Двадцатая международная конференция

«СОВРЕМЕННЫЕ ПРОБЛЕМЫ ДИСТАНЦИОННОГО ЗОНДИРОВАНИЯ ЗЕМЛИ ИЗ КОСМОСА»

14 – 18 ноября 2022, Москва

Декомпозиция, дизайн и архитектура программного обеспечения для спектрофотометра Брюэра

Савиных В. В.

Институт физики атмосферы им. А. М. Обухова РАН, Москва, Россия

E-mail: amita@ifaran.ru

Введение

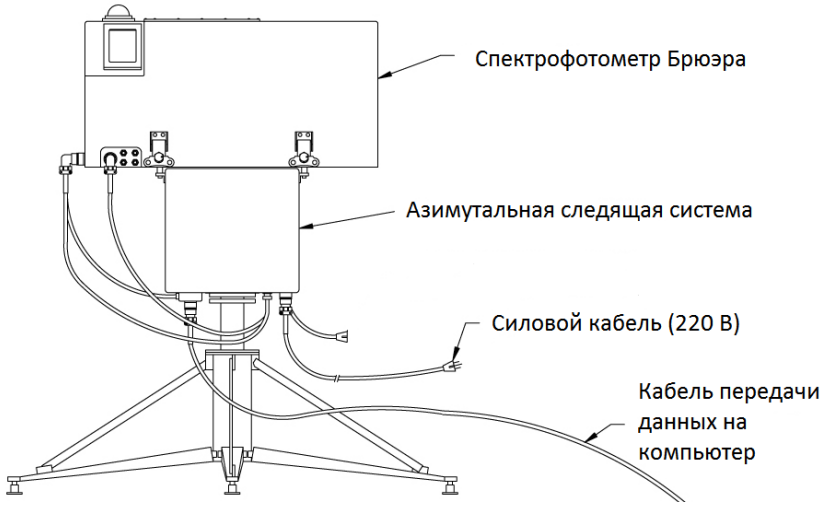

Рис 1. С/ф Brewer для измерений общего содержания O_3 и УФ радиации (рис. Kipp & Zonen)

Рис. 2. Спектрофотометры Brewer по миру; зелёный – Brewer MkII #043 КВНС ИФА (рис. Kipp & Zonen)

Озон (O_3) участвует во многих фотохимических реакциях и поглощает попадающее в стратосферу солнечное УФ излучение; уменьшение содержания озона вызывает изменения климата Земли [1]. Спектрофотометры Брюэра озонометрической сети (рис. 1) измеряют спектральный состав солнечного излучения при любых погодных условиях в коротковолновом УФ диапазоне [2]. Около 80 наземных станций в 40 странах мира оснащены более чем 200 с/ф Брюэра (рис. 2). Для наблюдений на Брюэре необходимо программное обеспечение (ПО) с современным дизайном [3].

Требования

Все программные системы имеют *требования* – функциональные особенности, которым должна соответствовать программа [4]. Требования к ПО для Брюэра:

- хранение данных, основанное на стеке, размер которого неограничен;
- возможность отмены и повторения операций;
- возможность удаления элемента из вершины стека и очистки всего стека;
- возможность сохранения данных на диске в файлах и реляционной БД;
- реализация основных операций при работе со с/ф Брюэра;
- реализация подключаемых модулей для расширения основных операций;
- реализация консольного и графического интерфейса пользователя (UI);
- наличие отказоустойчивости при неверном вводе пользователя;
- реализация пакетной работы и хранимых процедур (сценариев операций).

Дизайн

Дизайн ПО разрабатывается методом иерархической декомпозиции. Атрибутами хорошей декомпозиции являются *модульность*, *инкапсуляция*, высокая *связность* и низкая *связанность*. Модульность позволяет разделить сложную задачу на более мелкие компоненты. Инкапсуляция подразумевает, что внутренняя реализация модуля остаётся скрытой от других модулей, а взаимодействие между ними осуществляется через интерфейсы.

Связность означает, что код внутри модуля должен быть самосогласованным. Дизайн, смешивающий графический интерфейс с обработкой данных, не будет связным. Связанность – это, когда логический поток одного модуля требует вызова другого модуля для завершения своего действия. Низкая связанность обеспечивает взаимодействие модулей через чётко определённые интерфейсы.

Архитектура

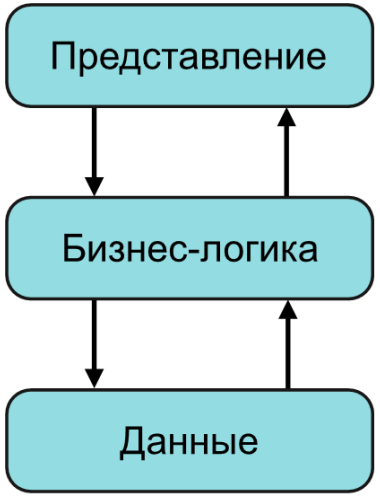

Рис. 3. Трёхуровневая архитектура (стрелки указывают взаимодействие).

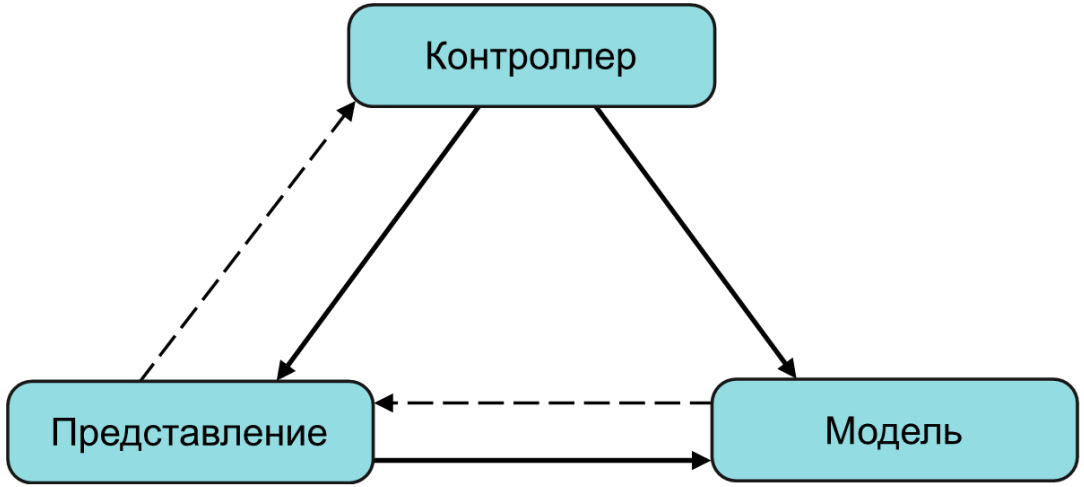

Рис. 4. Архитектура MVC. Сплошные линии – прямое взаимодействие, пунктирные – косвенное.

Разделение проекта на составные компоненты осуществляется с помощью паттернов проектирования (концептуальных шаблонов для схожих задач на уровне классов) [5] и архитектурных паттернов (стратегий проектирования всей системы). Небольшие проекты используют две архитектуры: *трёхуровневую* [6] когда интерфейс пользователя, логика и данные приложения взаимодействуют через смежные уровни (рис. 3), и *модель–представление–контроллер* (MVC) [7], когда модель абстрагирует данные, представление – UI, а контроллер управляет взаимодействием между моделью и представлением (рис. 4).

Архитектура

Важно отметить, что для обеих архитектур представление отвечает только за принятие команд, а не за их интерпретацию и выполнение, не смешивая уровни представления и логики.

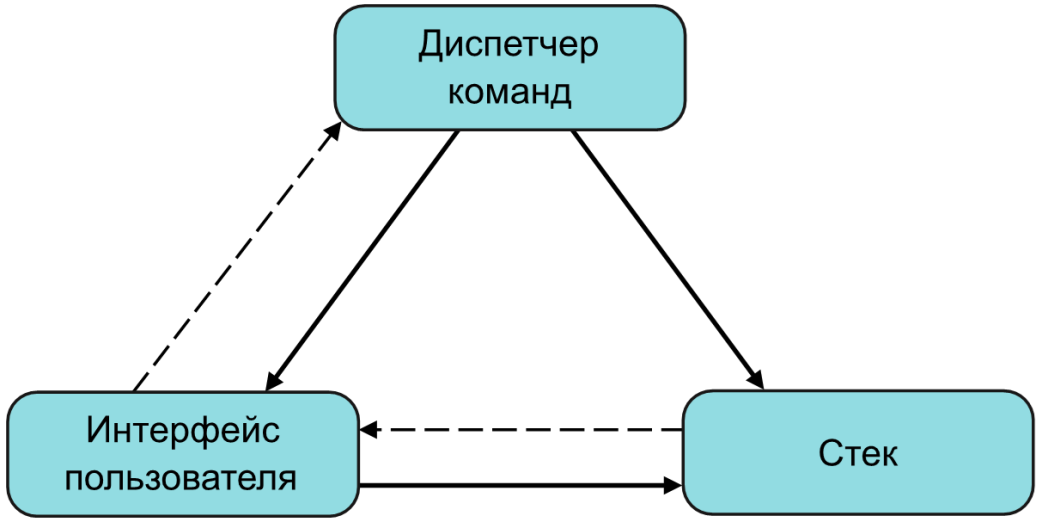

Рис. 5. Архитектура приложения Брюэра.

Приложение для Брюэра реализует архитектуру MVC, в которой *стек* (модель) действует как хранилище данных измерений, отображаемых пользователю в *интерфейсе пользователя* (представление), а *диспетчер команд* (контроллер) выполняет все операции с данными (рис. 5).

В архитектуре MVC *представление* может генерировать события для обработки *контроллером* (пунктирная линия) и получать данные для отображения прямо из *модели* (сплошная линия). Контроллер обрабатывает события от представления и напрямую управляет моделью и представлением. Модель генерирует события, обрабатываемые представлением для обновления своего содержимого.

Интерфейсы

Чтобы охватить все потоки данных в приложении были задействованы *сценарии использования*, они определяют действия пользователя с системой и рабочие процессы [8]. С их помощью обнаруживаются взаимодействия между модулями стека, UI и диспетчера команд для создания открытых интерфейсов модулей. Сценарии использования ПО для Брюэра согласно *требованиям* следующие:

- ввод данных в стек – сообщение об ошибке при недостоверных данных;
- отмена/повтор последней операции – сообщение об ошибке при неудаче;
- удаление верхнего элемента/очистка стека – ошибка при пустом стеке;
- сохранение данных в файле/БД – ошибка, если данные не удалось сохранить;
- выполнение операций с/ф Брюэра – ошибка, если операция не выполнена;
- загрузка плагина/хранимой процедуры – ошибка при неудачной загрузке.

Интерфейсы

В таблице перечислены общедоступные интерфейсы модулей:

Модули	Функции	События
Интерфейс пользователя	`void postMessage(const string)` `void onStackChanged()`	`commandEntered(string)`
Диспетчер команд	`void onCommandEntered(const string)`	`error(string)`

Стек	`void push(Data)`
	`Data pop()`
	`List<Data> getElements(int)`
		`stackChanged()`
		`stackError()`

Функция модуля стека `push()` помещает данные в стек. Стек сигнализирует о своём изменении (рис. 5), а интерфейс пользователя запрашивает из стека *N* элементов для отображения. Этот маршрут добавляет событие `stackChanged()` и функцию `getElements()` к модулю стека, а к модулю интерфейса пользователя – обработчик события `onStackChanged()`.

Интерфейсы

Диспетчер команд проверяет данные перед помещением в стек и сигнализирует интерфейсу пользователя об ошибке, которую UI обрабатывает. Этот маршрут добавляет событие `error()` к диспетчеру команд и функцию `postMessage()` в UI. Для извещения о вводе команды к UI добавляется событие `commandEntered()`, а к диспетчеру команд обработчик события `onCommandEntered()`; аргументом для этой пары событие/обработчик является строка, кодирующая команду.

Для извлечения данных к модулю стека добавляется функция `pop()`. Извлечение из пустого стека вызывает ошибку, это добавляет к стеку событие `stackError()`, чтобы отразить событие ошибки в диспетчере команд.

Интерфейсы

Реализацию интерфейса модуля хранилища данных (стека) на C# [9] демонстрирует следующий код:

```
public sealed class Storage<T> {
  private static readonly Storage<T> instance = new();           // синглетон (гарантирует существование только одного экземпляра)
  private readonly Stack<T> stack = new();                       // контейнер для хранения данных

  private Storage() { }

  public event EventHandler? StorageChanged;                     // событие изменения стека
  private void OnStorageChanged(EventArgs e) { StorageChanged?.Invoke(this, e); }

  public void Push(T data, bool suppressChangeEvent = false) {   // помещает данные в стек
    stack.Push(data);
    if (!suppressChangeEvent) OnStorageChanged(EventArgs.Empty);
  }

  public T Pop(bool suppressChangeEvent = false) {               // извлекает данные из стека
    var data = stack.Pop();
    if (!suppressChangeEvent) OnStorageChanged(EventArgs.Empty);
    return data;
  }

  public void GetElements(int n, List<T> list) {                 // запрашивает n элементов из стека
    if (n > stack.Count) n = stack.Count;
    list.AddRange(stack.ToList().GetRange(0, n));
  }

  public List<T> GetElements(int n) {                            // вспомогательный метод
    var list = new List<T>();
    GetElements(n, list);
    return list;
  }

  public static Storage<T> Instance => instance;                 // реализация синглетона
}
```

Заключение

Полученный дизайн ПО для с/ф Брюэра является модульным и связным:

- интерфейс пользователя принадлежит модулю UI;
- функциональная логика – диспетчеру команд;
- управление данными – модулю стека;
- каждый модуль инкапсулирует все свои функции;
- модули слабо связаны между собой, связь осуществляется через общедоступные интерфейсы;
- архитектура верхнего уровня соответствует известному паттерну MVC.

Работа выполнена в ходе исследования №0129-2019-0002.

Литература

1. *IPCC.* Climate Change 2014: Synthesis Report. Contribution of Working Groups I, II and III to the Fifth Assessment Report of the Intergovernmental Panel on Climate Change, Core Writing Team, Pachauri, R.K., Meyer, L.A. (Eds.). IPCC, Geneva, Switzerland, 2014. 151 p.
2. *Kipp & Zonen.* Brewer MkIII Spectrophotometer Operator's Manual, Rev F. Kipp&Zonen, Delft, 2015. 123 p.
3. *Singer A.B.* Practical C++ Design: From Programming to Architecture, 2nd Ed. Apress, NY, 2021. 312 p.
4. *Wiegers K.E.* Software Requirements, 3rd Ed. Microsoft Press, Redmond, 2003. 672 p.
5. *Gamma E., Helm R., Johnson R., Vlissides J.* Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Boston, 1994. 416 p.
6. *Wikipedia.* Multitier architecture. https://en.wikipedia.org/wiki/Multitier_architecture.
7. *Wikipedia.* Model–view–controller. https://en.wikipedia.org/wiki/Model–view–controller.
8. *Booch G., Rumbaugh J., Jacobson I.* The Unified Modeling Language User Guide, 2nd Ed. Addison-Wesley, Boston, 2005. 475 p.
9. *Albahari J.* C# 10 in a Nutshell: The Definitive Reference. Sebastopol: O'Reilly, 2022. 1058 p.

Благодарю за внимание!