Интеграция программной логики с пользовательским интерфейсом при реинжиниринге приложений

М.А. Мосиенко maxim@tercom.ru
А.Е.Тиунова aet@tercom.ru

Санкт-Петербургский государственный университет
198504, Университетский пр., 28
Санкт-Петербург, Россия

Аннотация

В статье описывается опыт создания средств автоматического перевода приложений на новые платформы с преобразованием текстового (text-based) пользовательского интерфейса в графический (GUI) в проекте RescueWare. Предлагаемый подход был опробован при переводе Cobol/CICS и Unisys Cobol/DPS приложений в Java/Html клиент-серверную архитектуру, однако он может быть распространен и на перевод приложений в других сходных системах.

Введение

В мире существует большое количество приложений, написанных несколько десятилетий назад, которые работают и по сей день. Сопровождение и модернизация таких приложений становятся все более трудоемкими и дорогостоящими, поскольку уменьшается число специалистов, знакомых с необходимым окружением — это и языки программирования (COBOL, PL/I), на которых написано большинство старых приложений, и операционные системы (OS/370, OS/390, AS/400, Unisys 2200), и созданные для них менеджеры транзакций (transaction servers — CICS, IMS).

Многие компании предпринимают попытки перенести существующее программное обеспечение на современные платформы — Windows, Java, Html. Создание новой системы "с нуля" обычно затруднено огромными объемами программного кода (нередко это сотни и тысячи модулей), в котором нужно сначала разобраться для выявления бизнес-логики, а затем запрограммировать заново в выбранной архитектуре.

Поэтому постоянный интерес вызывают методы автоматического перевода старых приложений на новые платформы. Исследованию проблем, возникающих при переводе приложений, имеющих пользовательский интерфейс, а также возможным методам их решения и посвящена данная статья.

Материалом для данной статьи является опыт разработки системы автоматизации реинжиниринга RescueWare, которая включает элементы перевода пользовательского интерфейса. При этом основное внимание уделяется вопросам интеграции интерфейсных компонент с бизнес-логикой, а также общей архитектуре целевой системы. Другие аспекты перевода собственно интерфейсных компонент и программных модулей оставлены за рамками данной публикации.

1 Существующие подходы

Поскольку задача перевода приложений является весьма актуальной, она широко исследуется во всем мире, создаются средства автоматизации различных этапов выполнения данной задачи.

В зависимости от целей выполняемого перевода интерфейса архитектура создаваемой системы, а значит, и сложность самого перевода, может варьироваться от простой эмуляции терминала 3270 на персональном компьютере с подключением к основному приложению на мэйнфрейме до полного переноса приложения на новую платформу. Преимущества и недостатки различных подходов подробно рассмотрены в [1] на примерах известных программных продуктов. В работе [5] приводится классификация разновидностей перевода на основе сравнения возможностей построения интерфейсов в исходной и целевой системах, а также краткий обзор основных трудностей, возникающих при переносе интерфейса между системами с различными возможностями.

Существует несколько направлений, в которых развивается преобразование пользовательского интерфейса. Интересный подход к построению собственно графического интерфейса предложен в [2]. Авторы создали систему распознавания диалоговых элементов по описаниям исходных экранных форм на основе применения шаблонов. Набор шаблонов настраивается на конкретное приложение, что позволяет получить максимальную отдачу при последующей автоматической генерации нового интерфейса.

Большой популярностью пользуются методы подсоединения нового интерфейса по протоколу обмена с терминалом — создание так называемых screen scrapers (см. [1]), являющихся эмуляторами терминала. Данный метод позволяет достаточно быстро обеспечить доступ с персонального компьютера на мэйнфрейм, в том числе через Internet, и даже заменить при этом интерфейс с текстового на графический, оставив код исходного приложения на мэйнфрейме нетронутым. В [3] описывается опыт перевода DOS-приложения в Windows с минимальными изменениями исходного кода путем "удаленного управления" (remote controlling), при котором основная программа (оставшаяся программой на COBOL) взаимодействует с интерфейсными модулями (реализованными заново с помощью системы S-Prog) через специальный буфер, содержащий образ экрана.

Более сложные подходы, описанные в [5, 4], кроме преобразования собственно интерфейса включают в себя также выявление интерфейсно-зависимых фрагментов программной логики и перенесение этой логики в целевой интерфейс. Выявление фрагментов может производиться как вручную, так и с помощью различных автоматизированных средств — от поиска по шаблонам до анализа потоков управления и построения диаграмм переходов между экранными формами и программами.

2 Задачи, проблемы и решения

Рассмотрим задачу перевода на новые платформы приложений с пользовательским интерфейсом более подробно. В этом разделе мы сформулируем основные задачи, возникающие при переводе таких приложений, перечислим связанные с ними трудности, а также рассмотрим возможные методы их преодоления.

2.1 Требования к создаваемой системе

Основной целью перевода приложения на новую платформу является снижение затрат на его сопровождение, поэтому на создаваемое приложение накладываются дополнительные условия. Оно не просто должно работать “как старое”, но и порождаемый код должен быть интуитивно понятным, с легко прослеживаемой логикой, похожий на код, который мог быть написан квалифицированным программистом на целевом языке. Поэтому обычные методы, используемые при трансляции программ в машинный код, не всегда применимы при переводе приложений на другой язык высокого уровня.

К эффективности самого процесса перевода также предъявляются определенные требования. Если генерация кода для одной программы будет занимать несколько часов, перевод приложения из тысячи модулей вряд ли будет возможен. Также необходимо, по возможности, минимизировать взаимозависимость отдельных частей приложения, чтобы при необходимости повторения перевода одной из программ не нужно было генерировать заново все приложение.

При переводе больших приложений может оказаться привлекательным метод постепенного перевода — сначала в относительно короткий срок получить *работающее* приложение на новой платформе, а потом продолжить модернизацию уже в новом приложении без потери его работоспособности. Модернизация может быть связана как с изменениями бизнес-процесса, так и с улучшением кода, полученного в результате первоначального перевода. При этом улучшения могут выполняться как путем повторного перевода отдельных модулей с изменением параметров перевода и использованием возможностей редактирования системы автоматизации реинжиниринга, так и путем изменения полученного кода вручную или с помощью имеющихся средств рефакторинга целевого языка, если эти улучшения нельзя получить в процессе автоматического реинжиниринга.

2.2 Проблемы перевода

Основную сложность при переводе приложений, имеющих пользовательский интерфейс, составляют серьезные отличия в архитектуре таких программ в старых и современных системах.

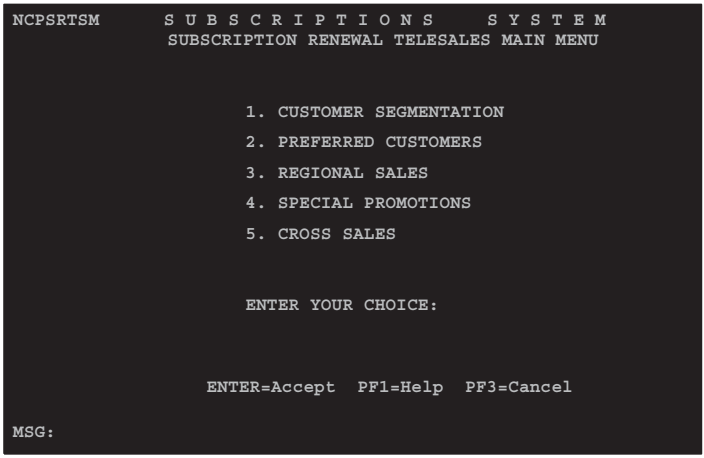

Рис. 1: Экранная форма на мэйнфрейме

Рассмотрим наиболее существенные их аспекты, которые необходимо учитывать при переводе приложений.

- *Клиент и сервер.* В старой системе приложение было чисто серверным, удаленный терминал лишь отображал данные, полностью подготовленные на сервере (включая всевозможные атрибуты — цвет текста, позиция курсора и т.п.). В новой системе чаще всего требуется создать архитектуру клиент-сервер, в которой разные части приложения работают на разных компьютерах. В зависимости от распределения задач между клиентом и сервером детали реализации могут несколько различаться: от менее трудоемкого “тонкого” клиента, служащего только средством ввода данных (аналог удаленного терминала), к “толстому” клиенту, при котором сервер является только сервером базы данных.

- *Исходный текст.* В современной системе формирование образа экрана и обработка пользовательских действий обычно находятся в одной программе, в старых системах экраны описывались отдельно от программы в файлах экранных форм — BMS (рис. 8), MFS, DPS, AS400 display files и др. При этом связывание программы с экраном происходило во время выполнения программы. Более того, опе-

раторы экранного обмена с терминалом могли работать не только со статическим именем экрана, но и с переменным, когда имя экрана, который должен быть отображен, определялось только в момент выполнения данного оператора (рис. 3)[1].

- *Элементы экрана.* Современные системы программирования позволяют формировать экраны из набора высокоуровневых элементов управления — полей ввода, меток, кнопок и др., каждый из которых имеет методы, позволяющие разрешать и запрещать редактирование, менять цвета текста и фона, скрывать элементы управления во время исполнения программы. В старых системах все поля были текстовыми (рис. 1), и все управление редактированием экрана осуществлялось путем установки разнообразных специальных атрибутов. Обмен с терминалом осуществлялся посредством пересылки "образа экрана", описывающего как собственно данные, так и атрибуты каждой позиции на экране. При компиляции экранной формы формировался объектный модуль, содержащий всю статическую информацию об экране, а также "символический" модуль (рис. 7), который после подключения к программе позволял подготавливать и обрабатывать данные для экранных операций в терминах полей для редактирования.

- *События.* В старой системе для завершения редактирования экрана пользователь должен был нажать одну из функциональных клавиш — `PF1-PF24`, `PA1-PA3`, `ENTER` или `CLEAR`, после чего терминал пересылал данные на сервер. Специальные системные средства позволяли из программы узнать код нажатой клавиши и, в зависимости от него, выполнить необходимую обработку данных. В последней строке экрана обычно выводилась подсказка пользователю о назначении функциональных клавиш на данном экране, а защита от нажатий непредусмотренных клавиш вставлялась в собственно код программы. В современной системе экранная форма обычно имеет элементы управления — меню и кнопки, по активации которых происходит вызов специфицированного программистом обработчика конкретного действия

[1] Здесь и далее примеры приведены для системы COBOL/CICS.

пользователя (например процедуры типа MyButton_Click() в Visual Basic).

- *Выполнение программы.* Операционная система и менеджер транзакций принимали активное участие в организации функционирования приложения, особенно в псевдоразговорной модели работы (которая описана далее), и часть этой поддержки необходимо включить в программный код новой системы.

2.3 Уточнение задачи

Конечно же, одним из вариантов перевода системы на новую платформу является трансляторный подход — полная эмуляция работы старой системы в части взаимодействия с экраном, включая поддержку всех атрибутов, операции с "образом экрана" и т.п. Однако сопровождение такой системы ничуть не проще, чем исходной. Кроме того, мы заинтересованы в создании модели перевода, рассчитанной не на какую-то конкретную платформу (например COBOL/CICS), а позволяющей переводить приложения с разных платформ, имеющих сходные черты. Поэтому этот вариант мы далее рассматривать не будем.

Учитывая вышеизложенное, задачу перевода приложения с пользовательским интерфейсом на современную платформу можно разделить на следующие подзадачи:

- приведение экранных форм к современному виду — с использованием полей ввода, меток, кнопок, меню и т.д.; при этом желательно предоставить пользователю некоторые возможности редактирования и влияния на генерируемые целевые экранные формы;

- создание модели взаимодействия процессов в целевой архитектуре клиент-сервер, позволяющей обеспечить функционирование приложения при наличии нескольких (многих) одновременно работающих пользователей;

- создание программной архитектуры целевого приложения, позволяющей обеспечить взаимодействие между бизнес-логикой и экраном в терминах поименованных полей, а не образа памяти терминала, в том числе и для управления атрибутами полей;

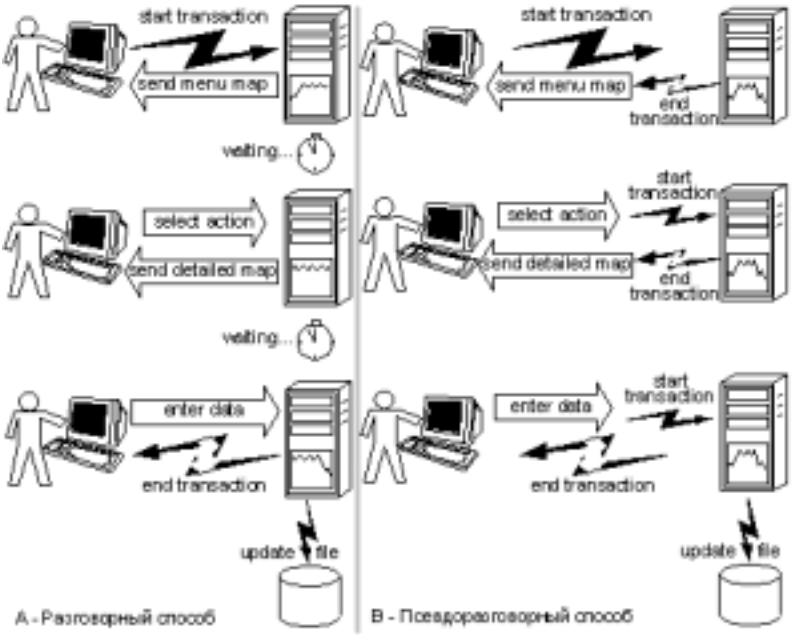

Рис. 2: Разговорное и псевдоразговорное взаимодействие

- создание механизма обработки пользовательских действий, позволяющего инициировать соответствующие части исходной программы по нажатию кнопок или выбору элементов меню.

2.4 Взаимодействие клиента и сервера

Старые системы использовали два способа организации взаимодействия программы с пользователем — разговорный (conversational) и псевдоразговорный (pseudo-conversational).

При *разговорном* способе (рис. 2, *A*) программа полностью контролирует взаимодействие, которое происходит по принципу "вопрос программы — ответ пользователя". Все время, пока пользователь подготавливает вводимые данные, программа остается в памяти компьютера, а после нажатия управляющей клавиши продолжается с оператора, непосредственно следующе-

го за оператором вывода данных на экран. Такой способ взаимодействия эффективен, если время ответа пользователя существенно меньше времени, затрачиваемого компьютером на обработку ответа.

При *псевдоразговорном* способе (рис. 2, *В*) внешне все выглядит так же — пользователь видит перед собой некоторую форму, вводит данные и через какое-то время получает ответ системы, вводит новые данные и т.д. — однако внутри все устроено по-другому. Пользователь инициирует программу, она запускается, выводит на экран первую форму и завершает работу. При этом в системе сохраняется информация о том, какую программу следует запустить после ответа пользователя (чаще всего это та же самая программа). После того как пользователь ввел данные и нажал управляющую клавишу, система инициирует указанную программу, программа читает данные с терминала и обрабатывает их в зависимости от того, какая функциональная клавиша была нажата, после чего выводит на экран новую форму и опять завершается. Для передачи информации между запусками программы используется специальная системная область памяти, в которую перед завершением программа может поместить информацию о своем состоянии, а после повторного запуска — прочитать ее. Данный способ эффективен в системах, где время ответа пользователя существенно превышает время обработки его запроса, поскольку программа не занимает ресурсов системы, пока пользователь думает.

Примечание. В обоих случаях время передачи данных между компьютером и терминалом включается во время ответа пользователя.

Эти же два способа взаимодействия можно применять и в целевой архитектуре клиент-сервер. Рассмотрим их преимущества и недостатки.

Разговорный способ:

- не масштабируем (требует выделения отдельного потока на каждого клиента);
- сложнее в реализации и сопровождении, требует наличия системы тайм-аутов для предотвращения зависания системы;
- включает в себя псевдоразговорный.

Псевдоразговорный способ:

- масштабируем;
- проще в реализации и сопровождении;
- не поддерживает приложения в разговорном стиле.

Мы выбрали для реализации псевдоразговорный способ работы как более эффективный при большом количестве клиентов, а также более подходящий для организации серверного приложения как набора независимых сервисов, реализующих различные бизнес-правила.

Заметим также, что при необходимости перевода приложения, работавшего в разговорной системе, его можно сначала преобразовать к псевдоразговорному виду, после чего осуществить перевод в указанную архитектуру. Для преобразования программы необходимо вставить перед операторами чтения данных с экрана операторы завершения программы с сохранением параметров состояния, а в начало программы добавить код обработки состояния с переходом на соответствующие операторы чтения.

2.5 Динамическое обращение к экранам

В некоторых приложениях используются операции с экранами, динамически определяемыми в процессе выполнения программы (рис. 3). Более того, в некоторых системах (например Unisys DPS) обращение к экрану происходит посредством вызова специальных программ с предопределенным набором параметров, и формы записи со статическим именем экрана просто не существует.

Если архитектура создаваемой целевой системы поддерживает только статические обращения к экранам, необходимо создавать специальные анализирующие средства, позволяющие для каждого вызова определить, какие экраны в нем могли использоваться, а затем преобразовать вызовы в статическую форму (рис. 4). В большинстве реальных приложений имя экрана определяется однозначно или принимает очень ограниченное число значений (как в примерах на рис. 3), однако возможны и более сложные ситуации (например, если это имя передается параметром из вызывающей программы), а в худшем случае список всех

```
05 WS-MAP-NAME                    IF WS-FIRST-TIME THEN
   PIC X(08) VALUE 'NCPMAP1'.        MOVE 'NCPMAP1' TO WS-MAP-NAME
                                  ELSE
...                                  MOVE 'NCPMAP2' TO WS-MAP-NAME
                                  END-IF

EXEC CICS RECEIVE                 EXEC CICS RECEIVE
   MAP     (WS-MAP-NAME)             MAP     (WS-MAP-NAME)
   INTO    (NCPSTLI)                 INTO    (NCPSTLI)
   RESP    (EIBRESP)                 RESP    (EIBRESP)
END-EXEC                          END-EXEC

A - значение однозначно              B - значение неоднозначно
```

Рис. 3: Обращение к экрану по динамическому имени

```
if (wsMapName == "NCPMAP1")   Ncpstli.sendNcpmap1();
else if (wsMapName == "NCPMAP2")  Ncpstli.sendNcpmap2();
else if (wsMapName == "NCPMAP3")  Ncpstli.sendNcpmap3();
```

Рис. 4: Преобразование динамических вызовов к статическим

возможных значений совпадает со списком всех экранов приложения.

Если же создаваемая архитектура позволяет использовать динамическое обращение к экранам, то создание таких анализирующих средств можно отложить, а в случае неоднозначности значения — избавиться от громоздких переборов всех возможных вариантов в генерируемом коде.

2.6 Связывание бизнес-логики и интерфейсных компонент

Типичное приложение клиент-сервер состоит из двух частей — серверные модули, реализующие бизнес-логику, и интерфейсные компоненты. Поскольку экранные формы раньше описывались отдельно от программ, связывание интерфейсных компонент с серверными модулями не является тривиальным и может быть вынесено на отдельный уровень — коммуникатор (рис. 5). Рассмотрим его более подробно.

В старых системах взаимодействие между программой и

Рис. 5: Архитектура целевого приложения

```
EXEC CICS RECEIVE          EXEC CICS SEND             EXEC CICS RECEIVE
  MAP    ('NCPSRTSL')        MAP    ('NCPSRTSL')        MAP      (WS-MAP-NAME)
  INTO   (NCPSTLI)           FROM   (NCPSTLO)           INTO     (NCPSTLI)
  RESP   (EIBRESP)           RESP   (EIBRESP)           RESP     (EIBRESP)
END-EXEC                   END-EXEC                   END-EXEC
```

A – Чтение экрана B – Запись экрана C–Чтение по динамич.имени

Рис. 6: Операторы чтения и записи

экраном происходит посредством набора системных вызовов (рис. 6), специфичных для каждого менеджера транзакций. Одним из основных параметров вызовов является "символический" модуль — промежуточная структура, служащая для передачи данных между программой и терминалом (рис. 7). Кроме того, в вызовах используются и переменные системного окружения, доступные программе (например, для CICS это системные переменные `EIBRESP, EIBAID, EIBCPOSN` и др.).

Структура символического модуля зависит от менеджера транзакций, и для каждого из них она фиксирована.[2] Поэтому, зная используемый менеджер транзакций, по символическому модулю можно определить набор полей экрана, предполагаемых в операторе работы с терминалом, а также установить соответствие переменных символического модуля полям и их атрибутам. В приведенном примере (рис. 7) можно идентифицировать поле `NCPSTL-ACCOUNT1-I` для чтения, это же поле используется для записи под именем `NCPSTL-ACCOUNT1-O`; переменные `NCPSTL-ACCOUNT1-L, NCPSTL-ACCOUNT1-F` представляют атрибуты этого поля при чтении, а `NCPSTL-ACCOUNT1-A` — атрибуты при записи.

С другой стороны, взаимодействие терминала с программой, как уже отмечалось, происходит в старых системах посредством

[2]Для некоторых менеджеров транзакций возможны несколько вариантов символического модуля, но их количество очень ограничено — так, в CICS их два, в зависимости от того, используются ли расширенные атрибуты терминала.

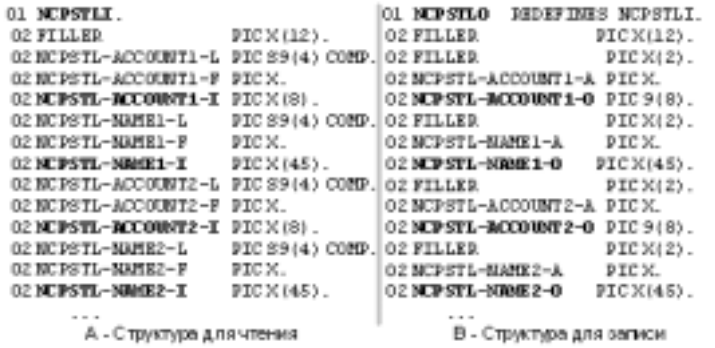

Рис. 7: Промежуточные структуры для чтения и записи

```
 ...
ACCOUNT1 DFHMDF LENGTH=008,POS=(009,002),                C
                PICOUT='99999999',                       C
                ATTRB=(NUM,PROT,NORM)
NAME1    DFHMDF LENGTH=043,POS=(009,014),                C
                ATTRB=(PROT,NORM)
ACCOUNT2 DFHMDF LENGTH=008,POS=(010,002),                C
                PICOUT='99999999',                       C
                ATTRB=(NUM,PROT,NORM)
NAME2    DFHMDF LENGTH=043,POS=(010,014),                C
                ATTRB=(PROT,NORM)
 ...
```

Рис. 8: Фрагмент описания экранной формы

пересылки памяти, и хотя файлы описания форм также содержат имена полей, они часто не совпадают с именами в программе. В исходной системе важен лишь факт наличия имени у поля экрана, поскольку только поименованные поля попадают в символический модуль, а не само имя. В нашем примере (рис. 8) соответствующее поле экрана называется `ACCOUNT1`.

Кроме того, имена экранов могут быть переменными (рис. 6, *C*).

В нашей реализации интерфейсная компонента представляет собой объект, свойствами которого являются поименованные поля, каждое из которых имеет набор атрибутов. Возможные типы атрибутов зафиксированы, поскольку каждый атрибут требует динамической поддержки. У объекта интерфейсной компоненты есть также две операции: чтение с экрана и запись на экран. По-

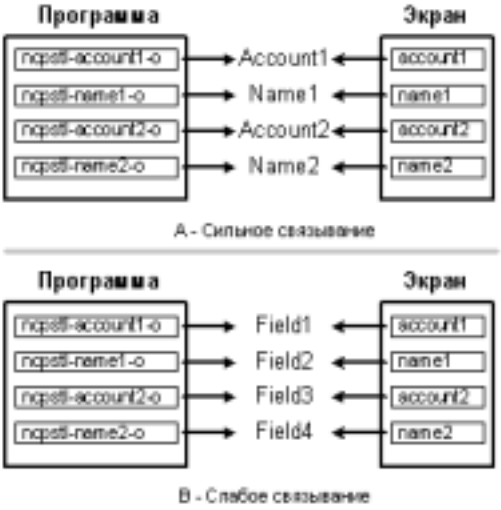

Рис. 9: Связь интерфейсной компоненты и программы

сле выполнения операции чтения содержимое полей ввода, набранных пользователем, а также идентификатор действия (например код нажатой клавиши) становятся доступными для ее клиента.

Серверная компонента, в свою очередь, работает со свойствами объекта, представляющего символический модуль, и использует этот объект при обращении к интерфейсной компоненте. Таким образом, задачей коммуникатора является установление соответствия между свойствами интерфейсной компоненты и символического модуля.

Можно выделить два вида отношения интерфейсной компоненты с бизнес-логикой по именам:

- *сильное* (рис. 9, A) — имена полей интерфейсной компоненты совпадают с именами полей в программе, идентификатор компоненты должен быть известен статически;
- *слабое* (рис. 9, B) — поля в программе и интерфейсной компоненте связаны порядком следования, идентификатор компоненты может определяться динамически.

Если использовать сильное отношение, то при трансляции

```
for(int rowIndex=0;rowIndex<10;++rowIndex) {
    screen.setField(nextScreenFieldName(),
                    getName0(rowIndex+1)));
    screen.setField(nextScreenFieldName(),
                    getAccount0(rowIndex+1)));
}
```

Рис. 10: Код для доступа к элементам списка

экранов и программ можно сгенерировать хорошо сопровождаемый код, обеспечивающий их взаимодействие. Это особенно удобно при создании локальных приложений или приложений клиент-сервер с "толстым" клиентом, когда и интерфейс, и бизнес-логика реализуются на одном языке программирования (например Java или Visual Basic), поля экранной формы действительно существуют и с ними можно явно работать:

```
str = NCPSRTSM.Inst.choice.getText();
```

Чтобы обеспечить совпадение имен свойств при генерации компонент, связанных сильным отношением, необходимо иметь таблицы соответствия полей экранов и программ для каждой пары "экран-символический модуль", встречающейся в приложении, поэтому данный способ применим только если все имена используемых экранов известны статически.

В случае слабого отношения для генерации кода взаимодействия приходится переименовывать поля при генерации программ и экранов посредством единой нумерации. В некоторых случаях именование полей может происходить даже динамически во время работы серверного модуля, а не во время его генерации, например при обращении к списку записей на экране (рис. 10), промежуточная структура для которого представляет собой массив. Это несколько усложняет получающийся код, но, с другой стороны, если целевой средой приложения является Internet и экраны переводятся в HTML или другой аналогичный формат, то поля в них существуют виртуально и косвенность обращения неизбежна, что ослабляет негативный эффект использования слабого способа обмена на сопровождаемость приложения.

При использовании слабого отношения интерфейсные компоненты могут генерироваться независимо от серверных модулей,

никаких таблиц соответствия имен не требуется, что позволяет применять этот способ при наличии обращений к экранам по динамически формируемым именам.

Мы решили существенно снизить зависимость целевого приложения от разных способов адресации полей путем сокрытия их внутри классов, которые порождены из символических модулей, участвующих в операциях работы с экранами. Эти классы наряду с обычными свойствами классов данных имеют также методы работы с экраном (чтение и запись), внутри которых и происходит обращение к интерфейсной компоненте. Именно эти методы работы с экраном и выполняют функции коммуникатора (рис. 5) между бизнес-логикой и интерфейсом.

Наша реализация была сделана в двух конфигурациях — клиент-сервер и полностью локальное приложение. В обоих случаях для серверных компонент использовалась Java, а клиентские части были переведены в Html/JSP и Java соответственно. Следует отметить, что благодаря инкапсуляции взаимодействия с экраном в классах данных реализация серверных программных модулей (собственно бизнес-логики) не зависит от того клиента, который будет с ними работать — можно сменить реализацию клиента, не трогая серверных модулей, в том числе заменить удаленного клиента локальным или сделать эти классы универсальными (рис. 11). Необходимо лишь, чтобы клиентская часть соблюдала принятый протокол взаимодействия с сервером.

Данный подход позволяет также упростить тестирование результирующего приложения, поскольку при замене интерфейсной компоненты на отладочный модуль можно перенаправить входные и выходные данные экранов во внешние файлы. Таким образом, становится возможным осуществить автоматическое регрессионное тестирование, что существенно облегчает дальнейшее сопровождение приложения.

2.7 Обработка событий

Рассмотрим возможные варианты организации обработки выбора пользователя (нажатой кнопки) в целевом приложении.

- *Полная эмуляция*[3] (рис. 12). На каждую экранную

[3]Целью эмуляции является обеспечение корректного функционирования сер-

```
public class Ncpsrtsmi {
  ...
  public void setNcpsrtsmChoiceI(int value) {...}
  ...
  public void receive() {
    String str, key;
    ...
    if (context.isRemote()) {
      ...
      screen.openReading();
      str = screen.getField("t1");
      key = (String)context.getAttribute("ACTION");
      screen.closeReading();
    } else {
      str = NCPSRTSM.Inst.choice.getText();
      key = NCPSRTSM.Inst.getAction();
    }

    setNcpsrtsmChoiceI(str);
    context.getDfheiblk().setEibaid(key);
  }
  ...
}
```

Рис. 11: Фрагмент класса для работы с экраном

форму добавляются кнопки, соответствующие функциональным клавишам мэйнфрейма — `PF1, PF2,..., PF24, ENTER, CLEAR`; строка подсказки о назначении клавиш остается без изменений. При нажатии одной из этих кнопок на сервер вместе с содержимым полей данных передается еще и системная переменная, содержащая код, соответствующий нажатой кнопке (для CICS это переменная `EIBAID`, принимающая значения `DFHPF1='1', DFHPF2='2',..., DFHENTER=QUOTE, DFHCLEAR='_'`). Этот способ достаточно прост в реализации, но вряд ли привлечет пользователей.

верной компоненты без внесения каких-либо изменений в код обработки полученной с экрана информации по сравнению с исходным текстом на COBOL. Это достигается путем имитации поведения менеджера транзакций (заполнение необходимых переменных системного окружения) при выполнении операций обмена с интерфейсной компонентой.

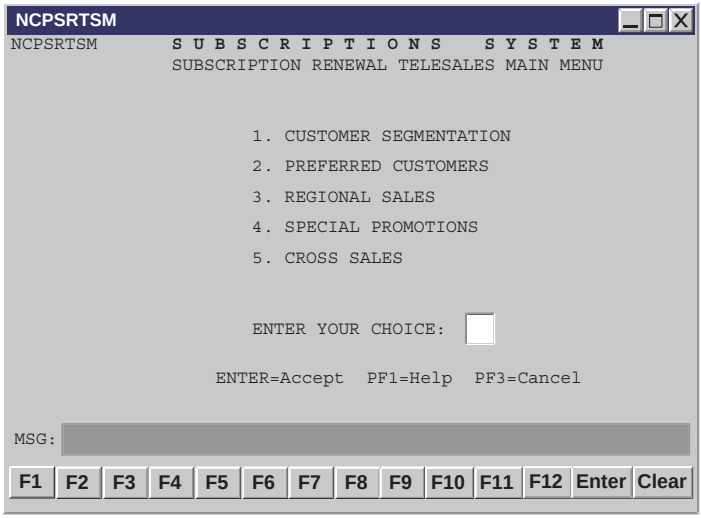

Рис. 12: Экранная форма при полной эмуляции

- *Частичная эмуляция* (рис. 13). Модификация предыдущего способа, при которой поле подсказки заменяется на набор кнопок с традиционными надписями, соответствующими выполняемым функциям, а на сервер передается строка, соответствующая нажатой функциональной клавише в старой системе (`'F1','F2',..., 'ENTER', 'CLEAR'`). Получив данную строку, программа (функция поддержки) преобразует ее в соответствующий код, присваивает его `EIBAID`, и все продолжает работать, как и раньше. Передача строки вместо кода клавиши позволяет уменьшить зависимость реализации интерфейсной части от исходной платформы (CICS, IMS, AS/400), в которых коды одних и тех же клавиш могут различаться, и спрятать эту специфику в серверный код поддержки.

- *Создание сервисов.* При данном подходе программа "разрезается" на части, соответствующие обработке одной из возможных нажатых клавиш по отдельности (рис. 14), каждая из частей преобразуется в самостоятельную программу-сервис. Обращение к такому сервису производится непо-

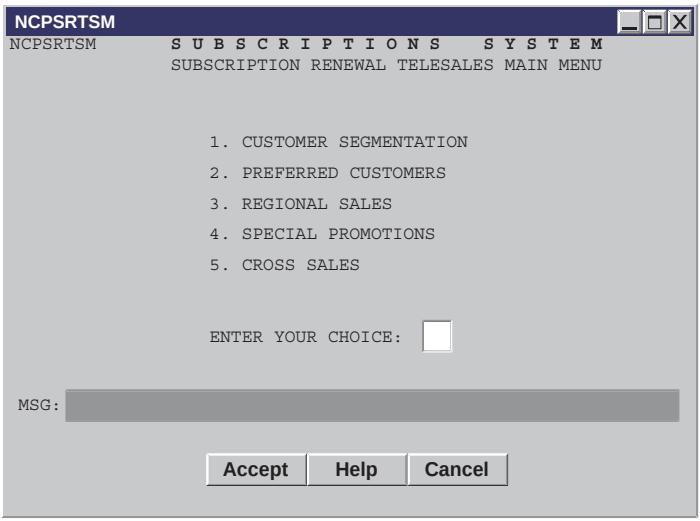

Рис. 13: Экранная форма при частичной эмуляции

средственно из клиентской части по нажатии соответствующей кнопки. Создание средств автоматического перевода, использующих данный способ, весьма трудоемко и требует также наличия мощных средств анализа программы. Более привлекательным является его применение при ручном или полуавтоматическом переводе приложений, когда основной скелет целевого приложения проектируется пользователем "с нуля", а средства автоматизации используются для извлечения знаний об исходном приложении и перевода его отдельных компонент.

Мы выбрали способ частичной эмуляции, позволяющий получить приемлемый результат в достаточно короткие сроки. С небольшими модификациями, данный метод позволяет не только перенести интерфейс на новую платформу, но и провести преобразование внешнего вида экранных форм (например экранов типа меню) к более традиционному виду (рис. 15) без потери работоспособности системы.

Заметим также, что при эмуляции необходимо устанавливать не только код нажатой клавиши (`EIBAID`), но и другие пере-

```
2000-RECEIVE-LIST-MAP SECTION.

EXEC CICS RECEIVE
   MAP      ('NCPSRTSL')
   INTO     (NCPSRTSLI)
   RESP     (EIBRESP)
END-EXEC.

IF EIBRESP NOT= DFHRESP(NORMAL)
   PERFORM 8000-CICS-ERROR.

IF EIBAID = DFHPF3
   PERFORM 7600-RETURN-MENU.

IF EIBAID = DFHPF7
   PERFORM  3000-READ-PREV.

IF EIBAID = DFHPF8
   PERFORM  3500-READ-NEXT.

IF EIBAID = DFHENTER
   PERFORM 1200-REFRESH-LIST.

IF EIBAID = DFHPF6
   PERFORM 1500-SHOW-DETAILS.
```

```
Private Sub Cancel_Click()
    Unload Me
End Sub

Private Sub Prev_Click()
    ReadPrev
End Sub

Private Sub Next_Click()
    ReadNext
End Sub

Private Sub
Refresh_Click()
    RefreshList
End Sub

Private Sub Edit_Click()
    ShowDetails
End Sub
```

Рис. 14: Создание обработчиков событий на Visual Basic

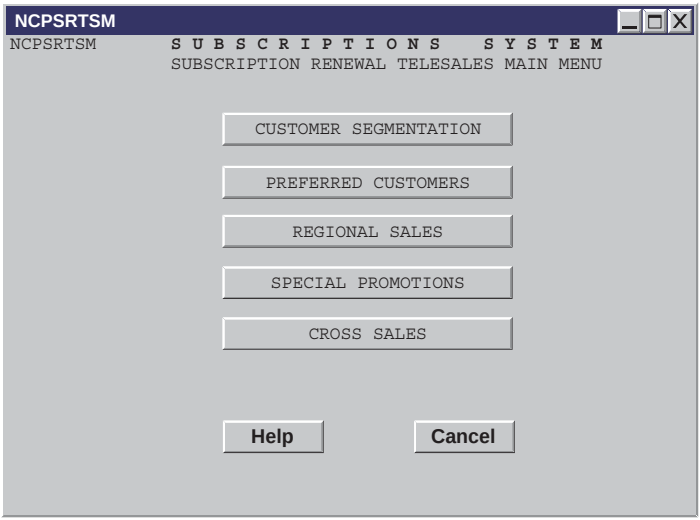

Рис. 15: Модифицированная экранная форма

менные системного окружения, использовавшиеся при работе с экранами (например `EIBCPOSN`).

Данный метод может использоваться в стратегии постепенного перевода приложения — после получения работоспособной системы на новой платформе любая из составляющих его программ, по мере необходимости, может быть заменена на набор сервисов. Реорганизация кода может быть проведена как вручную, так и с использованием полуавтоматических методов (Business Rules Extraction).

Следует заметить, что псевдоразговорные приложения хотя и выглядят монолитными программами, по сути значительно ближе к событийно-управляемому коду (сервисам), поскольку во время исполнения работают как обработчики нажатой функциональной клавиши, тогда как разговорные приложения требуют более существенной реорганизации для превращения в независимые сервисы.

2.8 Поддержка для других платформ

Как мы уже отмечали, нас интересует создание архитектуры целевого приложения, а также разработка средств автоматизации реинжиниринга, подходящих для перевода приложений, работающих на разных платформах. Поэтому после того как было найдено приемлемое решение для COBOL/CICS, мы попытались определить его пригодность для других платформ, оценить трудоемкость создания поддержки для них и возможность выделения каких-либо общих частей.

Первоначально может показаться, что вопросы взаимодействия программы с терминалом специфичны для каждого конкретного языка программирования и менеджера транзакций, и для каждой такой пары необходимо создавать независимые средства перевода. Однако в процессе работы, при более детальном ознакомлении с CICS выяснилось, что сама эта система является многоязыковой (существует, например, COBOL/CICS и ПЛ1/CICS) и практически все отличия программ ограничиваются особенностями синтаксиса CICS-операторов, накладываемыми используемым языком программирования. Это подтвердилось и сравнением программ на COBOL и PL/I. Таким образом, оказалось, что практически вся реализация автоматизирован-

ного перевода приложений, написанных для CICS на COBOL, годится и для приложений на PL/I (в части работы с терминалом), если параметризовать элементы взаимодействия с исходным языком.

Независимость от используемого языка характерна и для других менеджеров транзакций — например IMS и Unisys/DPS. Это и понятно: менеджер транзакций определяет архитектуру (скелет) программы, доступные свойства экранной формы и механизм их взаимодействия.

Можно пытаться обобщать взаимодействие программ с терминалом и дальше — различные системы, ориентированные на терминалы типа IBM 3270, должны обладать схожими чертами, определяемыми возможностями этого терминала, а значит можно пытаться выделить общие компоненты и при создании средств перевода для разных менеджеров транзакций.

3 Дальнейшее развитие

Конечно же, нам известны и некоторые направления дальнейшего развития как для автоматического, так и для полуавтоматического вариантов перевода приложений.

Прежде всего, для повышения эффективности полностью автоматического решения требуется использование анализирующих проходов, позволяющих собрать наиболее полную информацию о событиях (функциональных клавишах), с которыми работало исходное приложение. С одной стороны, более простым источником данной информации являются файлы экранных форм. Анализируя текст строки подсказки с использованием нескольких известных шаблонов, можно обнаружить большинство событий вместе с текстом, который нужно разместить на кнопках новой формы. Например, по строке

```
PF3=Cancel  PF6=Edit  PF7=Up  PF8=Down
```

можно определить, что использовались функциональные клавиши `F3, F6, F7, F8`, и заменить это текстовое поле на кнопки — `Cancel, Edit, Up` и `Down`. Однако иногда не все используемые клавиши перечислены в строке подсказки. Более точную информацию о событиях предоставляет программа, работающая

с экранной формой, поскольку для каждой используемой клавиши содержится соответствующий код обработки, но из него невозможно определить подходящую надпись для кнопки. Например, следующий код обрабатывает нажатие кнопки ENTER, не упомянутой на экране:

```
IF EIBAID = DFHENTER
   PERFORM 1200-REFRESH-LIST.
```

Другим необходимым элементом для получения законченного решения является создание диалоговых средств, позволяющих пользователю влиять на процесс перевода приложения — корректировать автоматически собранную информацию о событиях и связях между экранной формой и программой, добавлять новые события, а также управлять размещением элементов генерируемых экранных форм. Эти возможности особенно важны при проведении частичной реорганизации интерфейса, например при преобразовании экранов типа меню. Привлекательной может оказаться и возможность настроить генерацию интерфейсных компонент на определенный корпоративный стандарт — задать шрифты, добавить логотип компании и т.п.

При расширении данной модели на другие исходные платформы (IMS, AS/400) могут появиться новые требования или детали реализации.

Остальные направления развития относятся, скорее, к полуавтоматическому переводу приложения, возможность и целесообразность использования их в автоматическом решении требуют отдельного исследования.

Для повышения эффективности приложения и более оптимального распределения нагрузки между клиентом и сервером можно перенести первичную проверку введенных пользователем данных, не требующую запросов к базе данных, с сервера в клиентскую часть. Это также позволит уменьшить число обменов данными между клиентом и сервером в случае неверного ввода данных.

Для некоторых классов приложений, не использующих управление терминалом (атрибуты полей, позиционирование курсора), можно пытаться ликвидировать промежуточную структуру данных (символический модуль, связанный с экранной формой), установив соответствие полей экрана непосредственно с элементами данных, требующих отображения. Это

позволит несколько уменьшить объем целевого кода и упростить его понимание. С этой задачей связаны две другие, которые могут иметь и самостоятельную ценность — выявление управления терминалом и удаление лишнего кода.

Выявление управления терминалом позволяет создавать более понятный целевой код. Например,

```
MOVE -1 TO NCPSRTSM-CHOICE-L
```

при переводе на Java может быть заменено на

```
Ncpsrtsm.choice.requestFocus()
```

а

```
MOVE X'C9' TO NCPSRTSM-CHOICE-A
```

на

```
Ncpsrtsm.Choice.setForeground(Color.Red)
```

При этом существенно упрощается код поддержки, обеспечивающий динамическую интерпретацию атрибутов полей.

Удаление лишнего кода позволит упростить его понимание. Например, после перевода в новую архитектуру код, обрабатывающий нажатие неверной функциональной клавиши, никогда не будет исполняться. Также избыточный код может образоваться при изменении вида экранных форм, удалении полей и т.п. В некоторых случаях в процессе перевода образуются и лишние данные — например поля атрибутов полей экрана.

Как уже упоминалось выше, существенно отличающиеся результаты могут быть получены с разработкой методологии разбиения монолитного кода исходного приложения на набор независимых сервисов, предоставляющий те же функции. Для решения этой задачи необходимо не просто научиться извлекать логику, связанную с нажатием той или иной клавиши, но и создать средства для выявления и повторного использования кода, используемого в нескольких сервисах, иначе преимущество разбиения программы на части сводится на нет.

Выводы

Нами были исследованы и проанализированы основные проблемы, возникающие при переводе приложений, имеющих пользо-

вательский интерфейс, на новые платформы и затрудняющие создание средств автоматического перевода таких приложений, а также сформулированы некоторые требования к результирующему приложению и средствам автоматизации.

Рассмотрены различные подходы к переводу таких приложений, в том числе и к архитектуре целевого приложения, а также их вариации в зависимости от особенностей исходного и создаваемого приложений.

Рассматриваемая реализация опробована на нескольких пилотных проектах и признана перспективной, в настоящее время ведется доработка созданного прототипа до полноценной компоненты продукта RescueWare.

Мы считаем, что предложенное нами решение отличается от уже существующих комплексностью подхода — рассматривается перевод интерфейса и бизнес-логики в совокупности, в отличие, например, от работы [2], рассматривающей перевод только экранных форм. Подход позволяет достичь высокой степени автоматизации — для многих приложений работающий вариант целевой системы может быть получен без участия пользователя. Получающийся в результате программный код достаточно понятен и легче в сопровождении, чем эмуляторы терминала, рассматриваемые в [1, 3].

Таким образом, рассматриваемое решение позволяет в относительно короткий срок получить работающее приложение на современных платформах и языках программирования, что снижает стоимость его дальнейшего сопровождения по сравнению с исходным приложением. При этом решение не очень трудоемко в реализации, поскольку не требует создания мощных средств анализа программ, необходимых для подходов, описанных в [5, 4].

Список литературы

[1] Эрлих Л.А. Модернизация старых программ как ключевой фактор электронной коммерции // Автоматизированный реинжиниринг програм. — СПб.: 2000. — С. 20-42.

[2] Claben I., Hennig K., Mohr I., Shulz M. CUI to GUI Migration: Statis Analysis of Character-Based Panels // Proc. Software

Maintenance and Reengineering Conference. — 1997. — P. 144-149.

[3] Csaba L. Experience with User Interface Reengineering. Transferring DOS Panels to Windows // Ibid. — P. 150-156.

[4] Merlo E., Gagne P., Girard J. e.a. Reengineering User Interfaces // IEEE Software. — 1995. — Vol. 12. №1. — P. 64-73.

[5] Moore M., Rugaber S. Issues in User Interface Migration // Software Engineering Research Forum. — 1993.