Автоматный интерфейс

Новый метод создание логики интерфейса

Введение

Сегодня существует очень мало «не серверных» программ без пользовательского интерфейса. Нельзя сказать, что их нет совсем, но все-таки любая программа, претендующая на то, чтобы быть «удобной для рядового пользователя» должна обладать графическим пользовательским интерфейсом. При этом создание пользовательского интерфейса не настолько хорошо освещено, как разработка программного обеспечения. К сожалению, многим из-за этого кажется, что пользовательский интерфейс просто должен быть, и это одна из причин, почему на выходе получается так много плохо оформленных программ.

Вторая причина, на взгляд автора, это неудобство создания интерфейсов. Рассмотрим сегодняшнее положение дел в этой области разработки программного обеспечения.

Сегодняшние методы создания интерфейса

Можно рассматривать два совершенно разных метода создания интерфейсов. Первый – это классический, «ручками». То есть пишется код, который отвечает за создание элементов интерфейса, за обработку пользовательских событий. Это не всегда быстро, но такой способ дает максимальный контроль создания интерфейса и обработки событий. Второй – «графической редактор». То есть программный продукт, который позволяет нарисовать интерфейс, мгновенно наблюдая то, как он будет отображаться (*RAD*, *Rapid Application Development, «Быстрая разработка приложений»*). Данный метод гораздо нагляднее, но почти всегда менее гибок.

Оба метода прививают несколько совершенно разных, но на взгляд автора некорректных привычек. Кодирование руками – это все-таки кодирование. Много внимания у хорошего программиста отнимает качество кода, которое совершенно не нужно конечному пользователю, хоть и очень сильно влияет на качество программы. А удобству интерфейса, соответственно, достается меньше внимания, что сказывается на конечном продукте. С другой стороны «графический» метод позволяет аккуратно расставить компоненты внутри окошка, панели или фрейма. Это хорошо, но при этом далеко не всегда разработчик уделяет соответствующее внимание кодированию, например, обработчиков событий. Или не следит за возможными изменениями интерфейса (изменение размеров окошек, шрифтов, и так далее – это обычно не отслеживается автоматически, а разработчик не уделяет проблеме должного внимания).

Получается, что красиво и, одновременно, качественно (с точки зрения программного кода) никак? В данной статье предложен способ создания обработчиков событий, который позволяет автоматизировать этот, без сомнения сложный этап создания интерфейса.

Общие принципы

Если вспомнить, что такое конечный автомат и сравнить с тем, что происходит во время взаимодействия с пользователем, то можно понять, что в этих двух сущностях достаточно много общего. За состояния можно принять состояние интерфейса (нажата или отпущена кнопка, поставлен флажок или нет, выбрана первая закладка или вторая, и так далее), пользовательские события могут запускать автомат. Например, на рис. 1 представлен граф переходов автомата обычного диалогового окна ввода, если вводить нужно число (рис. 2). Пунктиром представлены события, сплошными линиями – состояния автомата. Для краткости и наглядности в данном примере в качестве «названий» состояний автомата взяты действия, выполняемые автоматом в данном состоянии.

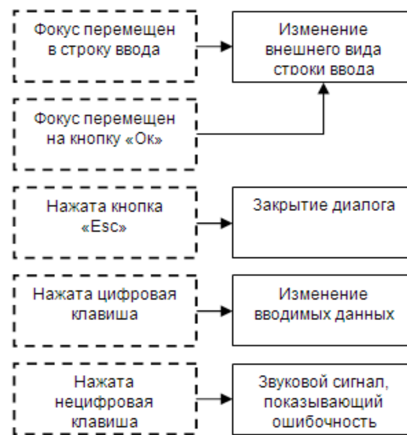

Рис. 1. Граф переходов автомата обработки событий диалогового окна, представленного на рис. 2.

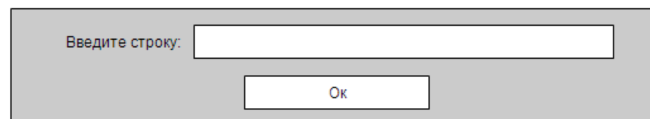

Рис. 2. Диалог

Как можно заметить, некоторые состояния (в данном случае «Изменение внешнего вида строки») используются повторно. Для полной реализации обработки событий интерфейса теперь нужно только лишь реализовать каждое состояние в виде кода и дать системе автоматически создать обработку событий с вызовами написанного кода. Поскольку каждый обработчик слабо связан с остальными, есть возможность разделения труда программистов. А так как он достаточно прост (только выполнение определенных действий, не создание логики), то это можно поручить малоопытному программисту («кодеру»), который сделает это совершенно без вреда для качества.

Также в случае такого разбиения логики интерфейса гораздо проще проводить тестирование отдельных состояний, переходов. Поскольку на графе видны все возможные входные данные, переходы показывают, как должны развиваться события, то и автоматическое тестирование также возможно.

При этом получается граф состояний пользовательского интерфейса, который достаточно легко создавать («рисуя»), который нагляден, просто преобразуется в код (вручную остается лишь написать код самих состояний, которые гораздо проще обработчиков событий). При этом преобразование происходит автоматически, что полностью исключает возможность ошибки.

Также улучшается наглядность представления обработчиков событий, появляется возможность повторного использования, как и использование шаблонов, что, несомненно, должно повлечь за собой улучшение качества логики разрабатываемого интерфейса.

Библиотека jInterface

Библиотека *jInterface* разработана с целью опробования возможностей описанного подхода при создании логики работы интерфейсов.

Структура библиотеки:

- `conditions` – пакет стандартных классов-условий, которые проверяются при проверке наличия «сработавших» переходов автомата.
    - `ICondition` – интерфейс условия.
    - `Always` – самое простое условие, которое выполнено всегда.
- `listeners` – переопределение стандартных «слушателей» для того, чтобы иметь возможность добавлять автоматную обработку событий в стандартном стиле AWT/Swing.
    - `Listener` – абстрактный класс-предок всех «автоматных классов слушатетей».
    - AutoFocusListener

   - ...

- `logging` – пакет, в котором располагаются некоторые вспомогательные классы, предназначенные для протоколирования создания и работы автомата.
   - `ILogger` – общий интерфейс
   - `SimpleLogger` – класс вывода протокола в стандартный вывод.
- `Automata` – класс автомата. Он обрабатывает все события, поступающие от слушателей.
- `Event` – событие, поступающее от слушателя.
- `IActions` – действия, которые могут исполняться при попадании в состояние или при переходе.
- `State` – состояние автомата.
- `Transition` – переход из состояния в состояние.
- `XMLAutomata` – класс, который расширяет автомат возможностью инициализации при помощи XML-файла.

Процесс создания автомата *состоит* из создания состояний, переходов, условий этих переходов и последующей «привязки» автомата к интерфейсу. Это осуществляется при помощи обычных событий *AWT* или *Swing*. При этом все события, которые необходимо обрабатывать автоматом, передают ему управление. На листинге 1 представлен фрагмент слушателя событий получения/потери фокуса (*FocusListener*).

Листинг 1. Класс слушателя событий, связанных с фокусом

```
public class AutoFocusListener extends Listener implements FocusListener {
    private boolean _actOnGained = false;
    private boolean _actOnLost = false;

    public AutoFocusListener(Automata aAutomata, State aStartState) {
        super(aAutomata, aStartState);
    }

    public void addListenerToComponent(Component aComponent, String aListenerType) {
        if (aListenerType.indexOf("gained") != -1) {
            _actOnGained = true;
        }

        if (aListenerType.indexOf("lost") != -1) {
            _actOnLost = true;
        }

        aComponent.addFocusListener(this);
    }

    public void focusGained(FocusEvent aEvent) {
        if (_actOnGained) {
            getAutomata().processEvent(new Event(getStartState(), aEvent));
        }
    }

    public void focusLost(FocusEvent aEvent) {
        if (_actOnLost) {
            getAutomata().processEvent(new Event(getStartState(), aEvent));
        }
    }
}
```

Автомат создается по информации, которая содержится в специальном *XML*-файле. При его загрузке инициализируются объекты и создается структура автомата (состояния, переходы, события). На листинге 2 представлен *XML*-файл, который соответствует несколько более сложному автомату, чем представленный на рис. 1 (введено поле пароля, немного другая логика обработки событий)

При разработке библиотеки *jInterface* не было создано программы для «рисования» графа логики интерфейса, но это было сделано исключительно из соображений экономии времени, так как приличный редактор графов создавать очень и очень трудно.

```
<automata>
    <rootComponent>com.jdnevnik.ainterface.mock.LoginFrame</rootComponent>

    <transition name="BadValue" targetStateName="EnteredErrorData"/>
    <transition name="Exit" targetStateName="ExitDialog"/>

    <event component="textFieldLogin" class="focus" type="gained"
                                        startStateName="ChangeColors"/>
    <event component="passwordField" class="focus" type="gained"
                                        startStateName="ChangeColors"/>
    <event component="buttonOk" class="focus" type="gained"
                                        startStateName="ChangeColors"/>
    <event component="buttonCancel" class="focus" type="gained"
                                        startStateName="ChangeColors"/>

    <event component="buttonOk" class="action"
                                        startStateName="ProcessLogin"/>
    <event component="buttonCancel" class="action"
                                        startStateName="ExitDialog"/>

    <event component="textFieldLogin" class="key" type="typed"
                                        startStateName="ChangeEnteredData"/>

    <state name="ChangeColors"
          actions="com.jdnevnik.ainterface.loginFrame.actions.ChangeColors"/>
    <state name="ProcessLogin"
          actions="com.jdnevnik.ainterface.loginFrame.actions.ProcessLogin">
        <condition class="com.jdnevnik.ainterface.conditions.Always"
                                                    transition="Exit"/>
    </state>
    <state name="ExitDialog"
          actions="com.jdnevnik.ainterface.loginFrame.actions.ExitDialog"/>
    <state name="ChangeEnteredData"
          actions="com.jdnevnik.ainterface.loginFrame.actions.ChangeEnteredData">
        <condition
           class="com.jdnevnik.ainterface.loginFrame.conditions.EnteredBadValue"
           transition="BadValue"/>
    </state>
    <state name="EnteredErrorData"
          actions="com.jdnevnik.ainterface.loginFrame.actions.EnteredErrorData"/>
</automata>
```

При получении события, автомат добавляет его в очередь событий. Поскольку *Swing* – это однопоточная библиотека, автор не стал рисковать и создавать многопоточный автомат. Тем более что это породило бы огромное количество проблем. В результате, события обрабатываются в одном потоке в том порядке, в котором они поступили на исполнение (Листинг 3).

Листинг 3. Фрагмент кода класса автомата. Добавление события в очередь.

```
    public void processEvent(Event event) {
        _eventsQueue.add(event);
        synchronized (MONITOR) {
            MONITOR.notifyAll();
        }
    }
```

Каждое состояние может содержать код, выполняемый при попадании в это состояние и список переходов с условиями. После выполнения кода, проверяются все условия переходов и выполняется переход по первому условию, которое оказывается выполненным (Листинг 4).

Листинг 4. Фрагмент кода класса автомата. Обработка состояния.

```
    private void startAutomata() {
        while (_currentState != null) {
            _currentState.processActions(this);
            Transition transition = _currentState.getTransition(this);
            if (transition != null) {
                transition.activate(this);
            } else {
                _currentState = null;
            }
        }
    }
```

В процессе перехода выполняется код, который может ему соответствовать, и происходит сам переход в новое состояние (Листинг 5).

Листинг 5. Фрагмент кода класса перехода. Обработка перехода.

```
public void activate(Automata aAutomata) {
    if (_actions != null) {
        _actions.process(aAutomata);
    }
    aAutomata.setCurrentState(_targetState);
}
```

Если нет ни одного перехода, либо все условия переходов не выполнены, то работа автомата завершается. Начинается обработка следующего события, или, если событий нет, поток автомата приостанавливается до их получения.

Дальнейшая разработка

В настоящий момент доказана работоспособность идеи (написана библиотека, она с успехом применяется в небольшом проекте). Для того чтобы использование такого рода инструментария было по-настоящему удобным, необходим редактор графа переходов автомата логики интерфейса. В настоящий момент автором ведется разработка такого редактора, как только он будет создан – появится первый продукт, основанный на идее создания логики интерфейса при помощи автоматов.

В дальнейшем предполагается разработка дополнений для популярных сред разработки приложений на *Java* (*IntelliJ Idea*, *Eclipse, NetBeans*). Поскольку идея достаточно проста, то автор надеется на то, что она приживется и позволит улучшить текущую ситуацию с «интерфейсостроением» в программировании.

Подход предполагает создание общей библиотеки шаблонов для повторного использования. Возможны как шаблоны автоматов, реализующих ту или иную функциональность, так и шаблоны состояний, переходов, групп состояний и так далее. Все это позволит еще более ускорить процесс разработки интерфейса, улучшив при этом его качество. Также это позволит создать наряду со стандартным набором «компонентов» для разработки интерфейса стандартный набор «состояний» для разработки логики интерфейса.

Выводы

Плюсы:

- Качество. Поскольку подход предполагает повторное использование логики поведения интерфейса, можно потратить больше времени на отладку отдельных частей.
- Скорость разработки. Это также следует из повторного использования, плюс из того, что часть «стандартного» кода создается автоматически.
- Удобство разработки. Разработка ведется (автор надеется, что в будущем будет именно так) в специализированном редакторе, который удобен. В данном контексте можно упомянуть девиз разработчиков *IntelliJ Idea* “*Developing for pleasure*”.
- Автоматизированность. Стандартный код создается автоматически, каждый раз заново приходится переписывать лишь уникальный для проекта код.
- Меньшие требования к уровню разработчиков. Поскольку сложная логика переходов из состояния в состояние в автоматном виде легко описывается в документации, наряду с действиями, производимыми внутри состояний и переходов, кодирование можно поручить менее квалифицированному разработчику, который сможет выполнить работу быстро и качественно.

Минусы:

- Некоторое влияние на скорость. Ничего не дается бесплатно. Естественно, автоматная обработка занимает некоторое время. Но, по сравнению со скоростью получения событий (действия пользователя) это время ничтожно. Следовательно, автомат все-равно будет большую часть времени проставивать и конечный пользователь ничего не заметит, с его точки зрения программа будет работать так же, как и ранее.
- Необходимость создания библиотеки компонентов. Также как и создания библиотеки работы с «автоматным» интерфейсом для различных языков, операционных систем и так далее. Но такая библиотека создается очень быстро, проста в использовании и поддержке.

Заключение

Метод, представленный в статье, кардинально отличается от тех методов, которые применяются сейчас для создания логики пользовательского интерфейса. Он позволяет в конечном счете улучшить качество конечного интерфейса и его исходного кода, ускорить разработку и упростить поддержку. Метод основывается на подходе, активно пропогандируемом Анатолием Абрамовичем Шалыто [1] и использует большую часть достоинств автоматного метода программирования.

В настоящий момент разработка библиотеки не прекращена, ведется активная доработка как теоретических основ, так и программных компонент.

Литература

1. http://is.ifmo.ru, СПбГУИТМО кафедра «Технологии программирования».