# Wstęp

Dzisiejszy rynek wytwarzania oprogramowania staje się coraz bardziej wymagający. Oprócz jakości i krótkiego czasu dostarczenia produktu, na ważności zyskują procesy związane z pracą nad projektem. W celu osiągnięcia jak najwyższej niezawodności i powtarzalności, wdrażane są rozmaite metodologie i procesy optymalizujące pracę. Jednym z takich procesów jest właśnie automatyzacja pisania kodu źródłowego.

W nowoczesnych środowiskach programistycznych dostępne są na przykład rozbudowane moduły generujące skrawki kodu (ang. *code* *snippets*) czy podstawy nowego projektu. Dostępne są również samodzielne programy, tworzące pliki źródłowe od podstaw, a także pozwalające na ciągłą pracę z projektem poprzez generowanie nowych elementów. Przykładem takiego oprogramowania są napisane we frameworku Yeoman generatory, w pewnym stopniu wyręczające programistów z pracy nad tworzeniem samego kodu.

W ramach przedstawianej pracy opisane zostaną podstawy automatyzacji wytwarzania kodu źródłowego, pokazany będzie proces tworzenia generatora Yeoman, sposób jego działania, zastosowane w nim technologie i rozwiązania.

W pierwszym rozdziale przedstawiona zostanie krótka historia generatorów Yeoman, korzyści płynące z włączenia ich w swój tryb pracy, a także przykładowe generatory i ich funkcjonalności.

W drugim rozdziale zawarte będą informacje o narzędziach stosowanych przez framework, przedstawiony sposób działania frameworku i wytłumaczone zostaną podstawowe pojęcia.

Trzeci rozdział zawierał będzie szczegóły dotyczące implementacji własnego generatora, przechodząc kolejno przez wszystkie warstwy struktury programu, odzwierciedlając tok pracy generatora. Pokazane zostanie, jak można użyć generatora dostępnego w sieci, zarówno jak i stworzonego przez siebie. Przedstawione zostaną również mechanizmy zabezpieczające użytkowników generatorów przed przypadkowym wyrządzeniem szkód w swoich programach.

# 1 Podstawy teoretyczne

## Automatyzacja pisania kodu

### Programowanie automatyczne

Jest to typ programowania, w którym stosowane są mechanizmy generowania programu, pozwalając programiście pracować na wyższym poziomie abstrakcji. Jedną z pierwszych wzmianek o programowaniu automatycznym jest artykuł Saula Gorn’a[[1]](#footnote-1) z 1940 roku, czyli czasów, kiedy kod źródłowy był zapisywany na podziurkowanych kartach. Autor artykułu odnosi się w ten sposób do asemblerów, pozwalających programiście pisać kod, który po interpretacji przez komputer wybijany był automatycznie na perforowanych kartach w postaci dziurek. Oznaczało to, że komputer automatycznie wykonuje zadania, które normalnie wykonywałby człowiek. Później, termin programowania automatycznego odnosił się do używania języków jak FORTRAN, w których programista specyfikował co chce osiągnąć, a komputer interpretował jego specyfikację generując kod maszynowy. Programowanie automatyczne oznaczało w końcu to, co dzisiaj rozumiemy poprzez programowanie w językach z wysokim poziomem abstrakcji.

David Lorge Parnas[[2]](#footnote-2), profesor Carnegie Mellon University w Kanadzie, w jednym ze swoich listów[[3]](#footnote-3) w których zastanawia się nad użyciem oprogramowania w systemach broniących przed atakami nuklearnymi uważa, że programowanie automatyczne niezbyt nadaje się do wykorzystania w systemach czasu rzeczywistego, z ograniczonym czasem na reakcję. Podnosi, iż kod pisany na wyższym poziomie abstrakcji jest mało efektywny i niedokładny, co powoduje jego nieużyteczność w systemach zarządzania bitewnego. David Parnas trafnie przewidział przyszłość, gdyż do dziś najważniejsze systemy militarne pisane są w językach niskiego poziomu, z bezpośrednim dostępem do sprzętu.

Ciekawe wyjaśnienie pojęcia programowania automatycznego podała Mildred Koss w swoim artykule[[4]](#footnote-4): (tłumaczenie własne)

„Pisane kodu maszynowego zawierało wiele nudnych kroków – rozbijanie procesu na odrębne instrukcje, przypisywanie specyficznych adresów w pamięci dla każdych komend i zarzadzanie 30 buforami IO. Po wykonaniu tych kroków i zaimplementowaniu procedur matematycznych, bibliotek z pod-procedurami i programów sortujących, naszym zadaniem było spojrzenie na proces wytwarzania oprogramowania z góry. Musieliśmy zrozumieć, jak możemy wykorzystywać ponownie przetestowany kod i jak maszyna może pomóc w programowaniu. Podczas programowania, analizowaliśmy proces i próbowaliśmy znaleźć drogę do abstrakcji tych kroków, aby włączyć je do języka wyższego poziomu. To doprowadziło do stworzenia interpreterów, asemblerów, kompilatorów i generatorów – programów zaprojektowanych do tworzenia innych programów, czyli *programowania automatycznego*”

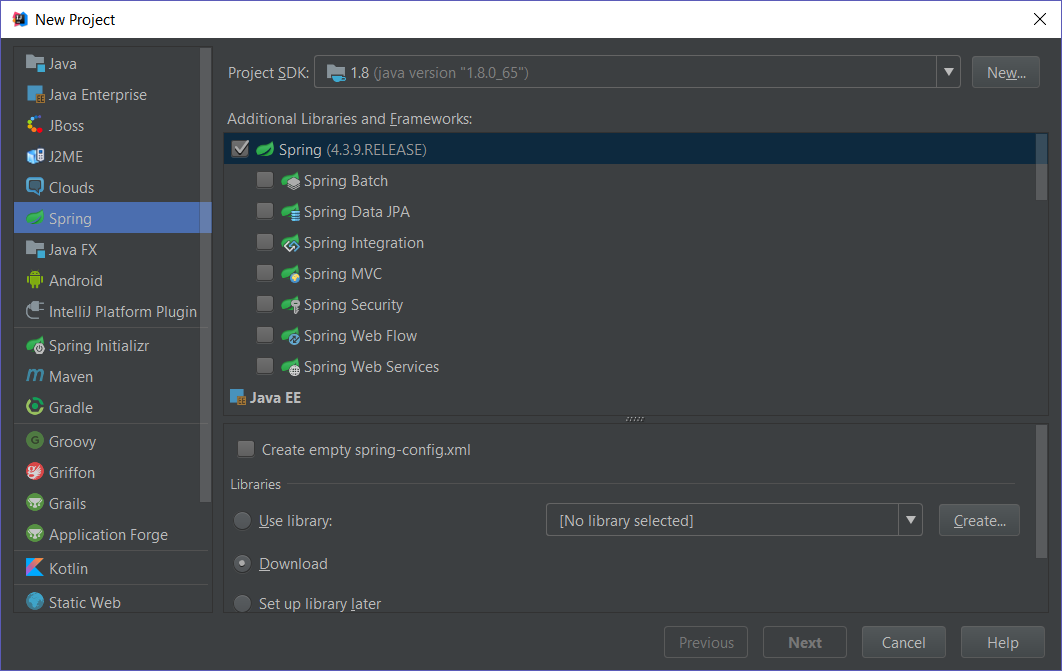
Programiści zwykle dążą do zwiększania swojej produktywności poprzez automatyzację zadań. Takie podejście doprowadziło do stworzenia języków programowania jakie znamy dzisiaj, kompilatorów kodu z jednego języka do drugiego, a także generatorów tworzących cały kod źródłowy projektu. Programowanie automatyczne w dzisiejszym znaczeniu oznacza w dużej mierze samo generowanie kodu źródłowego, transformację abstrakcyjnych modeli do kodu, a czasem nawet automatyczną refaktoryzację kodu w środowiskach programistycznych.

### Automatyczne generowanie kodu źródłowego

Z automatycznego generowania kodu źródłowego wynika wiele korzyści. Jedną z tych oczywistych jest przyspieszenie pracy. Interfejs użytkownika zwykle jest w aplikacjach jednolity, co oznacza, że ekrany są do siebie podobne. Gdyby programista posiadał pewien wzorzec, mógłby uzupełniać go automatycznie, zaoszczędzając sobie pisania wielu linii kodu. Automatycznie generowany kod jest zwykle dobrze napisany, co ułatwia pracę z nim. Takie podejście pozwala programiście na kreatywną pracę, koncentrację nad rzeczywistym problemem i niemartwienie się o powtarzalne czynności. Ostatecznie, generowanie kodu źródłowego zmniejsza koszty, poprzez pozostawienie mniej miejsca na nieporozumienia i ludzkie błędy.

### Generowanie kodu w IDE

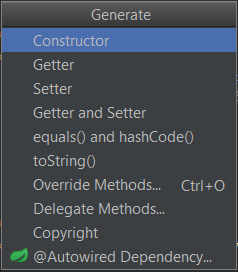
Funkcjonalność automatycznego tworzenia kodu źródłowego dostępna jest dzisiaj w każdym powszechnie używanym zintegrowanym środowisku programistycznym. Trudno jest wyobrazić sobie nowoczesne narzędzie, nieposiadające takich możliwości. Zazwyczaj proces generowania kodu w IDE jest interaktywny, co pozwala użytkownikowi dobrze dobrać potrzebne mu elementy. Zwykle dostępnych jest wiele archetypów, z których programista może wybrać typ swojej aplikacji. Kreatory interfejsów użytkownika, w których programista może dobrać wygląd swojego programu, są też w istocie generatorami kodu źródłowego.



Rysunek 1 Oferujący wiele archetypów nowego projektu kreator w IntelliJ IDEA

Oprócz tego typu automatyzacji, IDE często wyposażone są w potężne mechanizmy refaktoryzacji kodu. Na uwagę zasługuje środowisko IntelliJ IDEA, potrafiące np. odnajdować duże i skomplikowane pętle w języku Java, sugerując zamianę ich w bardziej optymalne wyrażenia lambda[[5]](#footnote-5). Może przeprowadzać mądrą refaktoryzację, wydzielając funkcje, zmienne, stałe i nowe klasy. Możliwa jest również transkompilacja kodu[[6]](#footnote-6) z jednego języka programowania do drugiego. Są to główne powody, dla których użytkownicy wybierają to środowisko programistyczne, co dobrze pokazuje jak ważną jest to kwestią.

Poza tworzeniem nowego projektu, IDE oferują często generowanie przy użyciu szablonów skrawków kodu w postaci metod, klas i nowych plików. Umożliwiają także rozszerzanie tej funkcjonalności przy użyciu pluginów, jak Acceleo dla Eclipse, zamieniający pliki w formacie EMF (Eclipse Modeling Framework) na kod źródłowy w wybranym języku, czy *Live Templates* w IntelliJ IDEA pozwalające na definiowanie własnych generowanych struktur.



Rysunek 2 Menu kontekstowe pozwalające na generowanie skrawków kodu w IntelliJ IDEA

### Ryzyko wynikające z automatyzacji

Z automatyzacji wynika na pewno wiele korzyści, ale nie sposób jest nie wspomnieć także o wynikającym z niej ryzyku. Tworzenie oprogramowania w sposób automatyczny nie jest idealne, służące do tego narzędzia pisane są przecież przez zwykłych programistów. Mogą zdarzyć się przypadki, w których wygenerowany kod jest niekompletny lub wręcz błędny. Takie błędne elementy łatwo jest jednak wychwycić, gdyż są oczywiste. Niekiedy program może generować zbyt ogromną ilość kodu, niezrozumiałą dla początkującego programisty. O ile niektóre nowo wygenerowane projekty są dobrze ustrukturyzowane, niektóre bardziej zaawansowane narzędzia mogą iść za daleko i tworzyć niepotrzebne pliki, które nigdy nie zostaną użyte, wprowadzając zamęt. Jest to realne zagrożenie, mogące negatywnie wpłynąć na efektywność programu, a także na jego jakość. Jednak o ile użyty zostaje program dobrze przetestowany i powszechnie używany, zazwyczaj nie ma się czego obawiać. Interaktywność rozwiązania pozwala zwykle trafnie dobrać elementy do swoich potrzeb.

## Yeoman

### 1.2.1 Historia frameworku

Yeoman został po raz pierwszy przedstawiony na konferencji Google I/O w 2012 roku. W swoim początku był to zintegrowany zestaw dobrze współdziałających narzędzi, upraszczając tryb pracy nad tworzeniem aplikacji internetowych. Pozwalał na utrzymywanie aktualnych wersji zależności aplikacji, generowanie kodu *boilerplate*[[7]](#footnote-7), udostępniał płynny proces budowania programu z trybem automatycznego odświeżania wprowadzanych zmian. Został bardzo dobrze przyjęty, od razu zostając jedną z dominujących technologii w zakresie generowania kodu. Jest projektem *open* source, zainicjowanym przez Addy’ego Osmani, inżyniera firmy Google. Kod źródłowy projektu jest dostępny na oficjalnym repozytorium w serwisie GitHub.com[[8]](#footnote-8).

Przez okres kilku lat, z pomocą wielu programistów, projekt ewoluował. Obecnie skupia się głównie na ekosystemie generatorów, nadal jednak udostępniając swoje kluczowe funkcjonalności. Aktualnie na oficjalnej stronie projektu[[9]](#footnote-9) dostępnych jest publicznie ponad 6000 generatorów utworzonych przez licznych członków społeczeństwa, pomagających wystartować z projektem w ulubionej technologii. *Yo* - narzędzie potrzebne do uruchamiania generatorów zostało dotychczas pobrane z repozytorium *npm* ponad 3 miliony razy, w tym ponad 100 000 razy w miesiącu maj 2017. Obecnie Yeoman jest zespołem narzędzi, składającym się z między innymi:

* npm – menadżer pakietów środowiska Node.js
* Yo - narzędzie linii komend do uruchamiania generatorów
* EJS - praca z szablonami plików za pomocą języka JavaScript
* InquirerJS - zespół interaktywnych interfejsów linii komend do komunikacji z użytkownikiem
* Gulp - system do automatyzacji powtarzających się czynności

### 1.2.2 Inspiracja i możliwości

Yeoman został zainspirowany często używanym przez programistów języka Ruby systemem generatorów frameworku *Ruby on Rails*, użytecznym podczas tworzenia aplikacji sieciowych. Wprowadził warstwę, która pozwala programistom na tworzenie generatorów niezależnych od docelowej technologii. Po uprzednim zidentyfikowaniu powtarzających się elementów kodu źródłowego aplikacji, programista może w prosty sposób napisać własny generator, znacznie ułatwiając sobie pracę z projektem. Wiele takich elementów zostało już zidentyfikowanych, co zaowocowało dostępnością implementacji dla praktycznie każdej popularnej dziś technologii.

Rysunek 3 Logo frameworku Yeoman

Generatory potrafią stworzyć np. podstawowe szablony HTML, szkice skryptów języka JavaScript, jak i kompletne projekty łączące w sobie wiele technologii i rozwiązań, na przykład aplikację internetową gotową do działania lub uruchomienia na chmurze. Na szczególną uwagę zasługuje możliwość wybierania w interaktywny sposób potrzebnych elementów, często implementowana w generatorach. W zależności od potrzeb, wygenerowana aplikacja może być tylko przykładowym szkieletem lub w pełni funkcjonalną implementacją. Z racji, że większość udostępnianych przez programistów generatorów jest dostępna na bazie oprogramowania *open source* na odpowiedniej licencji, użytkownik może w prosty sposób modyfikować działanie używanego generatora na swoje potrzeby.

### 1.2.3 Korzyści płynące z narzędzia Yeoman

Dobrze przygotowany generator potrafi błyskawicznie na podstawie interakcji z użytkownikiem utworzyć kompletny, kompilujący się i gotowy do uruchomienia program. Jednym z głównych założeń, które programista powinien mieć na uwadze pisząc własny generator jest to, aby nie powodował on w docelowym programie żadnych usterek i nie psuł jego struktury. Zawsze przed momentem, w którym pliki są zmieniane lub tworzone, użytkownik może dokładnie przejrzeć zmiany jakie zajdą w jego programie. Używanie generatorów w kompatybilnych projektach jest z reguły bezpieczne i nie powinno powodować żadnych negatywnych, nieoczekiwanych efektów.

Tworząc pliki w jasno określony sposób, Yeoman pozwala zadbać o dobrą strukturę katalogów i podział projektu na moduły, w pewien sposób wymuszając na programiście dobre praktyki i przyspieszając pracę z projektem. Generator do odpowiedniego działania musi wiedzieć na przykład, gdzie spodziewać się plików konfiguracyjnych, gdzie znajdują się pliki klas i jak wygląda struktura programu. Takie warunki jego działania, pomagają bez przerwy dbać o dobrą strukturę projektu, chociażby dla zachowania samej jego kompatybilności z przydatnym generatorem.

Generator Yeoman to nie tylko szybki start z projektem i dobre praktyki, ale także dynamiczne implementowanie konkretnych funkcjonalności. Wiele zadań w toku pracy nad oprogramowaniem jest powtarzalnych, dodawane kolejno nowe moduły często są kopią poprzednich implementacji, dotycząc tylko innych danych. Przykładowo w aplikacjach internetowych zapisujących obiekty w bazie danych, dla każdej nowej encji (klasy-modelu) zwykle implementuje się operacje dodawania, usuwania, edycji czy pobierania listy obiektów. Dodawanie takich funkcjonalności jest czynnością wysoce powtarzalną i rzadko różniącą się czymś więcej, niż na przykład nazwami używanych klas lub miejscem, w którym znajdują się pliki źródłowe. Jest to idealne zastosowanie dla generatora, który mógłby przyjmować jako dane wejściowe nazwę wspomnianej encji i jej podstawowe atrybuty, generując:

* Klasę modelu encji
* klasę DAO (*Data Access Object)* z podstawowymi metodami
* serwis będący miejscem na logikę biznesową
* kontroler stanowiący interfejs do interakcji z zewnętrznymi systemami

Zgodne z konwencją i dobrze nazwane wygenerowane klasy umieszczone byłyby w odpowiednich katalogach, zapewnione byłoby ich odpowiednie współdziałanie. Używanie generatora zostawia mało miejsca na pomyłkę programisty, z której mogłyby wyniknąć problemy mające wpływ np. na kompilowanie się projektu. Zaoszczędzony czas pozwala programiście nie skupiać się na powtarzalnych czynnościach i od razu implementować logikę biznesową, bez obaw o tworzenie w odpowiedniej, działającej struktury. Większość powtarzalnych czynności da się zaimplementować w generatorze, co zazwyczaj jest już zrealizowane. Takie nowoczesne podejście pozwala na uniknięcie błędów i znacznie przyspiesza pracę.

Częstym elementem pracy, o którym programistom zdarza się zapominać podczas tworzenia oprogramowania jest pisanie testów jednostkowych zapewniających o poprawnym działaniu programu. Zazwyczaj sytuacja taka wynika z niejasnej, nie działającej odpowiednio lub nieistniejącej struktury środowiska testowego w projekcie. W przypadku używania generatorów do tworzenia konkretnych funkcjonalności, realnym scenariuszem staje się generowanie również podstawowych testów ilustrujących działanie programu, a także nawet uruchamianie ich od razu po wygenerowaniu kodu.

### 1.2.4 Przypadki użycia generatora

Yeoman jest narzędziem efektywnym w praktycznie każdym używanym dziś języku programowania. Generowany kod może mieć dowolną formę, co powoduje, że Yeoman znajduje dziś wiele różnych zastosowań:

* tworzenie nowego projektu
* tworzenie nowych elementów istniejącego projektu
* tworzenie modułów i pakietów
* prototypowanie i przygotowywanie wersji demo aplikacji
* wymuszanie na programistach dobrych praktyk
* generowanie przykładowego kodu w do użytku w poradnikach
* tworzenie prezentacji multimedialnych

Sam framework, bez odpowiedniej implementacji nie podejmuje żadnych decyzji odnośnie generowanego kodu. Fakt, że wszystko zależne jest od programisty pozwala na elastyczność i tworzenie generatorów nie związanych nawet ściśle z programowaniem. Ciekawym przypadkiem użycia jest stworzony przez Adriana Bateman (Program Manager w zespole *Microsoft Edge Web Platform*) generator specyfikacji technicznej opartej na bibliotece *ReSpec*, przyspieszający tworzenie nowych dokumentów technicznych[[10]](#footnote-10). W swoim początku, Yeoman promowany był jako narzędzie mające na celu pomóc głownie programistom front-end, jednak w miarę upływu czasu znalazł zastosowanie w większej ilości dziedzin programowania.

### 1.2.5 Przykładowe generatory

Generatory Yeoman zwykle tworzone są z myślą o szerszym gronie użytkowników, dlatego na rynku dostępnych jest wiele ciekawych implementacji, powszechnie używanych, dobrze przetestowanych i ciągle rozwijanych:

* **JHipster** - bez wątpienia najbardziej popularny, bardzo dobrze realizujący założenia dobrze napisanego generatora i pokazujący jak bardzo użyteczny może być Yeoman. Generuje w pełni funkcjonalną aplikację internetową w języku Java (Spring Boot + Angular), zarówno w architekturze monolitycznej[[11]](#footnote-11), jak i mikro serwisów[[12]](#footnote-12). Pozwala na stworzenie samego projektu, jak i ciągłą z nim pracę przy użyciu sub-generatorów. Używany jest codziennie w wielu renomowanych firmach, jak Adobe, Bosch, Google, Siemens czy Orange. Oferuje szereg narzędzi, jak np. wtyczki do IDE czy sub-generatory konfiguracji chmurowej. Posiada dużą różnorodność w wyborze używanych technologii oraz duże wsparcie społeczeństwa programistycznego i wielu fanów.

Logo JHipster

* **Angular** – generator aplikacji jednostronnych[[13]](#footnote-13) Angular, pozwalający na utworzenie kodu zarówno w języku JavaScript, CoffeeScript jak i TypeScript. Posiada szereg podgeneratorów dla popularnych elementów aplikacji AngularJS, jak kontrolery, widoki, dyrektywy i filtry. Utworzony został przez zespół dostarczający rozwijający framework Yeoman.
* **generator** – generator generatorów Yeoman. Przygotowuje strukturę plików potrzebną do sprawnego rozpoczęcia implementacji własnego generatora, tworzy pliki zwykle dodawane w przypadku pracy z repozytorium kodu i Node.js.
* **release** – ułatwiający publikację aplikacji w repozytorium NPM, przeglądający zmiany dokonane od ostatniej publikacji, tworzący na ich podstawie dziennik zmian, pomagający utrzymaniu poprawnego jej wersjonowania

# 2 Środowisko Yeoman

Proces instalacji środowiska będącego zespołem narzędzi *open source*, którego główną ideą jest przyspieszanie i ułatwianie pracy, musiał być prosty i jednoznaczny. Oczywistym wyborem docelowej platformy uruchomieniowej stał się więc realizujący te założenia Node.js, wraz z intuicyjnym menadżerem zarządzania pakietami NPM. Łącząc wymienione narzędzia w całość, zarysowuje się wygląd Yeomana jako środowiska kompletnego, oferującego programiście pomoc w każdym aspekcie pracy nad jego projektem. Są to powody, dla którego Yeoman spotkał się z tak dobrym odbiorem, i jest dziś powszechnie używany na całym świecie.

## 2.1 Stosowane technologie

### 2.1.1 Node.js

Środowisko uruchomieniowe utworzone na podstawie silnika JavaScript V8 dostarczonego przez Google. Jest to platforma sterowana zdarzeniami[[14]](#footnote-14), zaprojektowana w celu tworzenia wysoce skalowalnych aplikacji internetowych. Nieblokujący model I/O[[15]](#footnote-15) pozwala na kontynuację wykonywania programu, obsługując zdarzenia poprzez zwrotne wywołania funkcji (ang. c*allback).* Posiada prosty system modułowości, pozwalający na ładowanie i używanie funkcjonalności z osobnych modułów.

Listing 1 - Przykład asynchronicznej, nieblokującej obsługi otwierania pliku używając modułu fs

const fs = require('fs');

fs.readFile('/file.md', (err, data) => {

if (err) {

throw err;

}

this.log(data);

});

### 2.1.2 npm

Repozytorium pakietów dla środowiska Node.js, a zarazem narzędzie wiersza poleceń pozwalające na ich pobieranie i instalację. Pozwala na automatyzację zarządzania zależnościami aplikacji poprzez deklarację używanych modułów w pliku *package.json.* W repozytorium NPM znajdują się setki tysięcy różnych pakietów gotowych do pobrania. NPM umożliwia pobieranie konkretych wersji dostępnych pakietów i pozwala w łatwy sposób publikować swoje moduły innym. Jest używany zarówno w procesie tworzenia swojego generatora Yeoman, jak i do pobrania zależności utworzonej aplikacji po wygenerowaniu projektu.

### 2.1.3 Gulp

Narzędzie do automatyzacji powtarzających się czynności w programowaniu. Pozwala na definiowanie zadań posiadających poszczególne kroki, uruchamianych z linii wiersza poleceń. Będąc kompatybilnym z wieloma zewnętrznymi bibliotekami, może znaleźć bardzo dużo zastosowań.

Jest narzedziem elastycznym, łatwo konfigurowalnym dla optymalnego procesu pracy z oprogramowaniem. Po wygenerowaniu kodu przez Yeoman, odpowiednie zadanie Gulp może na przykład przygotowywać serwer aplikacji, kompilować jej kod źródłowy, minifikować go i umieszczać w odpowiednich katalogach w zoptymalizowanej formie, przy okazji zapewniając odświeżanie projektu w przeglądarce po wykryciu zmian.

### 2.1.4 Bower

Narzędzie do automatycznego zarządzania zależnościami aplikacji. Umożliwia zarówno pobieranie wybranych bibliotek używając linii wiersza poleceń, jak i deklarację i automatyczne nimi zarządzanie przez plik *bower.json*. Używany jest głównie do pobierania zależności front-endowych, jak jQuery lub Backbone. W połączeniu z resztą narzędzi używanych przez Yeoman, umożliwa minimalną interakcję z projetem po wygenerowaniu, zarządzając instalacją jego zależności.

## 2.2 Sposób działania generatora Yeoman

### 2.2.1 Metody jako elementy bazowe

Programista rozszerzając prototyp Generator dostarczany przez moduł Yeoman, może definiować w jego wnętrzu różne metody. Każda metoda uważana jest za oddzielne zadanie do wykonania dla generatora i uruchamiana w sekwencji, pozwalając na poukładanie toku pracy programu w odpowiedni sposób.   
Istnieje również możliwość zadeklarowania metod prywatnych, nie wykonywanych automatycznie przez framework. Takie metody mogą służyć dodatkowej organizacji kodu i definiuje się je na trzy sposoby:

* poprzez prefiks podkreślenia w nazwie metody:\_privateMethod

\_privateMethod: function () {

this.log('This method wont run automatically');

},

* poprzez metody związane z konkretną instancją:

install: function () {

this.instanceMethod = function () {

this.log('This method wont run automatically');

};

},

* poprzez rozszerzanie generatora nadrzędnego:

class GenAPI extends Generator {

helperMethod() {

this.log('This method wont run automatically');

}

}

Użycie podczas tworzenia własnej implementacji generatora metod prywatnych, pozwala na zachowanie czystości kodu i odpowiedni podział na funkcjonalności. W ten sposób w pętli głównej programu znajdą się tylko metody wykonywane po kolei, jasno określając przebieg pracy. Pozwala to dokładnie zrozumieć zachodzący podczas generowania kodu proces. Jedne z metod mogą odpowiadać na przykład za odpytywanie użytkownika o dane, inne za wnioskowanie na podstawie jego odpowiedzi, kolejne zaś za weryfikację poprawności wprowadzonych danych lub użycie utworzonych obiektów podczas faktycznego generowania kodu. Yeoman organizuje takie metody-funkcjonalności, tworząc podstawową pętlę programową.

### 2.2.2 Pętla programowa

Wywoływanie metod generatora po kolei sprawdza się w większości przypadków, ale niekiedy aby zapewnić niezawodność, wymaganym jest aby jeszcze dokładniej określić porządek ich wywoływania. Umożliwione zostało to poprzez użycie wewnętrznie we frameworku modułu *Grouped-queue[[16]](#footnote-16)* i specjalnych, prototypowych metod o zarezerwowanych nazwach. Dopóki zadeklarowana przez użytkownika metoda ma nazwę odpowiadającą jednej z zarezerwowanych, jej wykonanie zostanie umieszczone na specjalnej kolejce. Ponadto, jeśli metoda nie odpowiada żadnej z priorytetowych nazw, zostanie dodana do grupy wykonania domyślnego  default .

Priorytetowe nazwy metod zarezerwowanych, w kolejności ich wykonania przez framework:

1. **initializing** – faza, w której w programie można zainicjalizować potrzebne zmienne, załadować konfigurację lub np. sprawdzić kontekst wykonania programu
2. **prompting** – odpytywanie użytkownika generatora, zapisując jego odpowiedzi w zmiennych programu do późniejszego użycia
3. **configuring** – miejsce na logikę, w którym na podstawie wcześniej zadeklarowanych odpowiedzi użytkownika można odpowiednio skonfigurować zmienne programu, na przykład operacje na napisach lub dostosowanie zmiennych do docelowego języka programowania
4. **default** – umieszczane są tutaj wszystkie funkcje, których nazwy nie pasują do priorytetowych i wykonywane po kolei
5. **writing** – faza, w której program faktycznie zapisuje pliki na dysku, kopiując je z szablonów a także operując na nich przy użyciu zmiennych zadeklarowanych wcześniej
6. **conflicts** – używana wewnętrznie przez framework, ale jednak dostępna do rozszerzenia przez programistę faza, w której rozwiazywane są konflikty na plikach już istniejących. Generator odpytuje użytkownika o wprowadzane zmiany, pokazując dokładnie różnice i pozwalając na zastanowienie się
7. **install** – etap, w którym powinno umieszczać się kod odpowiedzialny za instalację wygenerowanego projektu, uruchamianie zadań związanych z pobieraniem zależności i inne, zwykle z użyciem zewnętrznych bibliotek (npm, bower, grunt, gulp)
8. **end** – ostatni element pracy generatora, w którym można np. posprzątać po wykonywanych zadaniach, wyświetlić wiadomość pożegnalną

Stworzenie generatora zgodnie z zasadami, stosując odpowiednio zarezerwawane nazwy metod priorytetowych, pozwala na odpowiednie działanie generatorów w momencie ich zagnieżdzania. Yeoman udostępnia bowiem możliwość uruchamiania jednych generatorów w drugich, przy użyciu mechanizmu kompozycji (ang. *composability).* Polega to na wywołaniu metody generator.composeWith()  uruchamiajacej zależny generator.

Możliwe jest przekazanie do generatora dowolonych parametrów potrzebnych w jego pracy (np. nazwa modułu). O ile zachowana została konwencja odpowiedniego nazewnictwa kluczowych metod, framework zadba aby generator wykonał swoją pracę prawidłowo.

### 2.2.3 Zdarzenia asynchroniczne

W toku pracy generatora czasami potrzebna jest sytuacja, w której trzeba dokonać pauzy w jego działaniu. Jedną z takich sytuacji jest odpytywanie użytkownika w sekwencji o kilka różnych rzeczy. Pytania powinny pojawiać się po kolei, czekając na udzielenie odpowiedzi – nie wszystkie naraz. Odpowiedzi na jedne pytania mogą mieć wpływ na pojawianie się innych lub na ich treść. Istnieje kilka sposobów realizacji takiego scenariusza, jednym z nich może być używanie *promise[[17]](#footnote-17),* elementu nowoczesnego języka JavaScriptwprowadzonego w ramach standardu *ECMAScript 2015[[18]](#footnote-18)* lub używanie funkcji this.async() . Takie wywołanie powoduje zwrócenie funkcji, którą można wywołać ponownie po zakończeniu działania. Jeśli do wywołania zwracanej funkcji przekazany zostanie parametr błędu, pętla programowa zatrzyma się powodując wyjątek.

Tabela 1 Przykład wywołania synchronicznej operacji w funkcji asynchronicznej

function asynchronousFunction() {

var done = this.async();

synchronousOperation(done);

}

### 2.2.4 Z

# 3 Własna implementacja

## 3.1 Generator i podgenerator

## 3.2 Interakcja z użytkownikiem

## 3.3 Konfiguracja zmiennych programu

## 3.4 Zapisywanie plików i rozwiązywanie konfliktów

## 3.5 Efekt pracy generatora

## 3.6 Możliwości rozwoju

# 4 Podsumowanie

# 5 Bibliografia

<http://web.stanford.edu/class/cs99r/readings/parnas1.pdf>

http://cse360fall12.wikispaces.asu.edu/AutomatedCodeGroup38

Adele Mildred Koss, “Programming on the Univac 1: A Woman’s Account,” IEEE Annals of the History of Computing 25, no. 1 (January–March 2003): 56

https://adrianba.net/2015/03/14/using-yeoman-to-start-writing-technical-specifications-with-respec/

1. Saul Gorn – amerykański pionier informatyki, profesor Moore School na Uniwersytecie Pensylwanii, pracował nad pierwszymi komputerami: ENIAC i EDVAC [↑](#footnote-ref-1)
2. David Lorge Parnas – jeden z pionierów programowania, twórca koncepcji modularyzacji oprogramowania i ukrywania informacji, promotor etyki w wytwarzaniu oprogramowania, [↑](#footnote-ref-2)
3. SOFTWARE ASPECTS OF STRATEGIC DEFENSE SYSTEMS – kompilacja ośmiu krótkich esejów dołączonych do listu rezygnacyjnego Davida Panas, odchodzącego z panelu ds. użycia komputerów we wsparciu systemów bitewnych [↑](#footnote-ref-3)
4. Adele Mildred Koss, “Programming on the Univac 1: A Woman’s Account,” IEEE Annals of the History of Computing 25, no. 1 (January–March 2003): 56 [↑](#footnote-ref-4)
5. wyrażenia *lambda* – wprowadzona w Java SE 8 możliwość programowania w sposób bardziej funkcyjny [↑](#footnote-ref-5)
6. IntelliJ IDEA umożliwia na przykład zamianę kodu w języku Java na kod w języku Kotlin, zachowując jego logikę [↑](#footnote-ref-6)
7. *boilerplate code* – często powtarzający się kod źródłowy [↑](#footnote-ref-7)
8. https://github.com/yeoman/ [↑](#footnote-ref-8)
9. <http://yeoman.io/>generators [↑](#footnote-ref-9)
10. https://adrianba.net/2015/03/14/using-yeoman-to-start-writing-technical-specifications-with-respec/ [↑](#footnote-ref-10)
11. *Aplikacja monolityczna* – aplikacja typu wszystko-w-jednym, realizująca zarówno dostęp do bazy danych, zawierająca logikę biznesową, jak i prezentująca interfejs użytkownika [↑](#footnote-ref-11)
12. *Architektura mikroserwisowa* – architektura aplikacji podzielonej na luźno połączone moduły realizujące jasno określone funkcje, pozwalająca na niezależną pracę nad nimi przez wiele zespołów programistycznych [↑](#footnote-ref-12)
13. *Aplikacja jednostronna* – aplikacja, w której cała treść jest dynamicznie ładowana zawarta na jednej stronie [↑](#footnote-ref-13)
14. Programowanie sterowane zdarzeniami – paradygmat zakładający sterowanie programem poprzez zdarzenia, na które odpowiada aplikacja. Aplikacja reaguje tylko wtedy, kiedy nadejdzie nowe zdarzenie [↑](#footnote-ref-14)
15. <https://nodejs.org/en/docs/guides/blocking-vs-non-blocking/> - wyjaśnienie różnic pomiędzy blokującym i nieblokującym modelem I/O [↑](#footnote-ref-15)
16. https://github.com/SBoudrias/grouped-queue – system kolejki wewnątrz pamięci, stworzony przez Simona Boundrias, pozwalający na priorytetyzowanie zadań   
     [↑](#footnote-ref-16)
17. *promise* – specjalny obiekt języka JavaScript, pozwalający operować na zadaniach asynchronicznych w sposób synchroniczny [↑](#footnote-ref-17)
18. http://www.ecma-international.org/ecma-262/6.0/#sec-promise-objects [↑](#footnote-ref-18)