podany jako liczba całkowita określająca długość wspólnego sufiksu w *codepoint*-ach Unicode.

A.2.6.10 `textconcat`

```
textconcat Rd Ra Rb
```

Opis Łączy teksty znajdujące się w rejestrach `Ra` i `Rb`, a wynik zapisuje w rejestrze `Rd`.

A.2.7 Wektory

A.2.7.1 `vector`

```
vector Rd
vector Rd Rp n
```

Opis Konstruuje wektor w rejestrze `Rd`.

Pierwszy wariant konstruuje pusty wektor. Drugi wariant konstruuje wektor „pakując” n wartości rozpoczynając od rejestru określonego przez `Rp` (wszystkie „spakowane” wartości pochodzą z tego samego zestawu rejestrów, określonego przez zestaw rejestrów użyty w adresie rejestru `Rp`).

Przykład pakowania:

```
integer %2 local 1
integer %3 local 2
integer %4 local 3
integer %5 local 4
vector %1 local %2 local %4
```

Wektor znajdujący się w rejestrze `%1 local` spakuje wartości z rejestrów lokalnych 2, 3, 4 i 5 (czyli będzie miał długość 4).

Uwagi Wektor nie może spakować:

1. rejestru, w którym będzie skonstruowany
2. rejestru, który nie zawiera żadnej wartości
3. rejestru, którego indeks jest poza zakresem dla zestawu rejestrów, do którego odnosi się adres rejestru `Rp`

A.2.7.2 `vinsert`

```
vinsert Rd Rs void
vinsert Rd Rs Ri
```

Opis Przenosi wartość z rejestru `Rs` do wektora znajdującego się w rejestrze `Rd`.

Pierwszy wariant przenosi wartość na początek wektora (na pozycję 0). Drugi wariant przenosi wartość na pozycję określoną przez liczbę całkowitą znajdującą się w rejestrze `Ri`.

Wyjątki Jeśli żądany indeks jest poza zakresem długości wektora wygenerowany zostanie wyjątek.

A.2.7.3 `vpush`

```
vpush Rd Rs
```

Opis Przenosi wartość z rejestru `Rs` na koniec wektora znajdującego się w rejestrze `Rd`.

A.2.7.4 `vpop`

```
vpop Rd Rs Ri
```

Opis Usuwa wartość znajdującą się pod indeksem określonym przez liczbę całkowitą znajdującą się w rejestrze `Ri` w wektorze znajdującym się w rejestrze `Rs` i zapisuje ją w rejestrze `Rd`.

Jeśli rejestr `Rd` jest podany jako `void` to wartość usunięta z wektora nie jest zapisywana, a od razu niszczona.

Jeśli rejestr `Ri` jest podany jako `void` to wartość jest usuwana z końca wektora.

Rejestr `Rd` i `Ri` mogą być podane jako `void` jednocześnie (co spowoduje usunięcie ostatniej wartości w wektorze i jej nastychmiastowe zniszczenie).

Wyjątki Jeśli żądany indeks jest poza zakresem długości wektora wygenerowany zostanie wyjątek.

A.2.7.5 `vat`

```
vat Rd Rs Ri
```

Opis Tworzy wskaźnik do wartości znajdującej się pod indeksem określonym przez liczbę całkowitą znajdującą się w rejestrze `Ri` w wektorze znajdującym się w rejestrze `Rs` i zapisuje go w rejestrze `Rd`.

Wyjątki Jeśli żądany indeks jest poza zakresem długości wektora wygenerowany zostanie wyjątek.

A.2.7.6 `vlen`

```
vlen Rd Rs
```

Opis Odczytuje długość wektora znajdującego się w rejestrze `Rd` i zapisuje ją jako liczbę całkowitą w rejestrze `Rd`.

A.2.8 Atomy

A.2.8.1 `atom`

```
atom Rd '<value>'
```

Opis Konstruuje w rejestrze `Rd` atom o wartości *`value`*.

A.2.8.2 `atomeq`

```
atomeq Rd Ra Rb
```

Opis Sprawdza czy atomy w rejestrach `Ra` i `Ra` są identyczne i zapisuje wynik w postaci wartości boolowskiej w rejestrze `Rd`.

A.2.9 Struktury (rekordy)

A.2.9.1 `struct`

```
struct Rd
```

Opis Konstruuje pustą strukturę (rekord) w rejestrze `Rd`.

A.2.9.2 `structinsert`

```
structinsert Rt Rk Rv
```

Opis Do struktury znajdującej się w rejestrze `Rt`, do pola określonego przez atom znajdujący się w rejestrze `Rk` przenosi wartość znajdującą się w rejestrze `Rv`.

A.2.9.3 `structremove`

```
structremove Rd Rt Rk
```

Opis Ze struktury znajdującej się w rejestrze `Rt` usuwa pole określone przez atom znajdujący się w rejestrze `Rk` i zapisuje ją w rejestrze `Rd`.

Jeśli rejestr `Rd` jest podany jako `void` to usunięta ze struktury wartość jest natychmiast niszczona.

A.2.9.4 `structat`

```
structat Rd Rt Rk
```

Opis Tworzy wskaźnik do wartości w polu określonym przez atom znajdujący się w rejestrze `Rk` w strukturze znajdującej się w rejestrze `Rt`.

A.2.9.5 `structkeys`

```
structkeys Rd Rt
```

Opis Odczytuje listę pól struktury znajdującej się w rejestrze `Rt` i wynik zapisuje w rejestrze `Rd` jako wektor atomów.

A.2.10 Aktory

A.2.10.1 `process`

```
process Rd fn/0                                                    1
process void fn/0                                                  2
```

Opis Wywołuje funkcję tworząc nowy proces. Wariant 1 zapisuje PID nowego procesu w rejestrze `Rd`. Wariant drugi nie zapisuje PID nowego procesu.

```
process %4 local do_some_processing/1                              1
process void a_free_actor/4                                        2
```

A.2.10.2 `self`

```
self Rd                                                            1
```

Opis Konstruuje PID procesu wewnątrz którego instrukcja jest wykonywana.

A.2.10.3 `pideq`

```
pideq Rd Ra Rb                                                     1
```

Opis Porównuje wartości PID znajdujące się w rejestrach `Ra` i `Rb`. Wynik porównania zapisuje jako wartość boolowską w rejestrze `Rd`.

A.2.10.4 `join`

```
join Rd Rp                                                         1
join Rd Rp infinity                                                2
join Rd Rp <timeout>                                               3
```

Opis Zatrzymuje wykonanie procesu wewnątrz którego wykonywana jest instrukcja i oczekuje na zakończenie procesu identyfikowanego przez PID znajdujący się w rejestrze `Rp`.

Wariant pierwszy jest identyczny w zachowaniu jak wariant drugi i powoduje, że proces wykonujący instrukcję oczekuje w nieskończoność.

Wariant trzeci umożliwia podanie wartości *timeout*, która określa maksymalny czas oczekiwania na zakończenie pracy przez zadany proces. Czas może być podany w sekundach (*N*`s`) lub milisekundach (*N*`ms`), a wartość zero jest poprawna. Przykład:

```
join %1 local %2 local 1s                                          1
join %1 local %2 local 0ms                                         2
```

Jeśli rejestr `Rd` jest podany jako `void` wartość zwracana przez zadany proces jest natychmiast niszczona.

Uwagi Jeśli do momentu upłynięcia czasu oczekiwania zadany proces nie zakończył pracy zgłaszany jest wyjątek.

Jeśli zadany proces zakończył pracę z powodu awarii to nieobsłużony wyjątek, który tę awarię wywołał jest ponownie zgłaszany wewnątrz procesu, który wykonał instrukcję `join`.

A.2.10.5 `send`

```
send Rp Rs
```

Opis Wysyła wartość znajdującą się w rejestrze `Rs` jako wiadomość do procesu identyfikowanego przez PID znajdujący się w rejestrze `Rp`.

A.2.10.6 `receive`

```
receive Rd
receive Rd infinity
receive Rd <timeout>
```

Opis Blokuje proces wewnątrz którego wykonywana jest instrukcja i oczekuje na nadejście wiadomości. Jeśli jakaś wiadomość jest już dostępna w kolejce wiadomości procesu jest ona zwracana od razu, bez oczekiwania. Odebrana wiadomość jest umieszczana w rejestrze `Rd`.

Pierwszy i drugi wariant są identyczne w zachowaniu i blokują proces oczekując w nieskończoność na nadejście wiadomości.

Wariant trzeci umożliwia podanie wartości *timeout* określającej maksymalny czas oczekiwania na nadejście wiadomości w sekundach lub milisekundach. Czas oczekiwania może wynosić zero sekund lub milisekund.

Uwagi Jeśli po upłynięciu czasu oczekiwania żadna wiadomość nie będzie dostępna zgłaszany jest wyjątek.

A.2.11 Programowanie funkcyjne

A.2.11.1 `function`

```
function Rd function-name
```

Opis Konstruuje w rejestrze `Rd` wartość reprezentującą funkcję określoną przez *`function-name`*.

Wartość ta może być wykorzystana w instrukcjach `call`, `tailcall` i `process` aby wywołać funkcję niebezpośrednio. Przykład:

```
function %1 local a_function/0
frame %0
call void %1 local
```

A.2.11.2 `closure`

```
closure Rd closure-name
```

Opis Konstruuje w rejestrze `Rd` wartość będącą instancją domknięcia (ang. *closure*) określonego przez *`closure-name`*.

Zanim tak utworzona wartość będzie mogła być wywołana wszystkie wartości, do których domknięcie *closure-name* się odwołuje muszą zostać „schwytane” wewnątrz utworzonego domknięcia za pomocą instrukcji `capture`, `capturecopy` lub `capturemove`.

A.2.11.3 `capture`

```
capture Rc Ri Rv
```

Opis W rejestrze lokalnym o indeksie `Ri` wewnętrznym do domknięcia znajdującego się w rejestrze `Rc` tworzy referencję do wartości znajdującej się w rejestrze `Rv`.

```
text %3 local "Hello World!"
capture %1 local %1 %3 local
```

Przykład powyżej „chwyta" wartość tekstową znajdującą się w rejestrze `%3 local` i referencję do niej umieszcza w lokalnym rejestrze o indeksie 1 wewnętrznym do domknięcia znajdującego się w rejestrze `%1 local`.

Uwagi – czas życia Schwytanie wartości przez referencję powoduje, że przestaje ona być śledzona przez standardowy deterministyczny mechanizm zarządzania wartościami i jej czas życia może być trudny do określenia jeśli domknięcie jest zwracane z funkcji lub przekazywane dalej jak parametr.

Uwagi – przekazywanie wiadomości Wartości schwytane przez referencję nie mogą być przekazywane jako wiadomości ponieważ naruszyło by to ścisłą izolację procesów.

A.2.11.4 `capturecopy`

```
capturecopy Rc Ri Rv
```

Opis W rejestrze lokalnym o indeksie `Ri` wewnętrznym do domknięcia znajdującego się w rejestrze `Rc` tworzy kopię wartości znajdującej się w rejestrze `Rv`.

```
text %3 local "Hello World!"
capturecopy %1 local %1 %3 local
```

Przykład powyżej „chwyta" wartość tekstową znajdującą się w rejestrze `%3 local` i jej kopię umieszcza w lokalnym rejestrze o indeksie 1 wewnętrznym do domknięcia znajdującego się w rejestrze `%1 local`.

Odpowiada to instrukcji `copy`, dla której wynikowym zestawem rejestrów jest lokalny zestaw rejestrów wewnętrzny dla domknięcia.

A.2.11.5 `capturemove`

```
capturemove Rc Ri Rv
```

Opis Przenosi wartość z rejestru `Rv` do rejestru lokalnego o indeksie `Ri` wewnętrznego do domknięcia znajdującego się w rejestrze `Rc`.

```
text %3 local "Hello World!"
capturemove %1 local %1 %3 local
```

Przykład powyżej „chwyta" wartość tekstową znajdującą się w rejestrze `%3 local` i przenosi ją do lokalnego rejestru o indeksie 1 wewnętrznego do domknięcia znajdującego się w rejestrze `%1 local`.

Odpowiada to instrukcji `move`, dla której wynikowym zestawem rejestrów jest lokalny zestaw rejestrów wewnętrzny dla domknięcia.

A.2.12 Kontrola przepływu

Specyfikacje celów skoków

```
+<offset>
-<offset>
<marker>
```

Pierwszy i drugi wariant przenoszą kontrolę o *offset* instrukcji w przód lub w tył. Trzeci wariant przenosi kontrolę do pierwszej instrukcji za markerem *marker*.

A.2.12.1 `nop`

```
nop
```

Opis Instrukcja „pusta”.

A.2.12.2 `jump`

```
jump <jump-spec>
```

Opis Instrukcja skoku bezwarunkowego. Przenosi kontrolę do instrukcji określonej przez *`jump-spec`*.

A.2.12.3 `if`

```
if Rcond <true-jump-spec> <false-jump-spec>
```

Opis Instrukcja skoku warunkowego odczytuje wartość z rejestru `Rcond`, interpretuje ją jako wartość boolowską i podejmuje decyzję, który skok wykonać. Jeśli warunek jest prawdziwy wykonywany jest skok *`true-jump-spec`*, jeśli warunek jest prawdziwy wykonywany jest skok *`false-jump-spec`*.

A.2.12.4 `frame`

```
frame n
```

Opis Przygotowuje do ramkę wywołania zawierającą n rejestrów do przekazania parametrów, np. `frame %2` utworzy ramkę z dwoma rejestrami.

Przygotowana ramka jest potem „konsumowana” przez instrukcję `call`, `tailcall`, `process`, `defer` lub `watchdog`.

A.2.12.5 `param`

Instrukcja prywatna. Przekazanie parametrów przez kopię odbywa się za pomocą instrukcji `copy`.

Uwagi Instrukcje prywatne nie są dostępne dla kodu użytkownika, a programy zawierające je są odrzucane przez assembler jako nieprawidłowe.

A.2.12.6 `pamv`

Instrukcja prywatna. Przekazanie parametrów przez przeniesienie odbywa się za pomocą instrukcji `move`.

Uwagi Instrukcje prywatne nie są dostępne dla kodu użytkownika, a programy zawierające je są odrzucane przez assembler jako nieprawidłowe.

A.2.12.7 `call`

```
call Rr function-name
call Rr Rf
```

Opis Instrukcja wywołania funkcji lub instancji domknięcia. Konsumuje utworzoną wcześniej ramkę wywołania i dodaje ją na szczyt stosu wywołań.

Wariant 1 wywołuje funkcję określoną przez *function-name*. Wariant 2 wywołuje funkcję lub instancję domknięcia wskazywaną przez wartość w rejestrze `Rf`. Tylko wariant drugi może być użyty do wywołania instancji domknięcia.

W każdym wariancie wynik pracy funkcji jest umieszczany w rejestrze `Rr`. Jeśli rejestr `Rr` jest podany jako `void` to wartość zwrotna wywołanej funkcji jest natychmiast niszczona.

A.2.12.8 `tailcall`

```
tailcall function-name
tailcall Rf
```

Opis Instrukcja wywołania ogonowego funkcji lub instancji domknięcia. Konsumuje utworzoną wcześniej ramkę wywołania i zamienia ją z ramką znajdującą się na szczycie stosu wywołań – nie zwiększając rozmiaru stosu wywołań.

Wariant 1 wywołuje funkcję określoną przez *function-name*. Wariant 2 wywołuje funkcję lub instancję domknięcia wskazywaną przez wartość w rejestrze `Rf`. Tylko wariant drugi może być użyty do wywołania instancji domknięcia.

A.2.12.9 `defer`

```
defer function-name
defer Rf
```

Opis Instrukcja wywołania odroczonego funkcji. Konsumuje utworzoną wcześniej ramkę wywołania.

Wariant 1 wywołuje funkcję określoną przez *function-name*. Wariant 2 wywołuje funkcję wskazywaną przez wartość w rejestrze `Rf`.

Wywołania odroczone będą wywołane w odwrotnej kolejności zgłaszania ich w momencie, w którym ramka wywołania, w której zostały zarejestrowane jest zdejmowana ze stosu wywołań (np. na skutek wykonania instrukcji `return`, podmiany ramki przez instrukcję `tailcall` lub odwinięcia stosu na skutek nieobsłużonego wyjątku).

A.2.12.10 `arg`

Instrukcja prywatna. Odczyt parametrów przez przeniesienie odbywa się za pomocą instrukcji `move`.

Uwagi Instrukcje prywatne nie są dostępne dla kodu użytkownika, a programy zawierające je są odrzucane przez assembler jako nieprawidłowe.

A.2.12.11 `allocate_registers`

```
allocate_registers Rn
```

Opis Alokuje lokalny zestaw rejestrów w rozmiarze określonym przez operand `Rn`. Jest to pierwsza instrukcja, która musi być wykonana przez funkcję.

Uwagi Domknięcia *nie mogą* używać tej instrukcji. Ich zestaw rejestrów lokalnych jest alokowany przez instrukcję `closure`.

A.2.12.12 `return`

```
return
```

Opis Kończy wykonywanie funkcji i zdejmuje jej ramkę ze stosu wywołań.

A.2.12.13 `halt`

```
halt
```

Opis Natychmiast przerywa wykonywanie programu i wyłącza maszynę wirtualną.

A.2.13 Obsługa błędów

A.2.13.1 `watchdog`

```
watchdog <function-name>
```

Opis Ustawia funkcję *`function-name`* jako funkcję *watchdog*. Ta funkcja zostanie wywołana w momencie, w którym proces ulegnie awarii – czyli w momencie, w którym na skutek zgłoszenia wyjątku zostanie odwinięty cały stos wywołań w procesie, a wyjątek nie zostanie mimo to obsłużony.

W takim przypadku funkcja *watchdog* jest uruchamiana, a jako swój pierwszy parametr otrzymuje strukturę przechowującą informacje o:

1. funkcji głównej procesu, który uległ awarii
2. wyjątku, który spowodował awarię

Funkcja *watchdog* może obsłużyć awarię na wiele sposobów, ale typowe są dwie strategie:

1. powiadomienie procesu nadrzędnego o wystąpieniu awarii, a następnie zakończenie procesu, który tej awarii uległ
2. zrestartowanie procesu, który uległ awarii

Jeśli wykorzystywana jest druga strategia pozostałe procesy w systemie mogą nie zauważyć, że proces uległ awarii (co jest pożądane) ponieważ zachowuje on swój PID (swoją „tożsamość”) oraz wszystkie wiadomości, które znajdowały się w jego kolejce wiadomości w momencie wystąpienia awarii.

A.2.13.2 `throw`

```
throw Rs
```

Opis Zgłasza wartość znajdującą się w rejestrze `Rs` jako wyjątek.

A.2.13.3 `catch`

```
catch "<exception-tag>" <block-name>
```

Opis Wyłapuje wyjątki z tagiem *`<exception-tag>`* i przenosi kontrolę do bloku *`<block-name>`*, który zajmie się ich obsługą. Wyłapane wyjątki są umieszczane w specjalnym rejestrze, a pobrane mogą być z niego za pomocą instrukcji `draw`.

Uwagi Instrukcje `catch` zazwyczaj wykonywane są w serii – wyłapując kilka wyjątków w jednej ramce obsługi wyjątków. Przed każdą taką serią musi zostać wykonana instrukcja `try`. Ramka obsługi wyjątków musi istnieć zanim możliwe będzie rejestrowanie bloków obsługi.

A.2.13.4 `draw`

```
draw Rd
```

Opis Przenosi wyłapany wyjątek z rejestru wyjątków do rejestru `Rd`.

Rejestr `Rd` może być podany jako `void` żeby natychmiast zniszczyć wartość znajdującą się w rejestrze wyjątków. Wartość wyjątku nie zawsze jest potrzebna, ale rejestr wyjątków *musi* zostać zawsze opróżniony w każdym bloku obsługi.

Uwagi Jeśli w momencie wykonania tej instrukcji rejestr wyjątków jest pusty zgłoszony zostanie wyjątek.

A.2.13.5 `try`

```
try
```

Opis Przygotowuje ramkę, w której rejestrowane za pomocą instrukcji `catch` będą bloki obsługi dla wyjątków.

A.2.13.6 `enter`

```
enter <block-name>
```

Opis Przeniesienie kontroli do bloku *<block-name>*. Instrukcja konsumuje ramkę obsługi wyjątków utworzoną wcześniej przez instrukcję `try`, a całe wykonanie wewnątrz bloku chronionego *<block-name>* jest „chronione” przez bloki obsługi wyjątków zarejestrowane w tej ramce (przez instrukcje `catch`).

A.2.13.7 `leave`

```
leave
```

Opis Wyjście z bloku. Instrukcja powoduje zakończenie wykonywania bloku chroninego lub bloku obsługi wyjątków.

A.2.14 Systemu modułów

A.2.14.1 `import`

```
import module-name
```

Opis Instrukcja dynamicznie dołącza moduł *module-name* do uruchomionego programu.

Uwagi Użycie instrukcji `import` jest niezalecane. Zamiast niej należy używać dyrektywy import opisanej w rozdziale A.1.4 na stronie 145.

Dodatek B

Model obliczeniowy Viua VM

Model obliczeniowy Viua VM jest oparty na modelu aktorów. Bardzo abstrakcyjnie i skrótowo może być opisany podanymi poniżej cechami:

1. programy uruchamiane w Viua VM składają się z jednego lub większej liczby równolegle działających, odizolowanych od siebie aktorów (procesów)
2. każdy aktor jest implementowany przez pojedynczą funkcję „główną”
3. funkcje mogą wywoływać siebie nawzajem, tworząc stos wywołań
4. funkcje przyjmują wartości w parametrach, informują o wynikach swojej pracy zwracając wartości
5. każda funkcja jest reprezentowana przez uszeregowaną sekwencję instrukcji
6. instrukcje wewnątrz funkcji są wykonywane sekwencyjnie, jedna po drugiej
7. instrukcje wykonują pracę modyfikując wartości znajdujące się w rejestrach
8. każdy rejestr może przechowywać dowolną wartość
9. rejestry są grupowane w „zestawy”: lokalny (dla wartości lokalnych), argumentów (dla przekazanych argumentów), statyczny (dla wartości statycznych; ten jest zachowywany pomiędzy wywołaniami funkcji w ramach pojedynczego aktora)
10. każdy aktor może tworzyć nowe aktory
11. aktory komunikują się między sobą za pomocą asynchronicznej wymiany wiadomości
12. wiadomość jest dowolną wartością

O instrukcjach można myśleć jak o funkcjach, które na wejściu dostają zestawy rejestrów i adresy rejestrów, które powinny zmodyfikować, a na wyjściu podają zmodyfikowane zestawy rejestrów. Ich sygnatura w języku OCaml mogłaby wyglądać następująco:

```
val op : local_register_set                                    1
      -> static_register_set                                   2
      -> arguments_register_set                                3
      -> register_address list                                 4
      = <fun>                                                  5
```

Zasady działania poszczególnych instrukcji opisane są w dodatku A.

Dodatek C

Biblioteka `plain-websocket`

Biblioteka `plain-websocket` jest dostarczana jako zestaw funkcji opakowujący *POSIX sockets*, wyliczenie kodów operacji w protokole WebSocket i strukturę opisującą pojedynczą ramkę protokołu. Do użycia biblioteki nie jest wymagane projektowanie programu dookoła *event loop* – kontrola zostaje w całości po stronie programu używającego biblioteki.

Dla języka ViuAct biblioteka jest dostępna jako moduł `Websocket`.

C.1 Sposób użycia

Funkcji dostarczanych przez bibliotekę `plain-websocket` używa się między wywołaniem `accept(3)`, a `shutdown(3)`.

Po uzyskaniu połączenia za pomocą standardowych wywołań z API *POSIX sockets*, należy wykonać *handshake* protokołu WebSocket za pomocą funkcji `Websocket.handshake()`. Po zakończeniu pracy należy zakończyć połączenie w warstwie protokołu WebSocket za pomocą funkcji `Websocket.close()`.

C.1.1 C++

Przykład kodu w języku C++:

```
auto handle_websocket_connection(int sock) -> void {
    auto ws = maelkum::websocket::handshake(sock);

    auto message = maelkum::websocket::read_frame(sock);
    if (message.has_value()) {
        auto const text = maelkum::websocket::get_text(*message);
        if (text.has_value()) {
            std::cerr << *text << "\n";
            maelkum::websocket::write(sock, *text);
        }
    }

    maelkum::close(sock);
}

auto main() -> int {
    auto const host = "127.0.0.1";

    auto server_sock = socket(AF_INET, SOCK_STREAM, 0);
    sockaddr_in addr;
    addr.sin_family = AF_INET;
    addr.sin_port = htons(9090);
    inet_pton(AF_INET, host, &addr.sin_addr.s_addr);

    bind(sock, reinterpret_cast<sockaddr*>(&addr), sizeof(addr));
    listen(sock, 0);
```

```
    auto client = accept(sock, nullptr, nullptr);

    handle_websocket_connection(sock);

    shutdown(sock, SHUT_RDWR);
    close(sock);

    shutdown(server_sock, SHUT_RDWR);
    close(server_sock);

    return 0;
}
```

C.1.2 ViuAct

Przykład kodu w języku ViuAct:

```
(import Std.Posix.Network)
(import Websocket)

(let handle_websocket_connection (sock) {
    (Websocket.handshake sock)
    (let message (Websocket.read sock))
    (print message)
    (Websocket.write sock message)
    (Websocket.close sock)
    0
})
(let main () {
    (let server_sock (Std.Posix.Network.sock))
    (Std.Posix.Network.bind sock "127.0.0.1" 9090)
    (Std.Posix.Network.listen sock 16)

    (let client (Std.Posix.Network.accept sock))
    (handle_websocket_connection sock)

    (Std.Posix.Network.close sock)
    (Std.Posix.Network.close server_sock)
    0
})
```

C.2 API

C.2.1 `Websocket.handshake()`

```
Websocket.handshake(Integer)
```

Wykonanie *handshake* protokołu WebSocket. Po wykonaniu tej funkcji socket jest gotowy do użycia jako argument dla pozostałych funkcji z modułu `Websocket`.

C.2.2 `Websocket.read()`

```
Websocket.read(Integer)
```

Odczyt pojedynczej wiadomości tekstowej z socketu. Jeśli wiadomość nie jest możliwa do odebrania zgłaszany jest wyjątek z tagiem `Exception`.

C.2.3 `Websocket.write()`

```
Websocket.write(Integer, String)
```

Zapis pojedynczej wiadomości do socketu.

C.2.4 `Websocket.close()`

```
Websocket.close(Integer)
```

Zakończenie połączenia w warstwie protokołu WebSocket.

Dodatek D

Narzędzie śledzenia zadań `issue`

Narzędzie `issue` jest to rozproszony system śledzenia zadań autorstwa Marka Mareckiego. Zapewnia ono również wysoką ergonomię i prędkość pracy w środowisku tekstowym, oraz niezbędne minimum funkcjonalności potrzebne do wykonania typowych operacji związanych ze śledzeniem zadań w projekcie programistycznym. Narzędzie to jest rozwijane jako Free Software na licencji GNU GPL v3 i jest dostępne do pobrania z serwisu GitHub: `https://github.com/marekjm/issue`

D.1 Dane w rozproszonym systemie śledzenia zadań

Informacje o zadaniach, które są zachowywane obejmują:

1. tytuł i opis zadania
2. datę utworzenia zadania i datę zamknięcia zadania
3. osobę tworzącą i zamykającą zadanie
4. globalnie unikalny identyfikator (GUID) zadania
5. komentarze
6. tagi

Wszystkie modyfikacje są realizowane *offline* i zapisywane w formie plików „zmian" w zadaniu. Takie „rozproszenie" stanu każdego zadania na mnóstwo mniejszych plików powoduje, że baza danych systemu `issue` musi być okresowo indeksowana żeby jej stan prezentowany użytkownikowi nie był przekłamany, a jednocześnie uniknąć konsolidowania wszystkich zmian dotyczących zadania za każdym razem kiedy jest ono pokazywane użytkownikowi.

Decyzja o rozbiciu stanu zadania na mnóstwo małych plików jest wymuszona przez inną decyzję projektową – rozproszenie systemu i umożliwienie pracy całkowicie *offline*, co jest jedną z najważniejszych funkcjonalności `issue`. Zdecydowanie łatwiej jest pobrać wiele mniejszych plików i utworzyć z nich lokalny indeks niż projektować algorytm rozproszonej współpracy i negocjowania zmian w plikach.

„Baza danych" programu issue to po prostu zbiór plików umieszczonych w określonej strukturze katalogów. Umożliwia to bezproblemowe zagnieżdżenie jej w repozytorium systemu kontroli wersji Git. Narzędzie `issue` dostarcza również swój własny sposób dystrybucji plików opisujących zadania dla osob, które nie chcą zagnieżdżać repozytorium `issue` w repozytorium kodu źródłowego. Takie zagnieżdżenie jest jednak zalecaną strategią pracy z narzędziem `issue`.

D.2 Operacje

Operacje, które są oferowane przez `issue` to między innymi:

1. otwieranie nowych zadań

2. zamykanie zadań
3. łączenie zadań w drzewa (tworzy to zależności między zadaniami, a otwarte zadanie „liść" uniemożliwia zamknięcie zadania „gałęzi")
4. automatyczne tworzenie gałęzi w systemie śledzenia wersji Git na podstawie tytułów zadań
5. komentowanie zadań