Министерство образования Республики Беларусь

# Учреждение образования «Белорусский государственный университет информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра программного обеспечения информационных технологий

Дисциплина: Основы алгоритмизации и программирования

# ПОЯСНИТЕЛЬНАЯ ЗАПИСКА к курсовой работе на тему

ПРОГРАММНОЕ СРЕДСТВО ГЕНЕРАЦИИ КОДА НА ЯЗЫКЕ ПРОГРАММИРОВАНИЯ GOLANG ДЛЯ ИСПОЛНЕНИЯ SQL ЗАПРОСОВ

# БГУИР КР 6-05-0612-01 116 ПЗ

Студент: гр. 451001 Пузик В. А.

Руководитель: асс. Фадеева Е.Е.

Минск 2025

## 1 Анализ прототипов, литературных источников и формирование требований к проектируемому программному средству

### 1.1 Обзор литературы

**1.1.1** [1] SQLite – sql syntax [Электронный ресурс]. – Режим доступа:

<https://www.sqlite.org/lang.html>

Структурно, при помощи РДНФ схем, описанный синтаксис декларативного языка программирования SQL (диалект SQLite). Использование данных схем поможет в обработке файлов в формате SQL и при написании документации.

**1.1.2** [2] docwiki.embarcadero [Электронный ресурс]. – Режим доступа: <https://docwiki.embarcadero.com/>

Подробное описание работы с delphi и RAD Studio. Документация по процедурам и функциям в языке delphi. Подробное описание создания приложений при помощи RAD Studio.

**1.1.3** [3] Статья «ORM и его альтернативы» – Режим доступа : <https://university.ylab.io/articles/tpost/ujitflvnl1-orm-i-ego-alternativi>

В статье приведены преимущества, недостатки и альтернативы ORM. Это позволит оценить сильные стороны проекта и его преимущества перед другими способами выполнения SQL запросов.

### 1.2 Примеры решения аналогичных задач

Рассмотрим уже существующие программные средства, решающие задачу исполнения SQL запросов в приложениях, написанных с использованием языка программирования Golang:

1. GORM  
   ORM библиотека для языка программирования GoLang.

Достоинства:

А) Удобство для быстрого прототипирования за счет обширного набора встроенных функций и автоматизации рутинных задач

Б) Поддержка ассоциаций и связей между моделями

В) Высокоуровневый подход (объектно‑ориентированный интерфейс для работы с данными)

Г) Автоматизация миграций

Недостатки:

А) Непредсказуемость запросов

Б) Низкая производительность

В) Ограниченные возможности для оптимизации

Г) Ошибки во время выполнения

1. sqlf

Конструктор запросов (query builder) для языка GoLang нацеленный на производительность во время исполнения программы.

Достоинства:

А) Большая гибкость в сравнении с ORM библиотеками

Б) Код лучше отображает итоговые SQL запросы

В) Быстрая скорость работы

Недостатки:

А) Плохая читаемость кода

Б) Отсутствие статической типобезопасности

В) Ограниченные возможности для оптимизации

1. jet

Конструктор запросов (query builder) для языка GoLang c функциональностью генерации кода структур данных в соответствии со схемой таблиц.

Достоинства:

А) Гибкость в динамическом формировании запросов

Б) Интеграция со средами разработки

В) Компиляторная проверка типов

Г) Итоговый код приближен к синтаксису SQL

Д) Генерация структур для схем данных

Недостатки:

А) Сложность отладки запросов из-за динамической генерации

Б) Ограничения DSL

В) Тесная связь бизнес-логики и SQL запросов

Г) Возможность ошибок во время исполнения в связи с ручным написанием схем данных для хранения результатов выполнения SQL запросов

### 1.3 Требования к проектируемому программному средству

**1.3.1** Назначение разработки:

Проектируемое программное средство предназначено для генерации модуля на языке программирования Golang, ответственного за исполнение SQL запросов. Оно должно обеспечивать типобезопасность итоговых функций и прозрачность выполняемых SQL запросов. Система должна обеспечивать возможность писать запросы непосредственно на SQL, снижать вероятность ошибок и минимизировать уровни абстракции при исполнении SQL запросов.

**1.3.2** Состав выполняемых функций**:**

1. Генерация структур на основе запросов CREATE TABLE;
2. Генерация функций, исполняющих SQL запросы;
3. Генерация типизированных структур параметров запросов;
4. Генерация типизированных структур результата выполнения запроса;
5. Настройка посредством JSON конфигурации.

**1.3.3** Входные данные:

1. Текстовый файл, содержащий SQL запросы CREATE TABLE.
2. Текстовые файлы, содержащие INSERT, UPDATE, SELECT, DELETE запросы.
3. Конфигурационный JSON файл

**1.3.4** Выходные данные:

1. Файл с кодом на языке программирования Golang с описанием структур таблиц;
2. Набор файлов, содержащих наборы функций, которые исполняют соответственные SQL запросы.

**1.3.5** Требования к временным характеристикам:

1. Генерация модуля должна происходить не дольше, чем за 5 секунд;
2. Время выполнения функции, исполняющей SQL запрос, не должно превышать время выполнения SQL запроса напрямую к базе данных больше, чем на 10 миллисекунд.

**1.3.6** Требования к надёжности:

Программное средство должно валидно обрабатывать синтаксис SQL с ограничениями, задекларированными в документации. При невозможности обработки определенной конструкции программное средство должно отображать понятную для разработчика ошибку.

**1.3.7** Условия эксплуатации:

Программное средство должно поддерживать работу на ОС Windows, MacOS, Linux.

**1.3.8** Язык и среда разработки:

Язык разработки: Delphi, благодаря его возможности создания высокопроизводительных и надежных приложений с консольным интерфейсом.

Среда разработки: Embarcadero RAD Studio, так как она предлагает мощные инструменты для визуального проектирования и отладки приложений на Delphi.

**1.3.9** Дополнительные требования:

Интерфейс взаимодействия должен быть простым и понятным для пользователей. Это включает в себя использование понятных терминов, логичную систему конфигурации и описания параметров для подстановки. Для описания успешнего результата генарции необходимо выводить соответсвующее сообщение пользователю. Необходимо дать пользователю возможность именовать функции для большего удобства доступа.

**2. Анализ требований к ПС и разработка функциональных требований (до 6 стр.)**

Инструмент командной строки, который будет автоматизировать процесс создания кода на языке Go для работы с базами данных. Система должна анализировать SQL-файлы со схемой базы данных и запросами, а затем генерировать соответствующие структуры данных и функции для взаимодействия с базой данных через стандартную библиотеку Go `database/sql`.

Классический пользовательский сценарий выглядит следующим образом:

1. Пользователь создает SQL-файл с определениями таблиц (CREATE TABLE).

2. Пользователь создает SQL-файл с запросами, содержащими специальные комментарии и параметры.

3. Пользователь создает конфигурационный файл, указывая путь к файлу схемы и другие параметры.

3. Пользователь запускает SQLRC с параметром `--cfg`, указывающим на конфигурационный файл.

4. Система анализирует файл схемы, извлекая информацию о таблицах и их структуре.

5. Система генерирует Go-код с определениями структур данных, соответствующих таблицам.

6. Система сохраняет сгенерированный код в файл `schema.go` в указанной директории.

7. Система анализирует файл запросов, извлекая информацию о запросах, их параметрах и типах возвращаемых значений.

8. Система генерирует Go-код с функциями для выполнения запросов, структурами для параметров и результатов.

9. Система сохраняет сгенерированный код в файл `query.go` в указанной директории.

10. Пользователь импортирует сгенерированный пакет в свой Go-проект.

11. Пользователь создает экземпляр структуры `Queries`, передавая ей структуру, соблюдающую интерфейс `sql.DB` из стандартной библиотеки GoLang - `database/sql`.

12. Пользователь вызывает сгенерированные функции для выполнения запросов к базе данных.

13. Система обеспечивает типобезопасное взаимодействие с базой данных через сгенерированный код.

**2.3 Спецификация функциональных требований.**

1. Система должна читать конфигурационный файл в формате "<ключ>:<значение><перевод\_строки>".
2. Система должна корректно обрабатывать относительные пути в конфигурационном файле. Текущей рабочей директорией является директория, в которой находится конфигурационный файл.
3. Система должна выдавать понятные сообщения об ошибках при отсутствии обязательных параметров или некорректном формате конфигурационного файла.
4. Система должна читать SQL-файл с определениями таблиц (CREATE TABLE).
5. Система должна корректно обрабатывать комментарии в SQL-коде, игнорируя их при анализе.
6. Система должна корректно извлекать имена таблиц из операторов CREATE TABLE.
7. Система должна корректно извлекать имена, типы и атрибуты полей таблиц.
8. Система должна поддерживать следующие типы данных SQL:

- TEXT - соответствует типу string в Go

- INTEGER - соответствует типу int в Go

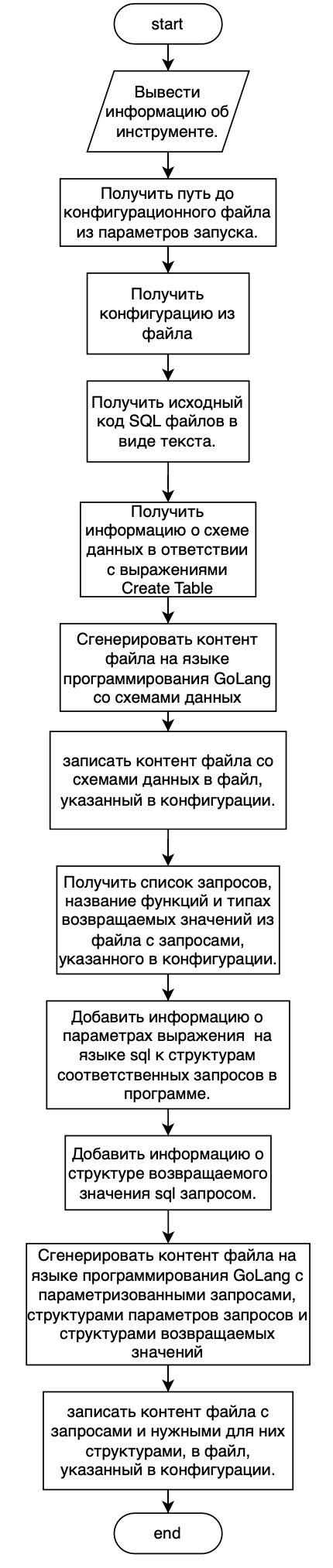
1. Система должна определять, является ли поле обязательным (NOT NULL) или опциональным.
2. Система должна корректно обрабатывать первичные ключи (PRIMARY KEY).
3. Система должна выдавать понятные сообщения об ошибках при обнаружении неподдерживаемых типов данных или несоблюдения ограничений, накладываемых инструментом в отношении синтаксиса SQL.
4. Система должна распознавать специальные комментарии формата  
   `--@ sqlrc:<ИмяФункции>:<ТипВозврата>`
5. Система должна поддерживать следующие типы возвращаемых значений:  
   `one` - запрос, возвращающий одну запись

`many` - запрос, возвращающий несколько записей

`exec` - запрос, не возвращающий данные

1. Система должна распознавать параметры запросов в формате `<@<имя\_параметра>:<тип\_параметра>@>`
2. Система должна корректно обрабатывать повторяющиеся параметры в запросе, заменяя их на нужное поле структуры параметров в сгенерированном Go-коде.
3. Система должна анализировать результаты запросов для определения структуры возвращаемых данных.
4. Система должна выдавать понятные сообщения об ошибках при обнаружении неподдерживаемых типов параметров или некорректного синтаксиса описания параметров.
5. Для каждой таблицы должна быть сгенерирована отдельная структура с полями, соответствующими полям таблицы.
6. Имена структур должны быть в формате PascalCase, с первой буквой в верхнем регистре.
7. Поля структур должны иметь теги `db` с именами соответствующих полей таблицы для совместимости с другими инструментами.
8. Сгенерированный код должен быть корректным Go-кодом, который компилируется без ошибок.
9. Для каждого запроса должна быть сгенерирована константа с SQL-запросом, в котором параметры заменены на плейсхолдеры `?`.
10. Для каждого запроса с параметрами должна быть сгенерирована структура с полями, соответствующими параметрам запроса.
11. Для запросов типа `one` и `many` должна быть сгенерирована структура с полями, соответствующими результатам запроса.
12. Для каждого запроса должна быть сгенерирована функция, принимающая контекст и структуру параметров и возвращающая результаты запроса.
13. Сгенерированный код должен корректно обрабатывать NULL-значения в результатах запросов.
14. Сгенерированный код должен быть корректным Go-кодом, который компилируется без ошибок.
15. Система должна выдавать понятные сообщения об ошибках при невозможности создания директории или записи файлов.
16. Система должна предоставлять интерфейс командной строки с параметром `--cfg <путь\_к\_конфигурационному\_файлу>`.
17. Система должна выводить информацию о ходе выполнения, включая сообщения о чтении файлов, генерации кода и записи файлов.

**3** **Проектирование программного средства**



**4** **Создание** (**конструирование) программного средства**

1. Главный модуль

- Точка входа в приложение

- Координирует работу всех модулей

- Обрабатывает аргументы командной строки

- Управляет процессом генерации кода

2. Модуль типов данных

- Определяет все типы данных, используемые в проекте

- Содержит структуры для конфигурации, схемы, запросов

- Предоставляет функции для преобразования типов

3. Модуль файловой системы

- Отвечает за чтение и запись файлов

- Обрабатывает конфигурационный файл

- Создает директории для сгенерированных файлов

- Записывает сгенерированный код в файлы

Модули парсинга

4. Парсер схемы

- Анализирует SQL-файл с определениями таблиц (CREATE TABLE)

- Извлекает имена таблиц и их структуру

- Определяет типы полей и их атрибуты

- Обрабатывает комментарии в SQL-коде

5. Парсер запросов

-Анализирует SQL-файл с запросами

-Распознает специальные комментарии формата `--@ sqlrc:ИмяФункции:ТипВозврата`

- Разделяет файл на отдельные запросы

- Определяет имена и типы запросов

6. Парсер параметров запросов

- Анализирует параметры запросов в формате `<@имя\_параметра:тип\_параметра@>`

- Определяет имена и типы параметров

- Обрабатывает повторяющиеся параметры

- Заменяет параметры на плейсхолдеры в SQL-запросах

7. Парсер результатов запросов

- Анализирует результаты запросов для определения структуры возвращаемых данных

- Связывает поля результатов с полями таблиц

- Определяет типы возвращаемых значений

Модули генерации

8. Генератор схемы

- Генерирует Go-код с определениями структур данных для таблиц

- Создает структуру `Queries` с полем `DB` типа `\*sql.DB`

- Преобразует типы данных SQL в типы Go

- Добавляет теги `db` к полям структур

9. Генератор запросов

- Генерирует Go-код с функциями для выполнения запросов

- Создает константы с SQL-запросами

- Генерирует структуры для параметров запросов

- Координирует работу генераторов для разных типов запросов

10. Генератор для запросов типа "one"

- Генерирует функции для запросов, возвращающих одну запись

- Создает структуры для результатов запросов

- Генерирует код для сканирования результатов запроса

11. Генератор для запросов типа

- Генерирует функции для запросов, возвращающих несколько записей

- Создает структуры для результатов запросов

- Генерирует код для итерации по результатам запроса

12. Генератор для запросов типа "exec"

- Генерирует функции для запросов, не возвращающих данные

- Генерирует код для выполнения запросов и обработки результатов

**5** **Тестирование, проверка работоспособности и анализ полученных результатов**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Специфика тестирования | Номер теста | Вводимые данные | Ожидаемый результат | Полученный результат |
| Парсинг полного конфигурационного файла | 1.1 | schema:schema.sql  queries:user.sql  remove\_trailing\_s:true  pakage.name:gen  pakage.path:./\_\_gen/ | Система корректно прочитает конфигурационный файл и извлечет все параметры |  |
| Парсинг конфигурационного файла с отсутствующими параметрами | 1.2 | schema:schema.sql  remove\_trailing\_s:true  pakage.name:gen | Система выдаст сообщение об ошибке, указывающее на отсутствие обязательного параметра "queries" |  |
| Парсинг схемы с поддерживаемыми типами данных | 2.1 | CREATE TABLE users (primary\_currency TEXT NOT NULL DEFAULT 'BYN', username TEXT NOT NULL UNIQUE, password TEXT NOT NULL, image TEXT NOT NULL, id INTEGER PRIMARY KEY NOT NULL, balance INTEGER NOT NULL DEFAULT 0); | Система корректно распознает таблицу "users" с 6 полями и их типами |  |
| Парсинг схемы с неподдерживаемыми типами данных | 2.2 | CREATE TABLE users (email REAL NOT NULL); | Система выдаст сообщение об ошибке, указывающее на неподдерживаемый тип данных "REAL" |  |
| Парсинг запроса с одним параметром | 3.1 | --@ sqlrc:GetSingle:one  SELECT \*  FROM users  WHERE username = <@username:string@>; | Система корректно распознает запрос с именем "GetSingle", типом "one" и параметром "username" типа "string" |  |
| Парсинг запроса с повторяющимся параметром | 3.2 | --@ sqlrc:GetRepeated:one\nSELECT id FROM users WHERE id = <@id:int@> AND <@id:int@> < 10; | Система корректно распознает запрос с повторяющимся параметром "id" |  |
| Парсинг запроса с неподдерживаемым типом возвращаемого значения | 3.3 | --@ sqlrc:GetSingle:invalid\nSELECT \*\nFROM users\nWHERE username = <@username:string@>; | Система выдаст сообщение об ошибке, указывающее на неподдерживаемый тип возвращаемого значения "invalid" |  |
| Парсинг запроса с неподдерживаемым типом параметра | 3.4 | --@ sqlrc:GetSingle:one \nSELECT \*\nFROM users\nWHERE username = <@username:invalid@>; | Система выдаст сообщение об ошибке, указывающее на неподдерживаемый тип параметра "invalid" |  |
| Генерация структуры для таблицы | 4.1 | { Name: "users", Columns: [ { Name: "primary\_currency", ColumnType: ctText, Nullable: false }, { Name: "username", ColumnType: ctText, Nullable: false }, { Name: "password", ColumnType: ctText, Nullable: false }, { Name: "image", ColumnType: ctText, Nullable: false }, { Name: "id", ColumnType: ctInteger, Nullable: false }, { Name: "balance", ColumnType: ctInteger, Nullable: false } ] } | Система сгенерирует структуру Go для таблицы "users" с корректными типами полей |  |
| Генерация функции для запроса типа "one" | 4.2 | { QueryToken: { SQL: "SELECT \* FROM users WHERE username = ?", Name: "GetSingle", ReturnType: qrtOne }, Params: [ { Name: "username", ParamType: otString, Positions: [0] } ], Results: [ { TableName: "users", Fields: [ { TableField: "primary\_currency", ReturningName: "primary\_currency" }, { TableField: "username", ReturningName: "username" }, { TableField: "password", ReturningName: "password" }, { TableField: "image", ReturningName: "image" }, { TableField: "id", ReturningName: "id" }, { TableField: "balance", ReturningName: "balance" } ] } ] } | Система сгенерирует функцию Go для выполнения запроса "GetSingle" с корректными параметрами и возвращаемыми значениями |  |
| Генерация функции для запроса типа "many" | 4.3 | { QueryToken: { SQL: "SELECT users.\* FROM users WHERE id < ?", Name: "GetMany", ReturnType: qrtMany }, Params: [ { Name: "id", ParamType: otInteger, Positions: [0] } ], Results: [ { TableName: "users", Fields: [ { TableField: "primary\_currency", ReturningName: "primary\_currency" }, { TableField: "username", ReturningName: "username" }, { TableField: "password", ReturningName: "password" }, { TableField: "image", ReturningName: "image" }, { TableField: "id", ReturningName: "id" }, { TableField: "balance", ReturningName: "balance" } ] } ] } | Система сгенерирует функцию Go, возвращающую срез структур результата |  |
| Генерация функции для запроса типа "exec" | 4.4 | { QueryToken: { SQL: "INSERT INTO users (username, password, image) VALUES (?, ?, ?)", Name: "InsertUser", ReturnType: qrtExec }, Params: [ { Name: "username", ParamType: otString, Positions: [0] }, { Name: "password", ParamType: otString, Positions: [1] }, { Name: "image", ParamType: otString, Positions: [2] } ], Results: [] } | Система сгенерирует функцию Go, возвращающую результат выполнения запроса |  |
| Проверка времени генерации модуля | 5.1 | Конфигурационный файл, файл схемы с 3 таблицами и файл запросов с 5 запросами   * time sqlrc --cfg ./example/config.txt | Время генерации модуля не превышает 5 секунд |  |
| Проверка времени выполнения сгенерированной функции | 5.2 | Сгенерированная функция GetSingle и база данных SQLite  Код для замеров:  func Wrapper (f…) {  defer TimeTrack(time.Now())  f ()  }  func TimeTrack(start time.Time) {  elapsed := time.Since(start)  pc, \_, \_, \_ := runtime.Caller(1)  funcObj := runtime.FuncForPC(pc)  runtimeFunc := regexp.MustCompile(`^.\*\.(.\*)$`)  name := runtimeFunc.ReplaceAllString(funcObj.Name(), "$1")  log.Println(fmt.Sprintf("%s took %s", name, elapsed))  } | Время выполнения функции не превышает время выполнения SQL запроса напрямую к базе данных больше, чем на 10 мс |  |
| Проверка интеграции сгенерированного кода в проект | 6.1 | Сгенерированный код в проекте Go | Код компилируется без ошибок и корректно взаимодействует с базой данных | пройден |